Schema Transformation without Database Reorganization

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

Abstract

We argue for avoiding database reorganizations due to schema modification in object-oriented systems, since these are expensive operations and they conflict with reusing existing software components. We show that data independence, which is a neglected concept in object databases, helps to avoid reorganizations in case of capacity preserving and reducing schema transformations. We informally present a couple of examples to illustrate the idea of a schema transformation methodology that avoids database reorganization.

1 Introduction

Schema transformations usually follow from evolutionary changes of the logical object structure, that is, the database schema. These changes arise due to many reasons: design flaws and lacks, changed real world domain, reusability and extensibility of classes, or cooperation with other systems. In this article, we assume that schema transformations are performed after the database was populated with objects (instances), and application programs have been implemented.

Schema transformations can be classified according to their impact on the object modeling capacity [AH88]: *Capacity preserving* transformations do not affect the modeling possibilities. That is, the same potential set of objects can be represented after transformation. *Capacity reducing* transformations reduce the modeling possibilities, such that with these transformations, information is lost. Finally, *capacity augmenting* transformations enhance the information contents of the schema.

Why should reorganizations be avoided? There are two major reasons: (i) Updates to the schema must propagate to physical updates of existing instances, since databases are supposed to be populated with objects. This is extremely time consuming, such that sophisticated implementation techniques are applied to delay propagation[1]. (ii) Compatibility of the transformed schema with existing applications must be guaranteed, since application programs are still running with the "original" database.

How can reorganizations be avoided? A concept that is well-known in relational databases tends to be forgotten in object-oriented systems: *data independence*. Very few o-o systems (and hardly none of the existing ODBMS products) include a notion of external schema or separate conceptual from physical representations of objects. This is a pity, since object identity alleviates with updating external schemata, and o-o concepts, such as encapsulation of objects into ADTs with type-specific methods, open new possibilities for data independence.

Can reorganizations always be avoided? In most cases: schema transformations that are capacity preserving and capacity reducing can always be avoided. However, the latter can produce some nondeterminism. Unfortunately, capacity augmenting transformations require some

[1] Cf. [HVZ90] for a performance analysis of different propagation strategies.

propagation to the physical level, such that reorganizations cannot be avoided completely.

2 Logical Data Independence in Object Databases

Data independence is commonly established by defining external schemata, i.e., schemata that are derived (computed) from the logical level. This process typically proceeds in two steps: first, a set of *views* are defined, and second, classes and views are put together into a *subschema*. Although there is no widely accepted concept of views in object databases, they are often understood as virtual (derived) classes, declared for example as

define view v **as** e

where e is a query expression. Since classes (and views) in a schema (or subschema) are arranged in a type and/or class hierarchy, the definition of new views involves positioning them in these hierarchies. Depending on the power of the view definition mechanism, this can lead to hard problems. In our object database model COCOON [SS90], which separates types and classes, positioning of the view's type in the type lattice is straightforward and can be determined at compile-time. Positioning the view's extent in the class hierarchy involves testing of predicate subsumption, which is undecidable in general. Therefore, an incomplete classification algorithm is used.[2] Our language offers the following possibilities to define views, i.e. support logical data independence [SLT91]:

Derived Class Extents: The extent of a class – the actual set of objects in the class – can be restricted by a predicate p, using selection views (**define view** v **as select**$[p](e)$). The extent of view v is a subset of the base class' objects, namely those satisfying the predicate. A selection view v becomes a subclass of the base class c, with fewer objects, but the same type.

Derived Object Types: The type of objects in a class can be restricted (generalized) by projection views (**define view** v **as project**$[f,\ldots](e)$). The changed type is given by the set of functions $f,\ldots$ in the projection list, which must be a subset of the original type's functions. The projection view v is a superclass of base class c, with a smaller set of applicable functions, but unchanged extent.

Another possibility to derive object types, is the type cast operator. Thus, together with projections, the range type of a particular function $f : dom \rightarrow ran$ can be restricted using type cast ($f :: T$). However, its application is limited to type generalization, such that the changed range type T of f must be a supertype of the original type ($T \preceq ran$).

Derived Object Properties: In contrast to projections, new computed functions can be added to the type of objects using extend views (**define view** v **as extend**$[f{:=}\ expr,\ldots](e)$). In general, arbitrary methods, e.g. written in C++, can be added. The view v becomes a subclass of c, with the same extent, but additional functions.

Combinations: Other algebra operators, like union and intersection (**define view** v **as** e **union** | **intersect** e;), or cascading of selection, projection, etc. modify both, class extents and object types. E.g., the extent of union views is the union of base class' extents, and the type consists of those functions that are common to all base class' types.

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 (cf. [Run92]).

3 Schema Transformation using Object Views

We argued, that data independence in object database systems is an important property. We presented two concepts to realize data

[2] Positioning of virtual classes is essential for object-oriented systems. However, it is not considered in most proposed view mechanisms. An alternative approach called "view derivation hierarchy" is presented in [Ber92].

independence: views (virtual classes) and subschemata (subsets of classes and views). We now show, how these concepts can be used to implement a methodology for schema transformation that avoids database reorganization whenever possible. It can be used to perform any capacity preserving and reducing transformation:

Step 1: Extend given schema by a set of views that simulate the desired schema reorganization. Position the views in the schema, i.e., the type and class hierarchies.

Step 2: Define a subschema of the extended schema, by selecting a set of classes and views that corresponds to the restructured schema. Make the subschema closed.

In the sequel, we give a brief – intentionally informal – overview of the methodology driven by a set of schema transformation examples.

3.1 Capacity Preserving Transformations

Renaming of classes and functions are obviously capacity preserving transformations. However, more interesting are equivalences of classification hierarchies. Consider classes persons P, youngsters Y, and workers W, drawn as white cycles in Figure 1. Every person has two properties, *name* and *age*. Youngsters are sufficiently defined as those persons with $age < 30$. Hence, class Y is a subclass of P. Another subclass of P is class W, since workers are defined as those persons with $age < 65$.

Equivalently to the above class hierarchy, we could position class Y as subclass of W instead of P. This follows directly from the subsumption of class predicates, since $age < 30 \Rightarrow age < 65$.

Following the above methodology, this capacity preserving transformation can be realized using object views. A selection view is defined as

define view Y' **as select** $[age < 30]$ (W)

and positioned as subclass of W (drawn as gray circle in Figure 1). The subschema, showing the transformed situation, is marked in Figure 1 as

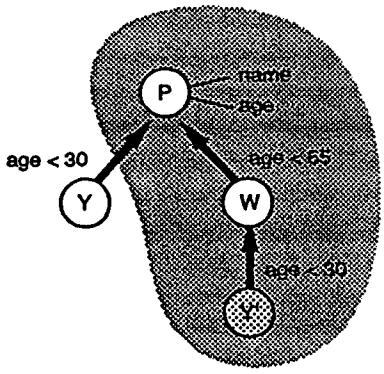

Figure 1: The classification example

the shaded area including classes P, W and view Y'.

3.2 Capacity Reducing Transformations

Many object models distinguish objects from values. The transformation from an object-oriented modeling to a value-oriented modeling of the same situation is an example for a capacity reducing transformation.

Consider classes persons P and addresses A in Figure 2. For each person, function *addr* returns an address. Addresses are an aggregate of three attributes *city, street, no*. The intended transformation is to "unnest" *city, street, no*, such that they are made direct attributes of class persons.

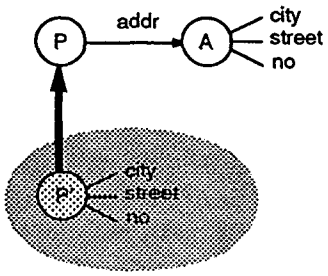

Figure 2: The object–value–transformation example

To model this schema transformation, an extension view P' is created as subclass of base class P. For each function of class A, an additional function with the same name is created. Its value is computed from the *addr* values for *city, street, no.* The final subschema realizing the transformed base schema is marked as the shaded area in Figure 2. In this simple example, the subschema consists of just one view, P'.

define view P' **as extend**
[*city* := *city*(*addr*(*p*));
street := *street*(*addr*(*p*));
no := *no*(*addr*(*p*))] ($p : P$);

Notice that object–to–value transformation is capacity reducing, and therefore, the information is lost, that the same address (represented by one address object) can be the address of multiple persons. Consequently, the reverse transformation, value–to–object, is capacity augmenting.

3.3 Capacity Augmenting Transformations

Transformations that are capacity augmenting cannot be simulated using views. Consider, e.g., the polyandry example of [AH88]. It assumes a culture, where families F are either a single woman W, or couples C of a woman W and a man M. Suppose that due to a change in laws, families are now defined differently: we allow only couples of exactly one woman and an arbitrary number of men (see Figure 3).

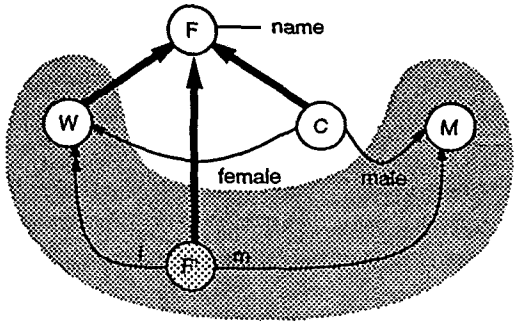

Figure 3: The polyandry example

This schema transformation includes the extension from zero or one man to multiple men per family, which is capacity augmenting. The solution shown as shaded area in Figure 3 does therefore not really perform the transformation. It just maps the actual schema into the changed situation, whereas it would not allow to store families with multiple women.

3.4 The Need for Meta-Information

More sophisticated transformations either need additional query language operations (see e.g. OSQL of IRIS [CL92]), or the meta-database must be dynamically accessible to the query language at run-time [TS92].

Consider Figure 4. Assume, that a given class of articles, A, has a variable number of disjoint subclasses, e.g., screws S, nails N, bolts B, ... The desired transformation shall result in a new function *kind* : $A \rightarrow$ *string*, that returns, for each article, the name (as a string, $'S', 'N', 'B', \ldots$) of the subclass to which it belongs.

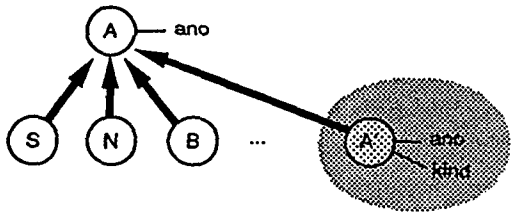

Figure 4: The subclass–function–transformation example

Since the number of subclasses of A is not fixed, the view mechanism must be able to determine it at run-time. Therefore, the extend view A' defines an additional derived function *kind*.

define view A' **as extend**
[*kind* := *name*(**pick**(**select**[$a \in s$]
(s : *subclasses*(A))))]
($a : A$)

For each article a, it first computes (from the meta-database) all subclasses of A, then selects those that include the article a (due to disjointness of subclasses, this is at most one class), and

finally assigns the class' name to *kind*. Again, the subschema showing the transformation is very simple, holding just one view A'.

3.5 Object and Meta-Object Unification

The object-oriented paradigm demands that every real world object is mapped into exactly one database (proxy) object. However, this is an ideal situation, and in practice, there are many reasons why this did not happen [Ken91]. Consider Figure 5a. Two classes $A1$ and $A2$ have been designed to model articles, both have a function article number *ano*.

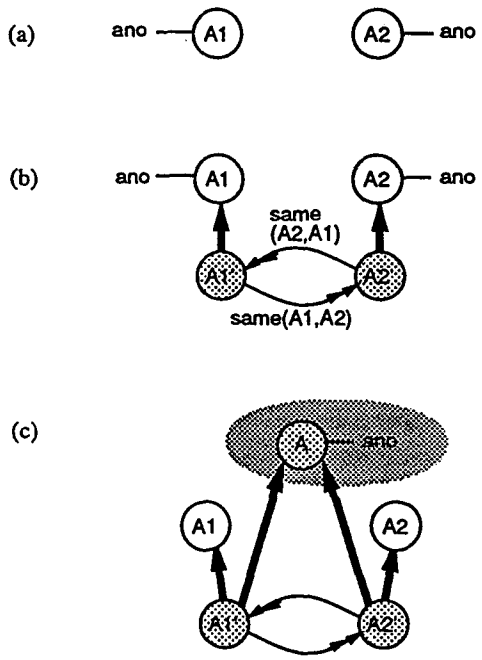

Figure 5: The "same"-object / "same"-function example

Now assume, we want to combine these two classes into a class articles A, such that, if there is an article $a1$ in $A1$ and $a2$ in $A2$ with the same article number, $ano(a1) = ano(a2)$, these objects are supposed to model the same real world entity article and should therefore appear only once in class A.

This process is called *(proxy) object unification*. We use extend views to define *same*-functions that identify objects from one class that are "the same" as an object from another class. The semantics of *same*-functions is known to the system, such that basic mechanisms, like object equality or the algebra operators, handle it correctly [SST92].

define view $A1'$ **as extend**
$[same_{A1,A2} :=$ **pick(**
select$[ano(a2) = ano(a1)](a2 : A2)]$
$(a1 : A1)$;
define view $A2'$ **as extend**
$[same_{A2,A1} :=$ **pick(**
select$[ano(a1) = ano(a2)](a1 : A1)]$
$(a2 : A2)$

Now that objects have been unified with *same*-functions (Figure 5b), we can define a union view, A, to generalize $A1'$ and $A2'$, such that objects with the same article numbers are represented only once in view A.

define view A **as** $A1'$ **union** $A2'$

The subschema showing the transformation contains nothing but view A (Figure 5c). The next issue is to unify functions, that is, to specify that function $A1.ano()$ and $A2.ano()$ are the same functions. The same *same*-functions used to unify objects, are now used to unify meta-objects that represent functions (see [SST92] for details).

3.6 Summary of Elementary Schema Transformations

An early reference to schema evolution is Banerjee's taxonomy of primitive schema updates for the ORION data model [BKKK87]. Similar taxonomies can be found for any other data model.

The taxonomy of the COCOON data model is organizaed differently: we group elementary schema updates according to their impact on the capacity of schemata, and thereby distinguishing between transformations of the function hierarchy, the intensional part of the class (Figure 6), and of the classification hierarchy, the extensional part (Figure 7).

In the two figures below, transformations from left to right (Fig. 6: $a \rightarrow b$, $a \rightarrow c$,

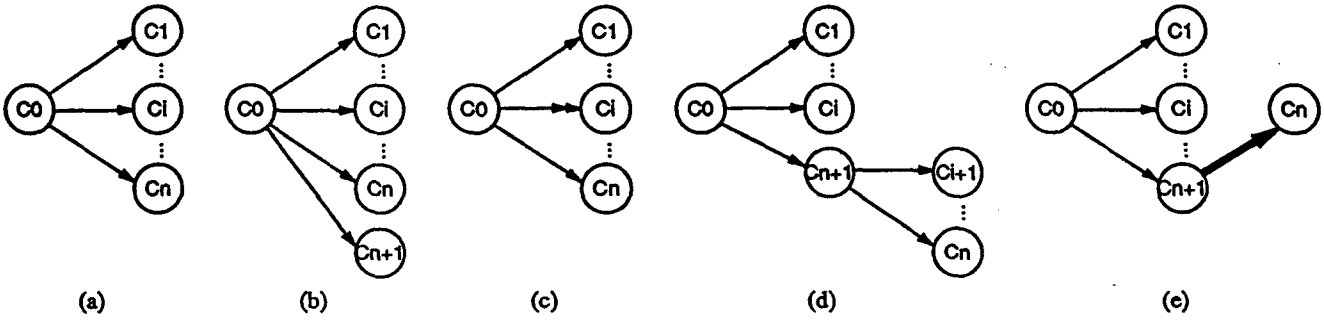

Figure 6: Function hierarchy transformations (class intension)

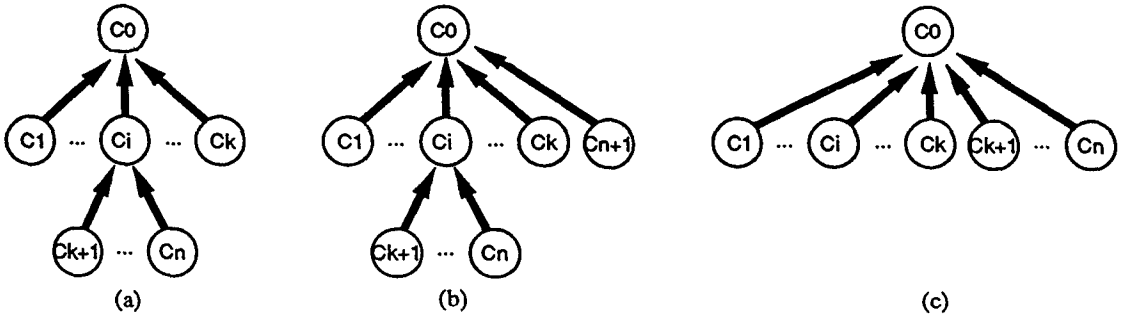

Figure 7: Class hierarchy transformations (class extension)

$a \rightarrow d$, $a \rightarrow e$; Fig. 7: $a \rightarrow b$, $a \rightarrow c$) are capacity augmenting, and those from right to left (Fig. 6: $b \rightarrow a$, $c \rightarrow a$, $d \rightarrow a$, $e \rightarrow a$; Fig. 7: $b \rightarrow a$, $c \rightarrow a$) are capacity reducing. Notice, that there are always two corresponding tranformations that are inverse to each other, e.g. in Fig. 6: transformation $a \rightarrow b$ is inverse to $b \rightarrow a$.

4 Conclusion

We presented an – intentionally example driven – tour through a methodology for schema transformations. We argued for more data independence in object database systems and showed how this helps to avoid physical database reorganizations due to capacity preserving and reducing schema transformations.

This presentation is neither formal nor complete. Precise definitions of the terms capacity preserving, reducing, and extending must follow. Furthermore, to guarantee compatibility of an augmented schema with existing applications, we have to show that our methodology can be used to define "compensating views". More work will consider updatability of schemata that are transformed using views. A starting point for view updates in COCOON was given in [SLT91].

References

[AH88] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62(1,2), December 1988.

[Ber92] E. Bertino. A view mechanism for object-oriented databases. In *Proc. 3rd Int'l Conf on Extending Database Technology - EDBT'92*, Vienna, Austria, March 1992. Springer LNCS 580.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD Record 1987*, San Francisco, California, February 1987. ACM Press.

[CL92] J. Chomicki and W. Litwin. Declarative definition of object-oriented multidatabase mappings. In [DOM92].

[DOM92] *Proc. Int'l Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[HVZ90] G. Harrus, F. Velez, and R. Zicari. Implementing schema updates in an object-oriented database system: A cost analysis. Technical report, GIP Altair, France, 1990.

[Ken91] W. Kent. The breakdown of the information model in multi-database systems. *ACM SIGMOD Record*, 20(4), December 1991.

[Run92] 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.

[SLT91] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991. Springer LNCS 566.

[SS90] M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. 3rd Int'l Conf. on Database Theory (ICDT'90)*, Paris, 1990.

[SST92] M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In [DOM92].

[TS92] 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.