Published in Proc. 3rd GI Workshop Information Systems and Artificial Intelligence, Hamburg, March 1994.
Springer LNCS 777, pp. 64–82.

Evolution towards, in, and beyond Object Databases

Marc H. Scholl and Markus Tresch

Faculty of Computer Science
Databases and Information Systems
University of Ulm, D-89069 Ulm, Germany
{scholl,tresch}@informatik.uni-ulm.de

Abstract. There is a manifold of meanings we could associate with the term "evolution" in the database arena. This paper tries to categorize some of these into a unique framework, showing similarities and differences. Among the topics touched upon are: extending traditional data models to become "object-oriented", migrating existing data to (not necessarily OO) databases, schema extension and modification in a populated database, integration of federated systems, and the use of "external services" to enrich DBMS functionalities. The following are presented in more detail: first, we describe the necessity of object evolution over time; second, we discuss schema evolution; and third, we present evolutionary database interoperability by identifying different coupling levels. A few basic mechanisms, such as views (derived information) and a uniform treatment of data and meta data, and type and/or class hierarchies, allow for a formal description of (most of) the relevant problems. Beyond presenting our own approach, we try to provide a platform to solicit further discussion.

1 Introduction

The term evolution is used in this paper in a very broad sense to denote several kinds of dynamics that occur in the context of database systems and database applications. We will identify three main lines of evolution that are addressed in the title:

– *Evolution towards object databases:*
  Here we address the advance of database technology in terms of data models. Data models have evolved from flat files via first generation DBMSs (network, hierarchical), second generation DBMSs (relational) to third generation DBMSs (extended relational, object-oriented, ...) [30]. It has been the primary concern of our prior work [21] to point out that particularly the latter advance can in fact be evolutionary, i.e., preserve the advantages of relational technology, such as powerful descriptive query and update languages.

- *Evolution in object databases:*
  This covers all aspects of the life-cycle of database objects. Traditional issues such as database updates (including maintenance of DB consistency), more advanced issues such as dynamic type changes of existing objects (often called "objects in different roles") or automatic classification of objects into subclasses (as known from AI systems), and finally dynamic schema modification in populated databases are addressed under this topic.
- *Evolution beyond object databases:*
  This third meaning of the term is the most challenging, both from an academic as well as from a practical point of view. Here we talk about interoperability among several data management tools (not all of which are necessarily database systems). Stepwise integration of preexisting autonomous databases into a federated system, (smooth) migration of file-based or relational data-intensive applications into ODBMSs, and the integration of external services into a DBMS are the problem areas we are aiming at here.

Clearly, the points mentioned above are relevant, important and difficult to solve. We can not provide complete solutions, rather we have tried over the last few years to address some specific technical questions within this framework. Most of the problems have two parts: policy and mechanisms. We have concentrated on the necessary mechanisms, and have not concerned ourselves with policy issues, such as, for example, *how* to resolve conflicts in schema integration, rather we provide a platform that can implement any conflict resolution strategy by means of formally defined schema transformation primitives (including instance conversion, when necessary). The main focus has been on the exploitation of query and update languages in the presence of a semantically rich (object) data model as an underlying formal mechanism.

The first line of evolution (the advances of the technology of DBMSs and of data models) is mentioned in this paper only in passing. It has turned out that the essentials of relational DB languages, together with ingredients from other fields (most prominently, AI knowledge representation languages and programming languages), can be carried over to ODB languages. The most important feature — as far as evolution *in* object databases is concerned — is to provide all kinds of derived information (e.g., computed attributes, views, subschemas, type inferencing, automatic classification). We have designed a prototype object database language, COOL, to prove the concepts and as an experimentation platform [25, 24, 23, 11]. Update operations of this COOL language respect all the integrity constraints, that is, they automatically propagate to all the derived concepts.

The second line of evolution (database dynamics) profits from those language properties described above. Once database updates are defined formally in such a way that integrity constraints and derivation rules are automatically maintained, there remain only a few modifications of data objects to be discussed. Among them are dynamic changes of the types of existing objects, a functionality that will be described below. The next level of complexity is changes to the structure, i.e., the schema of a database. Object databases typically have a meta schema that contains types/classes that describe the schema. Consequently, the database language can be used to query (data dictionary functionality, also present in most relational DBMSs) and update (schema modification functionality, typically prohibited in most DBMSs) those schema objects. We will show that the COOL language can be used to implement schema evolution by means of schema modification primitives that include propagation to the instance level (automatic database reorganization due to schema changes) [33, 34, 31].

Finally, the third line of evolution leaves the context of a single system and considers schema changes in the context of a stepwise integration of previously independent databases. It turns out that with very small extensions to the language semantics (mainly the introduction of an application-defined object identity predicate to unify objects across databases), the schema modification operations can be used to implement the integration strategies. We identify five levels of increasingly tight database interoperation that are distinguished by the language features they need to exploit. These levels can be observed in other approaches to ODBS integration, too. Hence, the results obtained here are not limited to the COOL language. An even wider interpretation of this last evolutionary line includes migration paths from file-based or RDBMS applications to ODBMSs, an issue that we will not elaborate in this paper, though. Some preliminary considerations are presented in [19]. Ultimately, database systems and other service providers in a cooperation of interoperable systems may interact in more subtle ways, for example, a DBMS may import computation services from other systems, in the sense of externally defined data types [22].

The paper gives a tour through several aspects of this framework. Particular focus lies on evolution mechanisms within a single DBMS and beyond single DBMSs for DB integration. Section 2 shows the basic evolution mechanisms provided by the COOL language. In Section 3, we discuss schema modification within a single database, and Section 4 presents the five levels of database interoperability, together with the COOL mechanisms that can be used to realize them. We conclude with a comparison with related approaches and a summary.

2 Fundamental Evolution Mechanisms

There are a couple of fundamental mechanisms in ODBSs, that are prerequisites for such systems to support dynamic evolution. We identify some of them, using the object model COCOON [23] as a platform.

2.1 Object Evolution

Object evolution describes the problem that a real world entity may be represented by a database object in several roles over time. Thus, we need the possibility in ODBSs to change dynamically type instantiation and class membership of objects. Furthermore, objects may be instances of multiple types and classes at a time.

Example 1. Consider an object $john$, being instance of object type $student$. It may be necessary that this object loses the properties (attributes) of $student$ and should be removed from class $UniUlmStuds$.

$\textbf{lose}[student](john)$; $\textbf{remove}[john](UniUlmStuds)$;
$\textbf{gain}[employee](john)$; $\textbf{add}[john](CompanyX)$

Later on, $john$ may gain properties of $employee$ instead, and should be added to class $CompanyX$.

Unfortunately, this kind of object evolution is a neglected concept in object database systems, and is supported by only a few data models, e.g. Iris [7]. Especially C++ based databases either ignore such mechanisms, or need costly off-line reorganizations to do so, e.g. ObjectStore [16], ONTOS [17].

One main reason is an implementation reason, namely that such systems do not manage OIDs with location transparency, but either consider an object's virtual memory address as its OID, e.g. ObjectStore, or encode class identifiers into the OID, e.g. ORION [10].

2.2 Logical Data Independence

Logical data independence is commonly established by defining external schemata, i.e., schemata that are derived (computed) from the logical level. Although there is no widely accepted concept of views in object databases [32], they are often understood as virtual (derived) classes, declared for example as

define view v **as** e

where e is a query expression. Our language supports logical data independence, i.e. it offers possibilities to define derived class extents, derived object types, and derived object properties (refer to [24] for a comprehensive discussion of the view mechanism).

We now argue that data independence in object database systems is a first attempt to realize database evolution. In fact, views can be used straight forward to simulate some schema evolutions.

Example 2. Consider an ODBS schema with two classes $Boys$ and $Girls$, and a schema evolution, integrating these two classes into one unified class $Persons$. The following view definition is a first approach

define view $Persons$
as $\textbf{extend}[sex{:=}\,(o \in Boys)](Boys\ \textbf{union}\ Girls)$

The extent of view $Persons$ holds the union of all objects from $Boys$ and $Girls$. The object type of the view consists of the intersection of the functions from the classes $Boys$ and $Girls$, plus an additional boolean function sex, distinguishing male ($sex = true$) and female ($sex = false$) persons.

The above example does not yet fully simulate the desired schema evolution. Although many updates of and into views are already managed by the system, update propagation rules must be added for non-standard transformation of view updates into base class updates. E.g., because *sex* is a derived function, any attempt to assign a new value to it, would be rejected by the parser already.

Example 3. We complete the previous example by adding update transformation rules. It shows, how assigning a new value to *sex* and adding/removing person objects is propagated into updates of base classes:

```
define view Persons
as   extend[sex:= (o ∈ Boys)](Boys union Girls)
on   set[sex:= x](o)   do if x then add[o](Boys); remove[o](Girls)
                                else  add[o](Girls); remove[o](Boys)
on   add[o](Persons)   do if sex(o) then add[o](Boys)
                                      else  add[o](Girls)
on   remove[o](Persons)   do remove[o](Boys); remove[o](Girls)
        ⋮
```

Finally, this is now a fully equivalent simulation of the desired schema evolution.

After view classes have been defined and positioned in the schema, a subset of existing classes and views can be collected in a subschema. Subschemata describe a part of the base schema (extended with views) that should be made available to external users. Typically, we have to require that subschemata are closed [20, 34].

Since external schema cannot be used to simulate all desired schema evolutions, some must be physically performed, because they augment the information content of the database. In the sequel, we search for a formal handle of that difference.

2.3 Uniform Treatment of Data and Meta Data Objects

In ODBSs, data and meta data are often treated uniformly as objects. That is, an object database can be partitioned into three disjoint sets of objects: (i) primary objects, representing user data, (ii) secondary objects, representing the user's application schema, and (iii) tertiary objects, representing the data model itself. Since all of these are "ordinary" objects, generic query and updates operations should be applicable as usual on any kind of object.

Whereas querying the meta database is quite common (cf. data dictionary), updating meta databases is usually only allowed through a special data definition language. However, meta databases should be updatable for schema evolution purposes.

Example 4. Consider the following generic update sequence, creating a new variable *a*, adding it to class *Variables*, and assigning values to its attributes, e.g. the variable name:

create$[variable](a)$; **add**$[a](Variables)$; **set**$[name := n; \ldots](a)$

Direct updates to meta databases may cause several side effects that must be handled: (i) to ensure that no incorrect database schemata are produced, a set of integrity constraints must be modeled in the meta database; (ii) to ensure that no inconsistency between the schema and the instance level arises, schema updates must be propagated to existing objects; (iii) to ensure that no run-time errors occur in existing (compiled) applications, these programs must be recompiled.

3 Schema Evolution

In this section, we follow our second line and consider evolution issues within one independent object database. First, we show, how external schemata can be used as a first attempt for schema evolution and why this is not yet satisfactory. Then, we develop a formal evolution model, which is independent from any concrete object model. Based on that, a set of elementary schema updates is presented and global database restructuring is discussed.

3.1 A Formal Evolution Model

In order to be able to characterize schema evolution (i.e. to determine, whether it can be simulated), we define a formal description of database evolution, which is independent of any concrete data model. The notion of *information capacity* [8, 1] in object databases forms the basis for our evolution model:

Definition 1. Let S be a given object database schema. The information capacity $\mathcal{DB}_S$ is the set of all potential states, a database can have with that schema.

The definition of schema evolution follows directly:

Definition 2. Let S, S' be schemata of two object databases with database states $\sigma \in \mathcal{DB}_S$ and $\sigma' \in \mathcal{DB}_{S'}$ respectively. A schema evolution as a pair

$$< u : S \mapsto S', r : \sigma \mapsto \sigma' > ,$$

with schema update u and database reorganization r.

Schema update u is an update of the meta-database, mapping one database schema $S \in \mathcal{S}$ into another schema $S' \in \mathcal{S}$, with $\mathcal{S}$ the set of all correct schemata of a particular data model. Database reorganization r is an update of the (primary) database, mapping the actual database state $\sigma \in \mathcal{DB}_S$ fitting schema S into another database state $\sigma' \in \mathcal{DB}_{S'}$, fitting new schema S'.

In general, schema evolution changes the capacity of an ODBS, which follows from that information capacity of an object database is determined by its schema. We use this fact to classify schema evolution:

Definition 3. Schema evolution $< u : S \mapsto S', r : \sigma \mapsto \sigma' >$ is called

- capacity preserving (CP), iff mapping r is bijective;
- capacity augmenting (CA), iff mapping r is injective, but not surjective;
- capacity reducing (CR), iff mapping r is surjective, but not injective;
- capacity changing (CC), otherwise;

From the mathematical definition of database reorganization it follows: (1) CP and CR schema evolutions can be simulated, that is, there exists an external schema of S (a view definition) that is equal to S'. (2) CP and CA schema evolutions are lossless and can therefore be compensated, that is, there exists an external schema of S' that is equal to S.

3.2 A Set of Local Schema Updates

Based on this formal evolution model, a set of elementary schema updates can be defined for any concrete data model. Such a set of operations must fulfill the following formal conditions:

- **Locality:** Each schema update must be local in the sense, that one particular variable, function, type, class, or view is changed at a time.
- **Correctness:** Each schema update must be correct, such that (i) only correct database schemata are produced[1], and (ii) existing instances are correct w.r.t the new schema.
- **Capacity:** The change of information capacity by each schema update must be well defined, that is CP, CR, or CA, and never CC.
- **Minimality:** The set of schema updates must be minimal, such that no operation can be replaced by others.
- **Completeness:** The set of schema updates must be complete, such that all desired local schema changes can be performed.

The resulting set of elementary schema updates is given by the semantic concepts of the considered data model. Such sets of elementary schema updates can also be found for ORION [4], O_2 [35], and GemStone [18]. For COCOON, these are changes of variables, functions, types, classes, and views. An exhaustive enumeration can be found in [33], here, we focus on the technique, how they are implemented.

As we know, any elementary schema update is described as a tuple, consisting of a schema update and a database reorganization. We understand that u and r are performed as an atom, that is, both together or none of them are executed. They are realized as follows:

1. Schema updates ($u : S \mapsto S'$) are (sequences of) update operations, applied to the meta database, and thus changing schema information.

[1] Correctness of database schemata is usually described by a set of schema invariants, that must not be violated.

2. Database reorganizations ($r : \sigma \mapsto \sigma'$) are (sequences of) update operations, applied to the ordinary database, and thus updating instances.

Consider the following example:

Example 5. Consider an elementary schema update, specializing the type of variable **var** $p : person$ into a subtype $employee \preceq person$. Assume, every schema object is represented by an object in the meta database: i.e. there is a meta object p, representing variable "p" in the meta database, and object *employee*, representing object type "employee". To change the schema, a simple meta database update is performed, that updates value *ran* (range type) of meta object p to new value *employee*:

u : **set**$[ran := employee](p)$

Afterwards, instances of the database may be invalid w.r.t the new schema. E.g. the actual value of p may be not an instance of type *employee*. There are several possibilities to reorganize the database. One can either make p to an instance of employee (see $r1$), or alternatively check, whether p is already of type *employee*, and if not, p is set to undefined (see $r2$):

$r1$: **gain**$[employee](p)$
$r2$: **if not** $employee(p)$ **then** $p :=$ *"undefined"*

Notice that both reorganizations are not injective, and therefore not lossless. With $r1$, the information is lost, whether new p has already been instance of p before or not, and with $r1$, the value of p may be lost (set to undefined). Thus, schema updates $< u, r1 >$ and $< u, r2 >$ are both capacity reducing.

To be complete, elementary schema operations should not only exist for schema changes, but also for schema definition and deletion. To build a user interface on top of the elementary schema updates, the data definition language (DDL) must be extended towards a schema manipulation language (SML).

To do so, in addition to the DDL statement **define**, three new statements must be introduced: **redefine** to change existing schema objects, **rename** to change the name of schema objects, and **undefine** to delete schema objects. The formal semantics of the SML is now given by its mapping into elementary schema updates (e.g. implemented as a SML parser/interpreter).

Example 6. Reconsider the above example of variable p, that is defined by the following SML statement (in fact, it is a DDL statement):

define var $p : person$

The definition is mapped into the elementary schema update "define variable". Later, the type of the variable can be specialized. To to redefine p, we use the following SML statement:

redefine var $p : employee$

The SML parser compares this redefinition with the actual definition in the meta database (see section 2.3) and identifies the differences. Here, the only difference is that the new definition uses a more special instance type *employee*. Thus, this statement is mapped into elementary schema update "specialize variable type", as implemented in Example 5.

3.3 Global Database Restructuring

The SML presented so far, forms an interface to the elementary operations for local schema updates. However, not only local changes, but also global database restructurings ere desired. We show, what must be added to the SML to perform such evolution as well.

Simulated schema evolution is just a new view on top of the existing (unchanged) databases. In some cases, simulation is not yet sufficient. Capacity augmenting schema evolutions, for example, cannot be simulated, since they enlarge the information contents of an ODBS.

Thus, we need in addition the possibility to really perform schema evolution. The corresponding language mechanism follows directly from the above simulation. It mainly consist of "conceptually materializing" the schema definition as a "snapshot".

Example 7. *Persons* is now defined as a class (not anymore a view). The class is initialized using the same query as in the previous example. However, to really perform the desired schema evolution, the query is now materialized by adding all objects to class *Persons*. Again, the object type of the new class is implicitly derived.

> **redefine database** *PersDB;*
>   **define class** *Persons*
>   **from extend**[$sex := (o \in Boys)$]($Boys$ **union** $Girls$);
>   **undefine class** *Boys*, *Girls*;
> **end** .

After that, base classes *Boys*, *Girls* can be deleted from the schema. Notice that update transformation rules are not required, since after the schema evolution is performed, updates are evaluated on the new schema.

Obviously, global database restructurings may need a sequence of SML statements that must be encapsulated in schema evolution transactions. Such transactions are as usual performed completely or not at all. **redefine database** ... **end** show begin and end of transaction.

To be more flexible in defining database restructuring transactions, the meta database must be queried at run-time.

Example 8. Consider a more general variant of Example 7, where class *articles* with a variable, but at any time finite number of subclasses *screws*, *bolts*, *nails*, ... is given, and it is further assumed that every article is

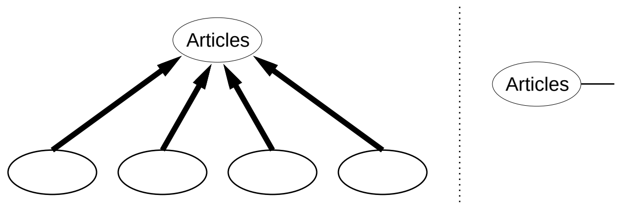

Fig. 1. Dynamic Function Definition

in exactly one subclass (cf. Figure 1). This database should be reorganized, such that only class *articles* exists, but with an additional attribute *kind* : **string**, such that every article holds as value *kind* the name of the subclass it belongs to.

define function $kind : article \rightarrow$ **string**
from $a : cname($**pick**$($**select**$[a \in extent(c)](c : subclasses(Articles))))$

This SML statement shows the desired function definition. For each article a, $kind(a)$ is set to the name of that class c, that holds a in its extent. The number of subclasses of *Articles* and its names are evaluated at run-time.

4 Evolutionary Interoperability

In this section, we follow our third line and consider evolution issues between multiple cooperating object databases.

Our point of view of evolutionary database interoperability is as follows: So far isolated database systems are starting cooperating with other systems very loosely, e.g., by global transactions. Later, global schemata (views and "others") are defined, such that systems are getting more tightly coupled, until they are finally completely integrated.

To classify multi-databases according to their form of cooperation, we distinguish five integration levels with increasing strength of coupling. Consider Figure 2. This refines the known classification of [28], distinguishing between losely and tightly coupled federated database systems.

At the left most end, level 0 represents non-integrated multi-database systems. This is the weakest form of database coupling, where component systems are completely independent of each other. Cooperation is established ad hoc, by transactions, using objects from multiple databases. Within a transaction, each operation is uniquely mapped to one component ODBS.

At the right most end, level IV represents fully integrated distributed databases. Either, there exists only one single global database management system, or participating component systems don't have any local autonomy. Hence, objects can physically be distributed, although there exists only one logical database.

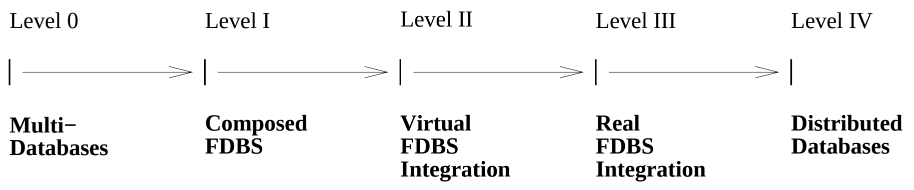

Fig. 2. Evolutionary Cooperation of Multi-Database Systems

Levels I to III are known as federated database systems (FDBS), where component ODBSs are integrated up to some degree, whereas some autonomy is lost. In the sequel, we focus on these three levels of FDBS.

4.1 Level I: Composed FDBS

Interoperability of this level is called schema composition. It is the elementary process to combine schemata of multiple local ODBS into one *composed schema*, and is therefore the foundation for establishing a federated database system. Schema composition places only minimal requirements on the degree of integration between participating systems. In fact, it basically just imports the names of all schema elements (variables, functions, types, classes, views) from component ODBSs and makes them globally available [26].

Example 9. The following statement composes two ODBSs, the library database *LibDB* and the student database *StudDB*:

```
define database GDB
    import LibDB, StudDB
end .
```

More precisely, composition of component systems DB_i into a federated system GDB combines the class and type systems of the local databases. First, the basic data types (integer, string, ...) of the different systems are unified. Second, the class and type hierarchies of the local ODBS 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**@GDB, a new bottom type **bottom**@GDB, and a new global top class **Objects**@GDB. The global type hierarchy is established, where all top types of the local ODBSs (**object**@DB_i) are made direct subtypes of the new global top element **object**@GDB. The new bottom type **bottom**@GDB is made common subtype of all local bottom types. No further subtype relationships are established.

[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 *Books* in *LibDB* has as globally unique name "*Books@LibDB*".

Similar, a global subclass hierarchy is composed, with the new top element **Objects**@GDB as common superclass to all local top classes **Objects**@DB_i.

Meta Schema Level: A global meta schema is created as well. This is the meta schema of the GDB. The meta schema of each DB_i contains meta types *variables, functions, types, class, views* and meta classes *Variables, Functions, Types, Classes, Views*, respectively (a detailed discussion of the COCOON meta schema is contained in [33]).

A global meta schema is created with types *variable@GDB*, *function@GDB*, *type@GDB*, etc. as well as new meta classes *Variables@GDB*, *Functions@GDB*, *Types@GDB* etc. The GDB meta schema has therefore exactly the same structure as that of each DB_i. Then, the local meta classes and meta types are made subclass/subtype of their global counterparts.

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

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

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

$$\mathbf{select}[\emptyset = \mathbf{select}[name(c) = name(s)](s : Students)](c : Customers)$$

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

$$\mathbf{select}[c \in Students](c : Customers)$$

Since the type of class *Students* is "student" and the type of c is"customer" and the two types customer and student 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 ODBs. Suppose, we want to store together with each student (of *StudDB*) the books (of *LibDB*), that he lent:

$$\begin{array}{l}\mathbf{extend}[lentbooks := \mathbf{select}[name(s) = name(lent(b))](b : Books)] \\ \qquad (s : Students)\end{array}$$

The new function *lentbooks* is an inter-objectbase link, from *StudDB* to *LibDB*.

4.2 Level II: Virtual FDBS Integration

Virtual integration of schemata is the next step in the process of evolutionary cooperation of database systems. It is based on the idea that views can be used to build a uniform, virtual interface over multiple databases (cf. e.g. [12, 14, 27]).

Since database integration is not the main topic of this paper, but should just be discussed in the context of evolutionary cooperation, we refer for more formal details on the mechanism (i.e., *same*-functions, global identity, solving structural and semantic conflicts) to [26]. We recall here some of the results, as far as they are necessary to understand differences between coupling levels.

Unifying Objects. Since component databases have been used independently of each other, one real world (entity) object may be represented by different database (proxy) objects in different component databases. 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 FDBSs [9]. We therefore need an additional notion: we say that two local proxy objects are *the same*, iff they represent the same real world (entity) object.

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

Example 10. To state that objects of class *Customers* of *LibDB* are identical with objects of class *Students* of *StudDB*, if they have identical names, for example, we have to defined the following *same*-functions:

$$
\begin{array}{l}
\textbf{extend}\ [same_{\text{LibDB,StudDB}} := \\
\qquad\qquad \textbf{pick}(\textbf{select}[name(c) = name(s)](s : Students))] \\
\qquad (c : Customers); \\
\textbf{extend}\ [same_{\text{StudDB,LibDB}} := \\
\qquad\qquad \textbf{pick}(\textbf{select}[name(s) = name(c)](c : Customers))] \\
\qquad (s : Students);
\end{array}
$$

Notice that the **pick** operator does a set collapse, that is, it takes a single object out of a set of objects ($\textbf{pick}(\emptyset)$ = "undefined").

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

We already mentioned that schema composition also constructs a global meta schema. To define correspondences between functions from different ODBSs, we make use of the fact that every function is represented by a meta object. Thus, integrating functions from DB_i and DB_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*@DB_i to meta type *function*@DB_j.

Example 11. To unify for example the functions *name@LibDB* and *name@StudDB*, the following *same*-function is defined on the meta schema of *GOB*:

```
extend [$same_{\mathrm{LibDB,StudDB}}$ :=
                 pick(select [$name(f)$ = "$name$" $\wedge$ $name(g)$ = "$name$"]
                                   ($g$ : Functions@LibDB))]
      ($f$ : Functions@StudDB);
```

In contrast to most other schema integration approaches that emphasis on solving semantic and structural conflicts among different systems [5, 29], we concentrate here on reducing schema integration to unifying meta objects.

We have now defined all prerequisites for virtual ODBS integration: we showed how to find "same" objects over multiple systems and discussed schema integration as unification of meta-objects.

Example 12. Following to the above method, we first compose the local schemata by importing *LibDB* and *StudDB*. Then, we extend the classes *Customers* and *Students* with *same*-functions. Finally, the meta class *Functions@LibDB* is extended to integrate *name@LibDB* and *name@StudDB* properties.

```
define schema GDB as
   import LibDB, StudDB ;
   define view Customers'@LibDB as extend ... ;       // see Example 10
   define view Students'@StudDB as extend ... ;       // see Example 10
   define view Functions'@LibDB as extend ... ;       // see Example 11
   define view Persons as Customers'@LibDB union Students'@StudDB ;
end ;
```

Now, we are ready to define views that span multiple ODBS, e.g., a view *Persons* as the union over the extended classes *Customers'@Lib* and *Students'@Stud*. The extent of this union view is defined to be the union of the objects of the base classes. However, if there would be a customer object and a student object, having equal names, they are defined through the *same*-function to represent 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'* and of *Students'* are disjoint, except of *name@LibDB* and *name@StudDB* functions that are defined to be the same, the type of the view is [*name*].

4.3 Level III: Real FDBS Integration

So far, component ODBSs are virtually integrated by unifying views, spanning over multiple component systems. We now show, how ODBSs can be further integrated, without completely giving up their local autonomy.

There are several disadvantages of virtual integration at level II, since cooperation was restricted to views only. E.g. functions with domain and range type in different component ODBSs, let's say DB_i and DB_j (cf. *same*-Functions), have only been allowed, if they are **derived** from a query expression. The state (values) of a **stored** inter-database functions can be formalized as a set of tuples $< argument, value >$, with *argument* as objects from DB_i and *value* as objects from DB_j. Consequently, such functions have not been possible, because their state could not be stored exclusively in one of these ODBSs. Furthermore, variables of a supertype of types of multiple ODBSs are not allowed at level II, since they can store objects of multiple databases, such that its state could not be hold in one specific ODBS. Level III removes this inadequacy.

The federation dictionary (FD). The main difference to the previous level is that real integration is not completely virtual, but yields a real global database state, which is store in the federation dictionary (FD). Whereas till now, the FD was exclusively used to store meta information (e.g. the view definitions of level II), from now on, we allow to store primary object data as well. We therefore enhance the possibilities of the federation dictionary.

From the technical point of view, the FDBS must be able to handle OIDs, that are extended (colored) by the identification of its original ODBS. This qualification becomes necessary, since for autonomy reasons, the component ODBSs must not be restricted in the way how they generate OIDs. The FD, being managed by the FDBS, must be able to store such colored OIDs. Notice that this does not mean that all objects from ODBSs are copied into the FD. However, if needed, local OIDs are store in the FD.

Cooperation level III now allows for global schema augmentation, which can be understood as capacity augmenting changes to the federated schema.

Example 13. Consider again the above example. It is now possible to define an inter-database function *favourite_book* from *StudDB* to *LibDB*, which is typically not derived, but stored:

define function $favourite\_book : student@StudDB \rightarrow book@LibDB$

The state of this function can neither be stored in *StudDB* nor in *LibDB*. A special case of that are stored *same*-Functions, like e.g.

define function $same_{LibDB,StudDB} : customer@LibDB \rightarrow student@StudDB$

Notice that we really get advanced possibilities, since we do not need to know a query, to retrieve *same* objects from other ODBSs. Assume variable c, holding a customer object from *LibDB* and s, holding a student object from *StudDB*. We can now directly assign these object as being the same:

define var c : *customer@LibDB;*
define var s : *student@StudDB;*
set$[same_{LibDB,StudDB} := s](c)$

In general, schema augmentation at level III is not limited to views. Additional possibilities to augment the global schema are: (i) to define global object types, that are subtypes of different ODBSs and therefore contain functions from muliple ODBS, (ii) classes that are subclasses from different ODBSs, and (iii) variables that can hold objects from multiple ODBSs as values. Notice that these global schema objects are only visible to the FDBS and are not known to a local ODBS.

Example 14. Consider the following type, which is a subtype of *customer* and *student*, both of which are from different ODBSs:

define type *emplstud* **isa** *customer@LibDB, student@StudDB*

The following variables are not visible to one of the component ODBSs. o can store any object from any system, and A can hold objects from different ODBSs at the same time.

define var o : **object**@*GDB*;
define var A : **set of** *emplstud*

5 Conclusion

We have discussed selected issues of evolution in and beyond database systems. We identified a number of elementary evolution mechanisms, that support adaptation and integration of ODBSs.

Dynamic object evolution provides the possibility that a database object can represent real world objects in different roles over time. Logical data independence allows for definition of views and subschemata, forming the possibility to simulate some schema updates. Uniform treatment of data and meta data objects, and the possibility to apply generic query and update operations on all levels of objects, can be used to perform schema evolution.

5.1 Evolution in Object Databases

First, on single ODBS is considered. Based on a formal evolution model, we showed, how the above mechansims are used to build a schema manipulation languages (SML). This language differs in two points from most other database systems supporting schema evolution, e.g. ORION [4], O_2 [35], and GemStone [18]:

- We did not invent special purpose schema update methods, but built elementary operations on top of our update language. Thus, the semantics of local schema changes is formally defined.

- We considered local schema changes as well as global database restructurings. We used snapshots to define new schema objects from existing ones.

Information capacity is shown to be a very important issue in context of database evolution, because it characterizes the impact of schema changes on existing instances as well as application programs, using the changed database.

5.2 Evolution beyond Object Databases

Second, consideration of one single ODBS is extended towards evolutionary cooperation between multiple autonomous object databases. Five levels of federated database systems (FDBS) with increasing strength of cooperation are described. Each coupling level is characterized by the use fundamental evolution mechanisms.

These five evolutionary levels of multi-database interoperability are well suited to compare multi-database system approaches among each other:

- Level I: we presented schema composition as the main mechanism of level I cooperation. This corresponds to the *connect to*-facility of existing relational multi-database systems, like e.g. Oracle SQL*Net or Ingres/Star.
- Level II: virtual database integration via views and other CP-schema evolutions is the main characteristics of cooperation level II. So in addition to the presented view mechanism, also the systems SuperViews [14] and Multibase [12] perform integration of this level, as well as the generalization constructs of VODAK [27, 15] and derived *unifier-/image*-Functions in Pegasus [3, 2].
- Level III: real integration of databases is done by CA-schema evolutions. Although this level is not considered in many systems, stored *image*-functions in Pegasus and the *merge*-operation of O*SQL [13] are additional mechanism of cooperation level III.

Acknowledgement

The authors wish to thank Hans-Jörg Schek who has been working with them for years on the topics described here. Many of the ideas worked out over that time have been inspired by him.

References

1. S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62(1,2), December 1988.
2. R. Ahmed, J. Albert, W. Du, W. Kent, W.A. Litwin, and M.-C. Shan. An overview of Pegasus. 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. 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.

4. J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *ACM SIGMOD Record*, 15(4), February 1987.
5. C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4), December 1986.
6. *Proc. IFIP DS–5 Semantics of Interoperable Database Systems*, Lorne, Australien, November 1992.
7. D.H. Fishman et al. Overview of the Iris DBMS. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* ACM Press, New York, 1989.
8. R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3), August 1986.
9. W. Kent. The breakdown of the information model in multi-database systems. *ACM SIGMOD Record*, 20(4), December 1991.
10. W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, Cambridge, MA, 1990.
11. C. Laasch and M. H. Scholl. A functional object database language. In *Proc. 4th Int'l Workshop on Database Programming Languages (DBPL-4)*, Manhatten, New York, August 1993.
12. 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.
13. W. Litwin. O*SQL: a language for multidatabase interoperability. In DS5 [6].
14. A. Motro. Superviews: virtual integration of multiple databases. *IEEE Trans. on Software Engineering*, 13(7), July 1987.
15. 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.
16. Object Design Inc., Burlington, MA. *ObjectStore Rel. 2.0, Reference Manual*, October 1992.
17. ONTOS Inc., Burlington, MA. *ONTOS DB 2.2 – Reference Manual*, February 1992.
18. D.J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Proc. Int'l Conf. on Object-Oriented Programming Systems and Languages (OOPSLA).* ACM Press, October 1987.
19. E. Radeke. Object management in federated database systems. Internal report, CADLAB, Paderborn, Geramny, 1993.
20. 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.
21. H.-J. Schek and M.H. Scholl. Evolution of data models. In A. Blaser, editor, *Proc. Int'l Symposium on Database Systems for the 90's*, Berlin, Germany, November 1990. LNCS 466, Springer Verlag, Heidelberg.
22. H.-J. Schek and A. Wolf. Cooperation between autonomous operation services and object database systems in a heterogeneous environment. In DS5 [6].
23. M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, and M. Tresch. The COCOON object model. Technical Report 193, ETH Zurich, Dept. of Computer Science, December 1992.

24. 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, Deutschland, December 1991. Springer, LNCS 566.
25. M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. 3rd Int'l Conf. on Database Theory (ICDT)*, Paris, France, December 1990. Springer, LNCS 470.
26. M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, San Mateo, California, 1993.
27. M. Schrefl. *Object-oriented database integration.* PhD thesis, Technische Universität Wien, June 1988.
28. A.P. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneuos, and autonomous databases. *ACM Computing Surveys*, 22(3), September 1990.
29. S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *The VLDB Journal*, 1(1), July 1992.
30. M.R. Stonebraker. The 3rd generation database system manifesto. In *Proc. IFIP TC2 DS-4 Conf. on Object-Oriented Databases – Analysis, Design & Construction*, Windermere, UK, November 1990. North-Holland.
31. M. Tresch. *Dynamic evolution of independent and cooperating object databases.* PhD thesis, University of Ulm, Germany, 1994.
32. M. Tresch and M. H. Scholl. Schema transformation without database reorganization. *ACM SIGMOD Record*, 22(1), March 1993.
33. M. Tresch and M.H. Scholl. Meta object management and its application to database evolution. In *Proc. 11th Int'l Conf. on Entity-Relationship Approach*, Karlsruhe, Germany, October 1992. Springer, LNCS 645.
34. M. Tresch and M.H. Scholl. Schema transformation processors for federated objectbases. In *Proc. 3rd Int'l Symp. on Database Systems for Advanced Applications (DASFAA)*, Daejon, Korea, April 1993.
35. R. Zicari. A framework for schema updates in an object-oriented database system. In *Proc. 7th Int'l IEEE Conf. on Data Engineering (ICDE)*, Kobe, Japan, April 1991.