*In Proc. Int'l Workshop on Distributed Object Management Edmonton, Canada, Aug. 1992.*

# Object Algebra and Views for Multi-Objectbases

**Marc H. Scholl***
University of Ulm
Department of Computer Science
D–W–7900 Ulm, Germany

**Hans-J. Schek**
ETH Zürich
Department of Computer Science
CH–8092 Zürich, Switzerland

**Markus Tresch***
University of Ulm
Department of Computer Science
D–W–7900 Ulm, Germany

## Abstract

This paper addresses the problem of defining views that span several objectbases. Views are expressed in terms of an object algebra. We are interested in the degrees of coupling and in the degrees of integrity maintained through multi-objectbase views. A key issue is the problem of global object identity. Each local system has its private object identifiers (OIDs) to represent objects. Globally, however, we need other mechanisms to uniquely refer to objects. We propose a technique that is based on the definition of (updatable) views: Queries to other objectbases are used as a referencing mechanism. The mechanism is useful as a formal basis for (partial or complete) schema integration in multi-objectbase systems.

## 1 INTRODUCTION

Object-orientation is supposed to offer advantages in the context of multi-database systems. The advanced modeling capabilities of object models provide generalizations and the possibility to include computations (in the form of methods) in schema definitions. This surely facilitates the interoperability of heterogeneous

*Work done while at Department of Computer Science, ETH Zürich.

data or object bases as far as schema integration is concerned [Kaul *et al.*, 1990; Saltor *et al.*, 1991]. Object query languages and algebras for objectbases reintroduce generic, general-purpose functionality to object management, similar to relational algebra [Straube and Özsu, 1990; Shaw and Zdonik, 1989; Kim, 1989; Cluet *et al.*, 1989; Bancilhon *et al.*, 1989].

Supporting views by allowing any query expression to serve as the definition of derived object collections (view classes) is the next logical step. We have been working on object preserving views in order to have a simple means for supporting updates. Object identity is the main feature used for it. Separation of types and classes helps to distinguish projection kinds of views from selection views [Scholl and Schek, 1990; Scholl *et al.*, 1991].

A further step is the application of the object algebra and of views for tasks which span several objectbases. Again we want to preserve the advantages of state-of-the-art commercial systems which already allow us to query and update several databases within one application/transaction (see for example ORACLE). But actually we want more support for integrity preserving updates in such a multi-objectbase scenario: instead of explicitly having to update related data in different local objectbases, we want a (partially) integrated global schema, such that the multi-objectbase system can automatically maintain inter-objectbase constraints. Further, we want to overcome the limitations of relational systems (w.r.t. view updates) and we do not want to lose the higher degree of integrity supported in OODBMS by the notion of objects, their types, classes and distinction between values and objects. Notice that we are not aiming at fully integrated global databases. Rather we want a flexible way to partially integrate existing databases while retaining their local autonomy.

The purpose of this paper is to extend the view definition mechanism that is already offered by the COOL language to serve as the formal basis for schema integration in multi-objectbase systems. We can apply the object algebra, which has been defined for a single objectbase so far, without major changes to multiple objectbases. In order to show this, the main questions that have to be answered are the following

1. What is the minimal degree of object schema integration in order to link several objectbases by algebra expressions? What kind of views can be used?
2. How do we define, cope with, and maintain the integrity that two objects from different objectbases are "identical" in the sense that they represent the same 'real-world' object?
3. Is there a way to obtain a spectrum of different degrees of integration of local objectbases; and what are the impacts on integrity maintenance during updates?

A lot of work has been devoted to the problem of schema integration (see [Batini *et al.*, 1986] for a survey); relatively few of them consider integration of existing objects, that is, populated objectbases. Exceptions are the Multibase project [Landers and Rosenberg, 1982] and Pegasus [Ahmed *et al.*, 1991; Krishnamurthy *et al.*, 1991]. These attack similar problems. However, Multibase is retrieval-only, and for Pegasus, we did not see precise definitions and maintenance of multi-objectbase identity, yet. Kent introduced a separation of real world entity objects vs. proxy objects [Kent, 1991]. The latter ones represent en-

tities in different objectbases. The next closest work we are aware of is that of Neuhold/Schrefl et al. [Schrefl, 1988; Neuhold and Schrefl, 1988; Kaul *et al.*, 1990; Geller *et al.*, 1991]. We will compare our approach with their work in more detail below.

Our approach differs from Pegasus and also from Schrefl/Neuhold in that we define (global) object id in terms of algebraic (extend) views. We show that, with a small revision of the notion of object identity, we can define views over multi-objectbases, that unify the proxy objects in different local objectbases, if they represent the same real-world entity.

The structure of the paper is as follows: Section 2 motivates the importance of views and subschemas in multi-objectbases and gives a summary of the COOL object algebra and its view definition mechanism. In Section 3 we describe how the algebra can define links between objects from different objectbases. A specialized linking mechanism is used in Section 4 to unify objects across multiple objectbases. The problem of update propagation is discussed in Section 5.

# 2 SCHEMAS AND VIEWS IN MULTI-OBJECTBASES

## 2.1 SCHEMA ARCHITECTURE OF MULTI-OBJECTBASES

Multi-objectbases are built up of several components – local objectbases (LOBs) –, each of which has the structure of its objects described by a local schema. The composite schema gives the structure of the global objectbase (GOB). Figure 1 shows this architecture, which is a part of the five-level schema architecture for federated DBS proposed in [Sheth and Larson, 1990]. Throughout this paper, we concentrate on homogeneous multi-objectbases. That is, we separate the issue of data model transformation from the rest, and assume that all schemas are already transformed into a uniform data model.

Views (virtual classes) and subschemas (collections of classes and views) play an important role within multi-objectbases [Motro, 1987; Abiteboul and Bonner, 1991]. For an LOB, the local schema can be tailored by adding local views to better meet external requirements (external schema). Afterwards, a subpart of the local schema (export schema) can be exported for use in the multi-objectbase system. In a GOB, views and subschemas can again be used to tailor the composite schema. In addition, the creation of views and subschemas that span several LOBs can be utilized as a formal basis for global integrity maintenance during updates, that is, to achieve a higher degree of integration.

The goal of the global schema is to allow queries and updates that involve several LOBs. The power of global transactions, in particular that of global updates, critically depends on the degree of integration of local schemas into a global schema. The tighter the integration is, the more powerful can integrity maintenance be. We distinguish the following levels of integration:

**Level 0** is the weakest level, where no integration of the LOB schemas is performed at all. Global transaction management allows us to query and update several LOBs within a single transaction. Each operation is explicitly directed to one

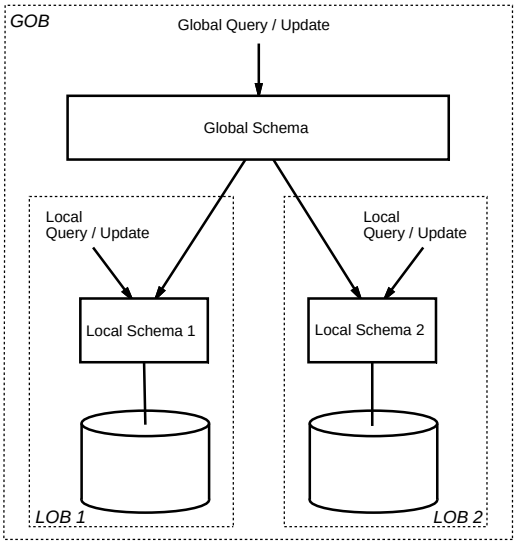


Figure 1: Schemas and Operations in Multi-Objectbase

LOB, it is not possible to address objects from multiple LOBs in one statement of the query language.

**Level 1** has the local schemas *composed*. Again, this is not yet a real integration. Simply the local schemas are put side-by-side, such that the name spaces of the local schemas are merged. Now, queries/updates can be performed on multiple LOBs (because the names of the classes and views are known to each other). However, as we will see later, objects are strictly separated.

**Level 2** finally supports real integration. Proxy objects from multiple LOBs must have been identified to represent the same entity object from the real world. Parts of the local schemas are unified as well. Now, real inter-objectbase queries/updates are possible. Updates propagate through multiple LOBs.

Our approach is to start using an object algebra in Level 1 to define the necessary parts to achieve Level 2. As a representative, we use the object model COCOON with the COOL language that is sketched below. However, this is just a representative to illustrate the basic principles. The global observations are valid for other models as well.

### 2.2 AN OBJECT MODEL AND ALGEBRA

Throughout this paper we use COCOON as the underlying object-oriented data model [Scholl *et al.*, 1992; Scholl and Schek, 1990]. Example 1 illustrates a definition of three COCOON databases, serving as the running example.

**Example 1** shows a case study of a possible multi-objectbase situation in a factory that produces screws and nails. The factory has several departments, each of which uses its own autonomous database (see Figure 2).

The design department is responsible for the design of new screws and nails. It works with the database *DesignDB* (or *DDB*, for short). The production department established a database called *ProductionDB* (or *PDB*) for information required for producing parts at the production branches. It also holds information about suppliers of raw materials. Finally, the sales department keeps in the *SalesDB* (or *SDB*) information on available articles and the company's customers.

```
define database DesignDB as
        define type item isa object = dno, matn: integer ,
                                      weight: kilo ;
        define type nail isa item = size: integer ;
        define type screw isa item = thread: string ;
        define class Items: item ;
        define class Nails: nail ;
        define class Screws: screw ;
end ;

define database ProductionDB as
    define type part isa object = cno, cplan: integer ,
                prodin: branch inverse produces ;
    define type branch isa object = bname, bstreet, bcity: string ,
                produces: set of part inverse prodin ;
    define type material isa object = mno: integer ,
                weight: pound ,
                supplby: supplier inverse supplies ;
    define type supplier isa object = sname, sstreet, scity: string ,
                supplies: set of material inverse supplby ;
    define class Parts: part ;
    define class Branches: branch ;
    define class Materials: material ;
    define class Suppliers: supplier ;
end ;

define database SalesDB as
    define type article isa object = ano, price: integer ,
                weight: kilo ,
                boughtby: set of customer inverse bought ;
    define type customer isa object = cname, cstreet, ccity: string ,
                bought: set of article inverse boughtby ;
    define class Articles: article ;
    define class Customers: customer ;
end ;
```

COCOON is an object-function model with the basic constituents *objects, functions, types, classes*, and *views*:

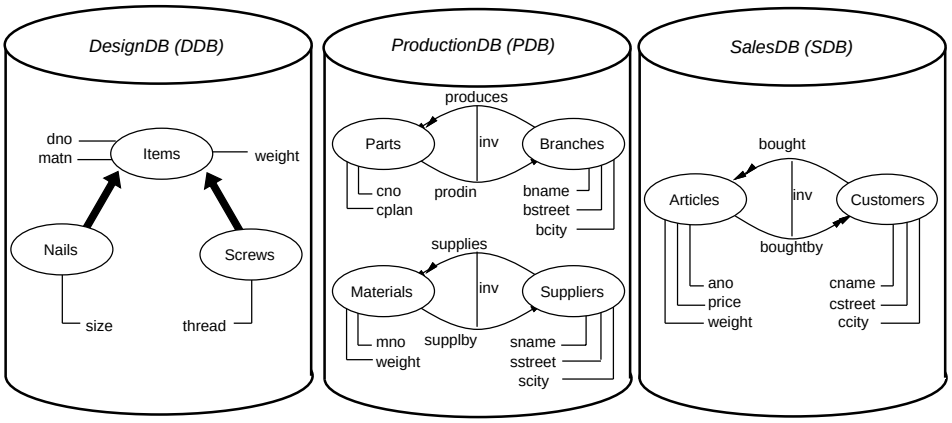


Figure 2: Multi-objectbase scenario for the running example

*Objects* are instances of abstract types, specified by their interface operations. *Data* are instances of concrete types (e.g., numbers, strings) or constructed types, such as sets.

*Functions* are either retrieval functions or update methods. They are described by their name and signature (that is, domain and range types). Functions may be set-valued.

*Types* are described by their name and the set of functions that are applicable to their instances. Types are arranged in a *subtype hierarchy*, where subtypes inherit functions from their supertypes. Objects are "instances of" types, possibly of more than one type at the same time ("multiple instantiation"). The top element of the type lattice is the most general type **object**. Therefore, all instances of any type in the database are also instance of **object**. The bottom element is the type **bottom**.

The language is strongly typed, in the sense that we support full static type checking. We can make use of type names as (type) predicates to check whether an object $o$ is instance of a specific type $T$ or not; e.g. *employee(p)* checks whether person $p$ is of type employee.

*Classes* are typed collections of objects ("type extents"). Classes are arranged in a *subclass hierarchy* that is defined by the set inclusion between the sets of objects they represent. Objects are "members of" classes, possibly more than one at a time ("multiple class membership"). Particularly, superclasses contain all members of their subclasses. The top element class is called **Objects**. All objects in the database are member of this class.

For each object class, its *membertype* is an associated type, and its *extent* denotes a set of objects of that type. An important additional feature distinguishes our model from others: class predicates. These are usually found in knowledge representation (classification) languages such as KL-ONE [Brachman and Schmolze, 1985; Borgida *et al.*, 1989]. Our classes may be constrained by a predicate that must be satisfied

by all members of the class. We distinguish two cases: class predicates may be only necessary or necessary and sufficient conditions.

We use a *set-oriented* query language similar to 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 generic update operations to results of a query, since these contain base objects.

View definitions introduce new (virtual) classes, whose extent is defined by the query. For views defined by each of the basic COOL operators, we describe what the membertype and extent is, and how these are positioned in the type and class hierarchies. For ease of presentation, assume that all views are defined over base classes. In general, views may also be defined over other views, or by composite queries (see [Scholl *et al.*, 1991] for a detailed exposition).

*Selection* ( **define view** $V$ **as select** $[P](C)$ ). The view class $V$ is a subclass of the base class $C$, with the same membertype as $C$. Notice, that we now have two classes, $V$ and $C$, of that type. The extent is the subset of $C$-members satisfying predicate $P$.

*Projection* ( **define view** $V$ **as project** $[f_1, \ldots, f_n](C)$ ). The view class $V$ is a superclass of $C$: The membertype, of $V$ is a supertype of the original (less functions are defined, only those listed in the projection), the extent of $V$ is the same as that of $C$.

*Extend* ( **define view** $V$ **as extend** $[f_i :=< expr_i >, \ldots](C)$ ). Projection eliminates functions, extend defines new derived ones. $< expr_i >$ can be any legal arithmetic, boolean, or set-expression. The view $V$ is a subclass of $C$: their extents are the same and the membertype of $V$ is a subtype of $C$'s (all the old functions plus the new ones are defined).

*Set operations* ( e.g., **define view** $V$ **as** $C1$ **union** $C2$ ). As the extent of classes are sets of objects, we can perform set operations as usual. With a polymorphic type system, we need no restrictions on operand types of set operations (ultimately, they are all instances of type **object**). The result type, however, depends on the input types: a union view is a common superclass of its base classes, where the membertype is the lowest common supertype (in the type lattice) of the input types. Difference views are subclasses of their base class with the same membertype; finally, an intersection view is a common subclass with a member type that is the greatest common subtype of the input types.

*Generic update operations.* Beyond type-specific update operations (i.e. methods), COOL provides a collection of generic update operators to facilitate set-oriented processing. First, there is a set-iterator for updates, **update**$[m](< set - expr >)$, that takes as argument a set of objects and an operation, $m$, to be performed on all elements. The other generic update operators are **insert** and **delete** for creation and destruction of objects, **add** and **remove** for including and excluding existing objects into/from sets (in contrast to **insert** and **delete**, **add** and **remove** have no effect on the existence of the objects), and **set** to assign return values to functions.

# 3 ESTABLISHING LINKS BETWEEN OBJECTBASES

In this section, we present a first approach to the interaction of objectbases that places only minimal requirements on the degree of schema integration between participating objectbases. First, the schemas are "composed", which basically just imports the names of schema elements from local objectbases. Second, we show that the COOL query language can be used to define inter-objectbase links by means of query expressions. This allows us to combine information from several sources (LOBs) into a global schema.

## 3.1 SCHEMA COMPOSITION

The least requirement that has to be imposed on two objectbases, in order to be used jointly, is that they make known to each other the names of their schema elements. That is, in the case of our object model, they have to export the names of their functions, types, and classes. For this purpose, we provide a *schema composition statement*, that lists the names of the participating objectbases. In contrast to schema integration, schema composition just makes names globally available, without really establishing any connection between the composed parts.

**Example 2** Suppose we want to work on the design and production objectbases in a global application. We define a composite schema by the following COOL statement:

**define schema** *CompositeDB* **as**
    **import** *DDB;*
    **import** *PDB;*
**end**;

A graphical representation of the composite schema is given in Figure 3.

We use the following naming conventions: names of local schema components are suffixed by "@" and the name of the local schema, for example, the class *Items* in the *DDB* has as globally unique name "*Items@DDB*", and similarly for types and functions. As a shorthand, when using generic names such as $LOB_i$, we often write, for instance, "$object_i$" instead of "$object@LOB_i$".

In general, the composition of (local) objectbases $LOB_i$ yields a global objectbase ($GOB$) where the two parts of the global schema, the type and class hierarchies, are combined in the following trivial way:

- The global type hierarchy consists of a new top element "*object@GOB*" that is the common supertype of the top elements of all component type hierarchies, $object_i$. Also, a new bottom element "*bottom@GOB*" is introduced as common subtype of the local bottom types. No other subtype relationships are established by schema composition. Therefore, the global schema is "integrated" at the very top level. No instance of "*object@GOB*" is instance of more than one component type $object_i$.
- The global class hierarchy has a new top element "*Objects@GOB*", which is common superclass of the local top classes, $Objects_i$. The extent of "*Objects@GOB*" is partitioned into disjoint subsets $Objects_i$.

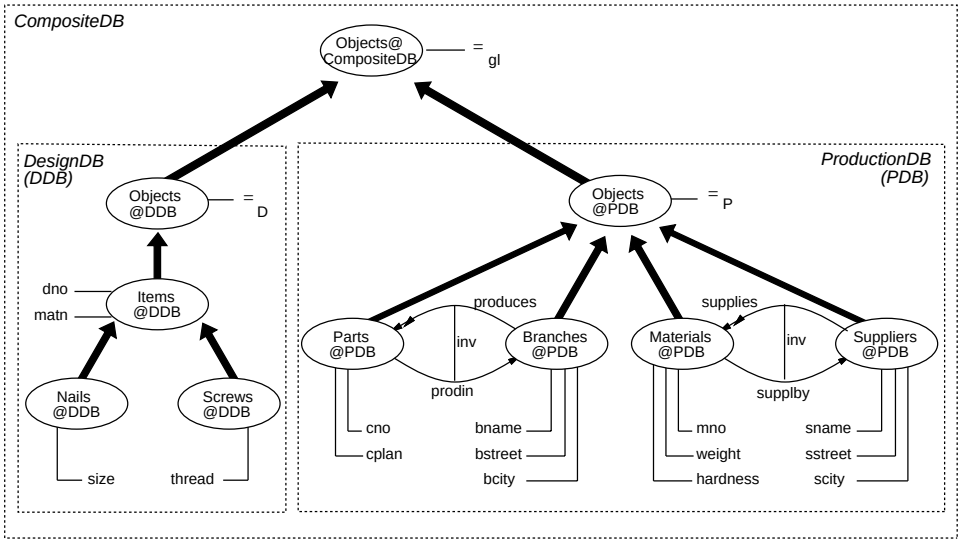


Figure 3: Composition of *DDB* and *PDB*

As a consequence, no two objects in "*Objects@GOB*" can be the same (identical), unless they originate from the same local objectbase, $LOB_i$, and are identical in $LOB_i$ (cf. Figure 3).

**Definition 1 (Global identity)** *Formally, the global object identity predicate '*$=_{gl}$*' on the new (global) type* object@GOB *is defined as:*

$$\begin{aligned} =_{gl} \quad &: \quad \text{object@GOB} \times \text{object@GOB} \rightarrow \textbf{boolean} \\ =_{gl}\ (o_1, o_2) \quad &= \quad \exists i : object_i(o_1) \wedge object_i(o_2) \wedge o_1 =_i o_2 \end{aligned}$$

In essence, with this first global schema we have not yet performed any "real" integration, but we do have a common basis for queries over composed schemas: a global parser can be provided with the (disjoint) union of names of schema elements from the component objectbases.

Despite the above basic assumption that schema composition just places the two (or more) participating schemas "side-by-side", notice that we do assume that the underlying *basic data types* (such as integer, boolean, string, ...) are unified.[1] Therefore, the global schema contains only one of each of these basic types. Hence, we can already compare the primitive data values contained in all component objectbases. This is a necessary prerequisite, since otherwise we could not even define the global identity predicate above (the unified base type **boolean** is used in the definition).

It is beyond the scope of this paper to prove that the object algebra defined on the component objectbases is now well defined also on the composed objectbase. In

[1] Conversion from one representation to another one are automatically provided.

order to prove this, we need to represent the global meta objects in terms of the local ones, see also later.

### 3.2 INTER-OBJECTBASE QUERIES

Once we have composed two (or more) local objectbases into a global schema, we are ready for formulating queries that span objectbases, that is, involve arguments from more than one LOB. In this subsection we analyze the possibilities and limitations of such inter-objectbase queries under the assumptions of the basic composition defined above. The analysis follows the structure of the COOL language, that is, we look at each COOL operator in turn.

#### 3.2.1 Connecting LOBs via Extend Operations

The extend operator of COOL allows us to define new, derived functions. The derivation is by a COOL query. As a result, the new function is available on the result type and is executed whenever it is used later. In the context of composite schemas, we can use **extend** to establish inter-OB links: suppose we apply an extend operation to a class from $LOB_1$ defining a derived function by a query on $LOB_2$.

**Example 3** In *CompositeDB*, let us define the following link from the *PDB* to the *SDB*: for each part we want to see the names of potential local customers, that is, the names of customers in the city where the part is being produced.

$$\textbf{extend}\ [\ plcn\ :=\textbf{extract}\ [cname] \\ (\textbf{select}[ccity(c) = bcity(prodin(p))](c : Customers))\ ] \\ (p : Parts)$$

The extend operation defines a new function *plcn* (for potential local customer names) by an external query: Parts is a class in the *PDB*, whereas Customers is one in the *SDB*. Notice this query is perfectly valid since the right-hand side of the function definition is executed in *SDB*: before that, however, '$bcity(prodin(p))$' is replaced by a constant, its function value from *PDB*. The selection predicate passes type checking, because basic data types (string, in the example) are unified by the composition of schemas.

The general principle behind this example is that in order for $LOB_i$ objects to refer to objects or values from foreign (i.e., other) objectbases $LO_j$, we use queries on $LOB_j$ to retrieve the information from the foreign LOB. Such queries are used from within $LOB_i$, that is, they can be "parameterized" with values from $LOB_i$. In our example above, the only parameter was the city address of the branch where the part is produced, $bcity(prodin(p))$.

Using queries as the external referencing mechanism seems very natural, since this is the way how all operations on any objectbase are expressed. The extend operator further lets us predefine query expressions as derived functions, such that later on we only have to call this function by its name. Since the function is executed once per reference to its name (at least conceptually), there are no problems with "dangling" references due to changes in the foreign objectbase. For example, if

customers change their address in *SDB*, no action is required in *PDB*, the reference is always up-to-date upon use.

### 3.2.2 Unary COOL Operators: Select, Project, Pick

All of the unary COOL operators take as (initial) argument a single class name, and return as result a (set of) object(s) from that input class. So essentially, they all operate *within one LOB*. All functions that can be used in selection predicates or in projection lists have to be defined on the member type of the input, so they have to stem from the same component schema (LOB).

Any attempt to use functions from another $LOB_j$ in an operation on $LOB_i$ would be rejected at compile-time by the type checker because of type incompatibilities. Remember that the type hierarchies have been "connected" only on the very top, and two objects from different objectbases are different because of our equality definition so far.

**Example 4** The type incompatibilities between the component schemas can be seen in the following selection predicate, which would be rejected (at compile time) by the type checker:

$\mathbf{select}\ [\ i \in \mathbf{select}[\ldots](Parts)\ ]\ (\ i : Items\ )$

Since the type of the inner selection is "set of parts" and the type of $i$ is item, and because item and part are not related to each other, this selection predicate will not pass type checking.

However, we can retrieve data values from another component objectbase and use it inside a selection predicate, as shown in the next query:

$\mathbf{select}\ [\ dno(i) = cno(\mathbf{pick}(\mathbf{select}[\ldots](Parts)))\ ]\ (\ i : Items\ )$

This query passes the type checker, for basic types, namely integers in this example, have been unified by schema composition. Therefore, we can compare the numbers of parts and the numbers of items.

Later we will provide the necessary means to compare not only data values, but also objects from different LOBs (Section 4).

### 3.2.3 Assignments

For the same reasons as above, no assignments "across" LOBs pass the type checker yet, since variables defined in a local type system of $LOB_i$ are incompatible with any types from another type system $LOB_j$, if $i \neq j$.

### 3.2.4 Binary COOL Operators: Union, Intersection, Difference

The binary operators of COOL are the set operators. The definition of these is basically the mathematical definition, so they all rely on the equality (i.e., identity) predicate. Therefore, set operations applied to two arguments from different LOBs can be permitted, but since no two objects from distinct LOBs are identical, the result is always obtained in a trivial way: for the union, it is the disjoint union, for difference the result is always the first argument, and for intersection the result

is always empty. On arguments from one LOB, of course, the operations work as usual (cf. Definition 1).

This defines the effect of the set operations on the collection aspect, that is, the extent of the result. For the intentional aspect, the result type, we know [Scholl and Schek, 1990] that difference results in the type of the first argument, union in a common supertype, and intersection in a common subtype. Because the component type systems of the LOBs have not been integrated yet, there are no common functions among the argument types, so the result type of a union is "*object@GOB*" and the result type of an intersection is "*bottom@GOB*". Section 4 presents techniques that allow for more useful applications of unions and intersections spanning LOBs.

#### 3.2.5 Inter-Objectbase Views

We have seen above that the extend operation of COOL (or any similar mechanism in other object database language) can be used to define "relationships" between objects from different local objectbases, once their schemas have been composed. While schema composition is only a very rudimentary first step towards the integration of multiple OBs, the **extend** mechanism can be used to more closely couple the objects from the different sources. By using this mechanism in view definitions [Scholl *et al.*, 1991; Scholl and Schek, 1990], we can enrich the global schema.

**Example 5** If the interconnection of parts and the names of their potential local customers is not only relevant for one application program but is used in a few of them, we would want to make the new derived function a part of the global schema by the following extended schema definition:

**define schema** *CompositeDB2* **as**
    **import** *SDB;*
    **import** *PDB;*
    **define view** *Parts'* **as extend** [ *plcn* := ... ] ( *Parts* );
**end**;

As the view definition is included in the schema definition, any user accessing schema *CompositeDB2* can use the class *Parts'* and the *plcn* function defined there.

## 4 UNIFYING VIEWS OVER OBJECTBASES

In the previous section, we have shown how the extend operator of COOL can be used to establish links between objectbases. No particular semantics were given to such links, they have been defined by arbitrary queries. Now we turn to the problem of identifying objects from multiple OBs.

The first subsection (4.1) addresses the problem in general and demonstrates a solution based upon inter-objectbase links. Subsection 4.2 applies the technique to schema objects, that is, objects representing types, functions, or classes, for example. In doing so, we show how (real) schema integration can be supported. The last of the subsections (4.3) combines the results and shows the full power of objectbase integration obtained by our approach.

### 4.1 UNIFYING OBJECTS

Since LOBs have been used independently of each other, one real world ("entity") object may be represented by different ("proxy") objects in different LOBs [Kent, 1991]. The fundamental assumption of object-oriented models, namely that one real world object is represented by exactly one database object is no longer true. Consequently, the fundamental concept of "object identity" found in objectbases needs revision. Within one OB, identity is based on object identifiers (OIDs).[2] Two objects are identical if they have been created by the same object creation event (that is, if they have the same internal OID).

Now that we consider several local objectbases, we need an additional notion: we say that two (local, or proxy) objects are *the same*, iff they represent the same real world (or entity) object. Notice that two proxy objects from two different LOBs are in general *not identical*, even if they are *the same*.[3] Following Kent, an *entity object* occurs in the real world, a *proxy object* is an LOB-specific representation of (some aspects of) an entity object. *Entity (or global) identity* unifies proxy objects, iff they represent the *same* entity object. *Proxy (or local) object identity* is defined only w.r.t. one LOB; in this LOB, it is the usual notion of object identity found in (single) objectbases, namely the one based on internal OIDs.

In the following, we deal with the "same" or global object identity problem in two steps: first, we define the *unification* of proxies, i.e., we discuss how to find out what the "same" object in another LOB is; second, we devise a way to re-establish a notion of global (entity) object identity.

#### 4.1.1 Finding the Same Object in other LOBs

Within one objectbase, the referencing mechanism is typically based on the internal OIDs. These have been introduced to uniquely refer to objects, especially in the case of sharing. From the outside of an objectbase, however, OIDs are not available (or at least, should not be available). Other means for the specification of the "object(s) of interest" are therefore needed. Every objectbase system provides some form of query language (that might just be C++-Code or an SQL-style language), which can be used to retrieve objects. Obviously, the query language of an LOB is a good (if not the only) choice for a referencing mechanism across LOBs.

Therefore, our approach to the first step, the *unification* of different proxy objects from different local OBs, is to use the inter-objectbase linking mechanism of COOL's extend operator. Given a proxy object $o_i$ in some $LOB_i$, the information we are after is: what is the proxy object $o_j$, if any, in some other $LOB_j$ that represents the same entity object in the real world? An extend operation can establish a link to some object(s) in that other $LOB_j$, all we need is a query expression that determines what the "*same*" object in that other objectbase is. We have to use the

[2] There is also a notion of equality between objects, this is based on properties associated with the objects. One can even distinguish several kinds of equality, such as shallow, deep, $n$-deep [Shaw and Zdonik, 1989]. As a result, two objects can be equal but still not identical.

[3] Even if two proxy objects from two LOBs happen to have the same internal OID, this does not mean anything!

standard mechanisms offered by $LOB_j$ to select objects of interest: an appropriate selection predicate. Notice that, due to the complex structure of the schema of $LOB_j$, we may use simple properties (atomic values associated to $LOB_j$-objects) as well as complex context conditions (such as properties of other $LOB_j$-objects related to them via object-valued functions).

For example, if two LOBs contain classes of persons, we might assume that two person proxies from the two LOBs represent the same real person if they agree on the name and birthdate properties (atomic valued functions defined of the corresponding types in both LOB schemas). Alternatively, as an example of a more complex condition using context information, we could assume that names and birthdates as well as those of spouses and children (if any) have to agree. In the latter case, we made the assumption that both LOBs carry the necessary information and that we have already found out that this is the case (see also 4.2).

### 4.1.2 Global (Entity) Identity

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 given a query expression that determines, for a given $T_i$-object $o_i$, 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.

Given the definition of some of these same-functions, we can now extend the definition of global object identity (cf. Def. 1 and Figure 3) to account for these user-defined same-functions (see also "merge conditions" in [Dayal, 1983]).

**Definition 2 (Global identity – revised)** *The global (entity) object identity predicate '$=_{gl}$' on the global type* object@GOB *is defined as:*

$$
\begin{aligned}
=_{gl} \quad &: \quad \text{object@GOB} \times \text{object@GOB} \rightarrow \textbf{boolean} \\
=_{gl}\ (o_1, o_2) \quad &= \quad (\exists i : object_i(o_1) \wedge object_i(o_2) \wedge o_1 =_i o_2) \\
&\quad\ \vee\ (\exists i, j : \exists T_i, S_j : \exists same_{T_i,S_j} : \\
&\qquad\qquad object_i(o_1) \wedge object_j(o_2) \wedge o_2 =_j same_{T_i,S_j}(o_1))
\end{aligned}
$$

If we use this notion of object identity throughout the object algebra, that is, in selection predicates and in all set operations (as the criterion for duplicate detection, for example), then we have a global notion of object identity which is a generalization of (proxy) object identity as supported by each local objectbase: two objects are the same, if they stem from the same LOB and are identical in it, or if they have been defined (by the user) to be the same in a same-predicate.

### 4.1.3 Examples

In our example multi-objectbase application, there are plenty of opportunities to define a global notion of identity.

**Example 6** All LOBs, *DDB*, *PDB*, and *SDB* contain some information about the parts that are being designed, produced, and sold by the company. Let us

assume that an analysis of the local schemas has found out that *Items@DDB*, *Parts@PDB*, and *Articles@SDB* all represent the same real world entity objects. Notice that the correspondence need not be total in the sense that all items of the DDB are necessarily present as articles in the SDB as well, but some of them surely are, and vice versa.

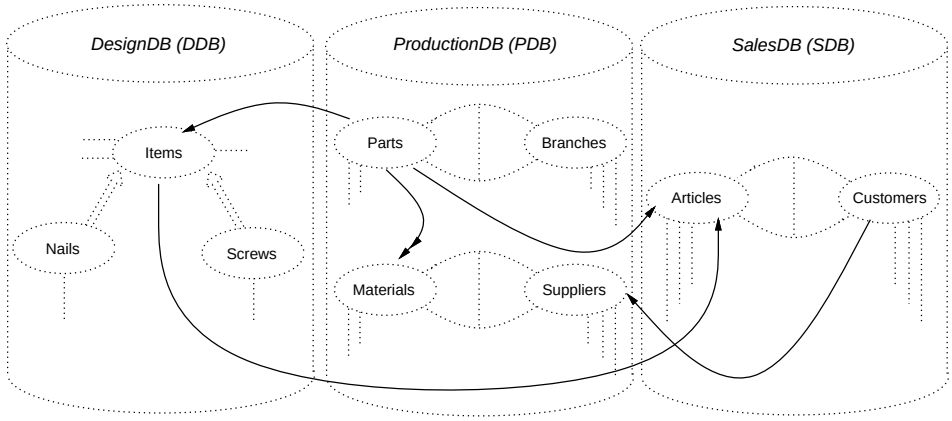


Figure 4: Multi-objectbase scenario extended with derived functions

For simplicity reasons, let us further assume that *dno, cno, ano* are actually all represented in the same way and that they agree for the same real world objects. In this simple case, we can define some of the same-functions as follows:

**extend** $[same_{part,item} := \mathbf{pick}(\mathbf{select}[cno(p) =_D dno(i)](i : Items))]$
$(p : Parts)$;

**extend** $[same_{part,article} := \mathbf{pick}(\mathbf{select}[cno(p) =_S ano(a)](a : Articles))]$
$(p : Parts)$;

**extend** $[same_{item,article} := \mathbf{pick}(\mathbf{select}[dno(i) =_S ano(a)](a : Articles))]$
$(i : Items)$;

The other three same-functions are defined analogously. Notice, that same-functions define an equivalence relationship and are therefore transitive. In addition to each same-function, the inverse is defined automatically. See Figure 4 for an illustration. For the sake of readability, we left out the full specification of global names (e.g., *Parts* instead of *Parts@PDB*) since local names are globally unique. Further note that "$=_D$" stands for local (proxy) object identity within *DesignDB*, and so on.

A more complex condition for global identification may involve path expressions in either of the LOBs. Example 8 shows such a case. In order to prepare for it, and to show another interesting case, namely one where we define an LOB-internal link based upon inter-OB links, we give a further example before that.

**Example 7** The following statement defines a derived function within *PDB* that uses information from the *DDB*: For each part, we want to know the raw material used for its production. This information is not explicitly represented in *PDB*, but it can be obtained from the *DDB*, where materials are recorded for all items (see Figure 4).

$$\textbf{extend}\ [mat := \textbf{select}[mno(m) = matn(same_{part,item}(p))](m : Materials)]\ (p : Parts)$$

Using this newly derived function, we can proceed to the complex case, where objects from the LOBs are unified using their context.

**Example 8** Obviously, two of our local objectbases contain information about other companies: *Suppliers@PDB* and *Customers@SDB*. Typically, some of our supplier companies may well purchase some of our products. Therefore, we may have some pairs of proxies in the two classes that actually represent the same entity.

Let us assume the following: if a supplier $s$ and a customer $c$ agree on their names (*sname* and *cname*, respectively), and if the customer has not bought any of the articles produced from the raw materials supplied by the supplier, we conclude that they are the same. We do not make use of the address information contained in either of the LOBs, since order and delivery addresses of the other companies usually disagree. Notice that the identification predicate actually involves our third database, namely by reference to the *mat* function defined above.

We define the same-function from *Customers@SDB* to *Suppliers@PDB* as

$$\begin{aligned}&\textbf{extend}\ [same_{customer,supplier} := \textbf{pick}(\\ &\quad \textbf{select}\ [\textbf{select}\ [sname(supplby(mat(same_{article,part}(a)))) = cname(c)]\\ &\qquad\qquad (a : bought(c)) = \emptyset \wedge cname(c) = sname(s)]\\ &\qquad (s : Suppliers))]\\ &\quad (c : Customers)\end{aligned}$$

Notice that, even though we use object-valued functions in the path expression, the final comparisons are again between data values, so we do not need to subscribe the equality signs '=' in the selection predicate (see Figure 4 again).

The last example showed that criteria for unifying (proxy) objects from different LOBs can be based on quite involved query expressions. Conceptually, however, there is no other way to refer to foreign objects than by a query.

### 4.2 SCHEMA INTEGRATION

We have presented a mechanism to unify (proxy) objects in different LOBs. We now concentrate on schema integration, that is to find out, what the common parts in the local schemas are, and to define a correspondence among them [Civelek *et al.*, 1989; Spaccapietra *et al.*, 1992].

We can consider Example 1, where *Items, Parts*, and *Articles* have a number function defined on it. Although they are named different, their semantics may be

the same. Moreover, three functions with name *weight* are defined in the multi-objectbase. The weight of *Items* has the same semantics as the weight of *Articles* (synonyms). Nevertheless, one is given in kilo and one in pound. Yet another weight is the one of *Materials*. Though having the same name, its semantics are different: here it means the specific weight of raw materials (homonym).

### 4.2.1 The global Meta Schema

In Section 3, when introducing schema composition, we said that the effect is to create a new "*object@GOB*" and "*bottom@GOB*" type, as well as a new "*objects@GOB*" class. Also, we assumed that the basic data types (integer, etc.) are unified. Actually, more than this happens when two (or more) local schemas are composed. As all LOB schemas contain a meta schema that is structured in the same way [Tresch and Scholl, 1992], also these meta types and meta classes are combined.

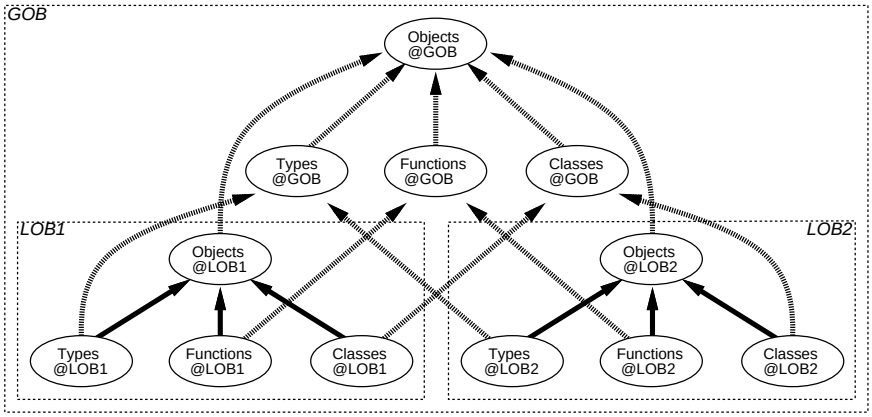


Figure 5: The Global Meta Schema

To be precise, each LOB contains three meta types (*type, function, class*), and three meta classes (*Types, Functions, Classes*), respectively. Schema composition creates a global meta schema, with new types "*type@GOB*", "*function@GOB*", and "*class@GOB*", as well as the new classes "*Types@GOB*", "*Functions@GOB*", and "*Classes@GOB*". Thus, the meta schema of the GOB has the same structure as that of each LOB. In addition to this, schema composition makes the local meta types and classes subtypes/subclasses of their global counterparts (cf. Figure 5). Thus, the global meta objects are the union of the local meta objects, representing local meta types and classes ($\ldots@LOB_i$), plus the meta objects representing the newly created global meta types and classes ($\ldots@GOB$).

Furthermore, meta functions are unified during the composition process, e.g. in $LOB_1$, there is a meta function *super_types(t)* finding all supertypes of type $t$. This function is automatically unified with the *super_types* function of $LOB_2$, and so on.

### 4.2.2 Unifying Meta Objects

The next step in schema integration (after schema composition that combines the meta schemas) is to define correspondences among functions of different LOBs. We make use of the fact that every function is represented by a meta object. In Figure 5, these function objects are kept in the meta classes $Functions@LOB_i$.

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 the meta type $function@LOB_i$ to the meta type $function@LOB_j$.

**Example 9** To unify the functions *dno* with *ano* and *weight* with *weight* between *DDB* and *SDB* the following *same*-function is defined on the meta schema of *DDB*:

**extend** [ $same_{\ function@DDB, function@SDB} :=$ **pick**(
**select** [ $(name(f) =$ "dno" $\wedge\ name(g) =$ "ano") $\vee$
$(name(f) =$ "weight" $\wedge\ name(g) =$ "weight")
] $(g : Functions@SDB)$) ]
( $f : Functions@DDB$ )

For each further correspondence of functions, the selection predicate must be extended with another name-equality term.

This is the way to integrate homogeneous functions from different databases. If there are structural or semantic inhomogeneities, these must be resolved before. Examples include different types for currency values or units of measure in general (here we need conversion functions), functions that have different names in different LOBs despite carrying the same semantics (here we need renaming), functions with the same names but different semantics (again, we need renaming), and so forth.

Renaming can easily be realized in COOL with the extend operator: we define a new derived function (with the new name) that takes exactly the value of the old function. Also, conversion functions can be expressed using the extend operator, since we can apply arbitrary computations in the definition of derived functions.

**Example 10** Consider the inhomogeneities between the different *weight* functions. They are resolved with the following redefinitions:

**extend** [ $weight\_kilo := weight(i), weight\_pound := weight(i) * 2.2046$ ]
( $i : Items$ );

**extend** [ $weight\_kilo := weight(a) * 0.453, weight\_pound := weight(a)$ ]
( $a : Articles$ );

**extend** [ $spec\_weight := weight(m)$ ]
( $m : Materials$ );

Notice that, in general, derived functions can not be updated. In many cases, however, it is quite obvious how to propagate such updates. In the case of conversion functions as shown above, we can expect that the system automaticlly determines *inverse* functions (e.g. *pound_to_kilo* vs. *kilo_to_pound*). Consequently, these functions can be updated. More generally, we allow the

definition of update methods "*set_f*" for each function $f$ that will be used when assigning values to $f$ in generic update statement.

### 4.3 INTEGRATION OF OBJECTBASES

We now have defined all prerequisites for tight objectbase integration: On the object level, we showed how to find the "same" object in other LOBs and introduced a notion of global identity based on the *same*-functions. On the schema level, we discussed integration of functions as a unification of meta objects.

**Example 11** We stated in Example 6 that all LOBs, *DDB, PDB, SDB* contain information about the same real world entities, namely nails and screws. We can now integrate the three objectbases, for example by defining an intersection view *Elements* over multiple object-bases: On the object level, we unify *Items, Parts,* and *Articles* if they have equal numbers ($dno = cno = ano$). On the schema level, we integrate the number functions (renaming it to *eno*) and the weight functions (resolving semantic inhomogeneities).

**define schema** *ElementsDB* **as**
**import** *DDB, PDB, SDB;*
[1] **define view** *Items'@DDB* **as**
[1a] **extend** [ $same_{item,part}(i) := \ldots$,
[1b] $same_{item,article}(i) := \ldots$,
[1c] *eno:= dno(i),*
[1d] *weight_kilo:= weight(i),*
[1e] *weight_pound:= weight(i) * 2.2046* ]
( *i:Items@DDB* );
[2] **define view** *Parts'@PDB* **as**
[2a] **extend** [ $same_{part,item}(p) := \ldots$,
[2b] $same_{part,article}(p) := \ldots$,
[2c] *eno:= cno(p),*
[2d] *mat:=* **select** [ ... ]
( *m:Materials@PDB* ) ] ( *p:Parts@PDB* );
[3] **define view** *Materials'@PDB* **as**
[3a] **extend** [ *spec_weight:= weight(m)* ]
( *m:Materials@PDB* );
[4] **define view** *Articles'@SDB* **as**
[4a] **extend** [ $same_{article,item}(a) := \ldots$,
[4b] $same_{article,part}(a) := \ldots$,
[4c] *eno:= ano(a),*
[4d] *weight_kilo:= weight(a) * 0.453,*
[4e] *weight_pound:= weight(a)* ]
( *a:Articles@SDB* );
[5] **define view** *Functions'@DDB* **as**
**extend** [ $same_{function@DDB,function@PDB}(f) := \ldots$,
$same_{function@DDB,function@SDB}(f) := \ldots$ ]
( *f:Functions@DDB* );
[6] **define view** *Functions'@PDB* **as**
**extend** [ $same_{function@PDB,function@SDB}(f) := \ldots$,
$same_{function@PDB,function@DDB}(f) := \ldots$ ]

( *f:Functions@PDB* );

[7] **define view** *Functions'@SDB* **as**
**extend** [ $same_{function@SDB,function@DDB}(f)$:= ...,
$same_{function@SDB,function@PDB}(f)$:= ... ]
( *f:Functions@SDB* );

[8] **define view** *Elements* **as** *Items'@DDB* **intersect** *Parts'@PDB*
**intersect** *Articles'@SDB* ;

**end** ;

The method of database integration by finding corresponding properties and proxy objects was investigated by Neuhold and Schrefl [Schrefl, 1988; Neuhold and Schrefl, 1988] as well. They introduce for each kind of semantic relationship between objects a new special type of generalization class: data-type-generalization, role-generalization, history-generalization, counterpart-generalization, category-generalization. Object coloring is used to decide from which local database the objects in the generalization class originate and message forwarding is presented to overcome semantically inhomogeneous representations.

Our approach makes exclusive use of an algebraic object views, using the **extend** operator to establish links between objectbases, in order to find the same object in other LOBs and to resolve semantics inhomogeneities. All LOBs are strongly typed systems, such that static type checking of objects is possible.

Consider Example 11: extensions ($1ab, 2ab, 4ab$) define the *same* functions to unify proxy objects. This is similar the Schrefl's "object-correspondence-rules" in the generalization classes. The derived functions ($1c, 2c, 4c$) rename the different number functions to *eno*, and ($1de, 4de$) resolve semantic inhomogeneities between the *weight* functions, such that "pounds" are translated to "kilos" and vice versa. To avoid confusion, "specific weight" gets a new name ($3a$). To perform this, Schrefl introduces "data-type-generalization" classes with the possibility the define "transformation methods". Once all inhomogeneities are resolved, we are ready to define the *same* functions ($5, 6, 7$) on the meta level to unify the functions *eno, weight_kilo, weight_pound* from all LOBS. Schrefl's generalizations achieve the same by introducing "corresponding relationships/attributes". Our approach needs no special generalization mechanism to define a unifying view of these databases, i.e. to create an intersection view (8) over several databases.

# 5 PROPAGATION OF UPDATES

A critical issue in multi-database systems is the propagation of updates , particularly when admitting local transactions. In fact, the view mechanism that we propose offers a big advantage in that respect: since all COOL views are updatable, we will see that a basic update propagation across multiple LOBs comes for free.

## 5.1 INTER-OBJECTBASE LINKS

If we establish links between different LOBs via extend views, that is, by means of query expressions against the foreign (referenced) objectbase, all updates that happen within the foreign objectbase are automatically reflected in the referencing

LOB. No special care has to taken. Whether the referenced objects change some of their properties (including "relationships" to other objects) or whether they are removed or new ones are created within the foreign OB, all these changes are visible from within referencing LOBs, since they all use query expressions that are evaluated upon each reference.

Symmetrically, we can even update foreign objects (in the referenced $\text{LOB}_i$) from within transactions on $\text{LOB}_j$: like with all **extend**-views, the objects obtained from the application of the derived function can be updated (using the update statements available in $\text{LOB}_i$ in our case).

**Example 12** In our running example we could, for instance, have a function *lc*, returning for each supplier of *PDB* its local customers of *SDB*. Assuming that *s*0 is one particular supplier, we can modify the names of its local customers as follows.

**update** [ cname:="New Yorker" ] ( lc(s0) )

### 5.2 UNIFIED OBJECTS

If we have unified (proxy) objects in different LOBs, we would expect that a global delete operation will actually delete all proxies in all LOBs. In fact, since updates on union views are propagated to all base classes [Scholl *et al.*, 1991], this is exactly what will happen. Conversely, if individual proxy objects are deleted by autonomous local transactions, only the local proxy is removed, but other proxies representing the same entity object may still be present in other LOBs. Again, this is what we would expect if we permit autonomous updates to LOBs that are not executed under the control of a global integrity monitor.

Technically, this effect is automatically achieved by the fact that the **delete** operation is actually derived from COOL's **lose** operation that dynamically changes the type of an object: in each $LOB_i$, **delete**(o) is defined as **lose**[$object_i$](o) [Laasch and Scholl, 1992; Scholl *et al.*, 1992]. Therefore, since in the global context $object_i$ is not the root of the type lattice, only the local proxy, but not all the other proxies (if any) are removed.

**Example 13** Consider the deletion of a part object in the first case locally on the production objectbase *PDB*, and in the second case globally on the integrated objectbase. The global schema includes the same functions between parts, items, and articles, and the global superclass "Elements". The effect of the deletion depends on the target objectbase to which it is applied:

- A *local* execution against *PDB* removes the part (proxy object) with the given construction number from the production objectbase.

  **delete** ( **select** [cno=123456789] (Parts) )

  However, no propagation into the other two objectbases is implied, since local execution does not even know about the existence of these.
- A *global* execution on the other hand means that *all proxies* in all participating LOBs are removed, since a global delete 'knows' about potential proxies in the other LOBs that represent the same entity.

**delete** ( **select** [eno=123456789] (Elements) )

In case we had no globally integrated schema that unified the three classes, we would have had to issue three separate delete operations, one per LOB.

## 6 CONCLUSION

We presented an approach towards the stepwise integration of (populated) objectbases into a multi-objectbase system. The basic mechanisms that we described allow for the following steps:

(i) Schema composition makes the names of schema elements (such as types, classes, and functions) known across local objectbases. As a result, transaction against a composite schema can issue queries and updates on classes from different LOBs. We can define *inter-objectbase links* by defining new, derived functions across LOBs using the **extend** operator. Without any further means, however, only basic values (such as strings or numbers) can be 'transferred' between LOBs.

(ii) "Same" functions can be used to unify proxy objects from different LOBs that represent the same real world entity object. This accounts for the fact that different LOBs have created their own representations independently.

(iii) A global notion of object identity can be defined based upon these "same" functions. It is important to notice that this global notion of object identity is essentially based on values.

(iv) The mechanism of same functions can be used on the meta level in order to unify schema elements from different LOBs. This provides a sound formal basis for what is generally known as "schema integration". Once correspondences between local schemas have been detected, we provide a way of unifying them.

The advantage of our approach is that it uses almost exclusively the expressive power of a query language, together with a view definition facility. The only extension that we added is the global object identity predicate. Given that an objectbase system allows for the redefinition of all generic functions, this does not even require to change anything in the implementation of local systems.

Because our views are updatable, this approach is also a starting point for multi-objectbase updates. This distinguishes our approach from [Landers and Rosenberg, 1982], for example. However, to provide full updatability, we need further work, e.g., on the derivation of inverse functions for extend views.

While from a conceptual point of view our approach looks simple and clean, we have to spend more effort on the implementation issues. Particularly an efficient, but overridable, implementation of the identity test seems to require further investigations. Till now, comparing OIDs is typically hardcoded into the implementation of local OBMSs. As we have seen, the (global) identity test now depends on user-defined *same*-functions. Therefore, we need efficient support by indexes or some other form of replicated information.

## References


[Abiteboul and Bonner, 1991] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conf. on Management of Data*, Denver, Colorado, May 1991. ACM Press.

[Ahmed *et al.*, 1991] R. Ahmed, P. De Smedt, W. Du, W. Kent, M.A. Ketabchi, W.A. Litwin, A. Rafii, and M.-C. Shan. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12), December 1991.

[Bancilhon *et al.*, 1989] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the $O_2$ object-oriented database system. In *2nd Int'l Workshop on Database Programming Languages*, Oregon Coast, June 1989. Morgan Kaufmann.

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

[Borgida *et al.*, 1989] A. Borgida, R.J. Brachman, D.L. McGuinness, and L.A. Resnick. CLASSIC: A structural data model for objects. In *Proc. ACM SIGMOD Conf. on Management of Data*, Portland, OR, June 1989.

[Brachman and Schmolze, 1985] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9, 1985.

[Civelek *et al.*, 1989] F.N. Civelek, A. Dogac, and S. Spaccapietra. An expert system appraoch to view definition and integration. In *7'th Int. Conf. on Entity-Relationship Approach: A Bridge to the User*, Rome, Italy, November 1989.

[Cluet *et al.*, 1989] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. RELOOP, an algebra based query language for an object-oriented database system. In *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, Kyoto, December 1989. North-Holland.

[Dayal, 1983] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proc. 9th Int'l Conf. on Very Large Data Bases (VLDB)*. Morgan Kaufmann, September 1983.

[Geller *et al.*, 1991] J. Geller, Y. Perl, and E.J. Neuhold. Structural schema integration in heterogeneous multi-database systems using the dual model. In *Proc. 1st Int'l Workshop on Interoperability in Multidatabase Systems (IMS)*, Kyoto, Japan, April 1991. IEEE Comp. Soc. Press.

[Kaul *et al.*, 1990] M. Kaul, K. Drosten, and E.J. Neuhold. Viewsystem: Integrating heterogeneous information bases by object-oriented views. In *Proc. 6th Int'l IEEE Conf. on Data Engineering (ICDE)*, Los Angeles, CA, February 1990. IEEE Comp. Soc. Press.

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

[Kim, 1989] W. Kim. A model of queries for object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, Amsterdam, August 1989.

[Krishnamurthy *et al.*, 1991] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, Colorado, June 1991.

[Laasch and Scholl, 1992] 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 (IS/KI)*, Ulm, February 1992. Springer IFB.

[Landers and Rosenberg, 1982] T. Landers and R.L. Rosenberg. An overview of Multibase. In *Proc. 2nd Int'l Symp. on Distributed Data Bases*, Berlin, Germany, September 1982. North-Holland.

[Motro, 1987] A. Motro. Superviews: virtual integration of multiple databases. *IEEE Trans. on Software Engineering*, 13(7), July 1987.

[Neuhold and Schrefl, 1988] E.J. Neuhold and M. Schrefl. Dynamic derivation of personalized views. In *Proc. 14th Int'l Conf. on Very Large Data Bases (VLDB)*, Los Angeles, California, September 1988. Morgan Kaufmann.

[Saltor *et al.*, 1991] F. Saltor, M. Castellanos, and M. Garcia-Solaco. On canonical models for federated DBs. *ACM SIGMOD Record*, 20(4), December 1991.

[Scholl and Schek, 1990] M.H. Scholl and H.-J. Schek. A relational object model. In *ICDT '90 – Proc. Int'l. Conf. on Database Theory*, Paris, December 1990. LNCS 470, Springer.

[Scholl *et al.*, 1991] 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, December 1991. LNCS 566, Springer.

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

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

[Shaw and Zdonik, 1989] G.M. Shaw and S.B. Zdonik. An object-oriented query algebra. *IEEE Data Engineering Bulletin*, 12(3), September 1989. Special Issue on Database Programming Languages.

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

[Spaccapietra *et al.*, 1992] 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.

[Straube and Özsu, 1990] D.D. Straube and M.T. Özsu. A model for queries and query processing in object-oriented databases. In *Proc. ACM SIGMOD Conf. on Management of Data*, December 1990.

[Tresch and Scholl, 1992] 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.