Fifth International Workshop on the Deductive Approach to Information Systems and Databases : September 19 - 21, 1994, Aikuablava - Costa Brava (Catalonia) / Antoni Olivé ... - Barcelona : Universitat Politècnica de Catalunya, 1993. - S. 1-23. - (Report de recerca ; 94,28,R)

Change Management in Objectbases

Christian Laasch, Marc H. Scholl
Faculty of Computer Science, Databases and Information Systems
University of Ulm, D–89069 Ulm, Germany
e-mail: {laasch, scholl}@informatik.uni-ulm.de
Tel: +49-731-502-4135, Fax: +49-731-502-4134

Abstract

In OODBMSs type-specific methods are used for manipulating objects, in order to maintain the consistency of the database. This is, however, of little help for the method implementor as far as the model-inherent constraints are concerned. We propose a set of *generic update operations*, including operations for object evolution that maintain integrity constraints that can be expressed in database schemas. On the one hand integrity constraints such as types, class memberships, subtype-, subclass-relationships, class predicates, and inverse functions are kept consistent after update operations, on the other hand the capabilities to express semantics in a schema are chosen such that such a set of update operations exists. The update operations can be used for implementing type-specific update methods or directly by applications. We present an approach to consistently define update semantics for an object model including classes, views, and variables that is based on necessary and sufficient predicates akin to defined concepts in KL-ONE style languages.

1 Introduction

One of the main objectives of the object-oriented approach to modeling is that clients use only type-specific methods for manipulating objects, in order to guarantee consistency when updating objects. The more powerful method implementations can be, the richer the application semantics that are encapsulated in these. While type-specific methods can provide integrity-preserving updates for clients of the methods, they alone provide little help for the method implementor as far as integrity preservation is concerned.

To facilitate the implementation of consistency-preserving methods, OODB models should provide a set of generally applicable *generic update operations* that maintain the model-inherent integrity constraints. They can be used by type implementors to define type-specific update methods. Furthermore, many applications might make direct use of these generic update operators.

Generic update operations have been used in the relational model (e.g., in the SQL language). For a long time they ignored the maintenance of integrity constraints, i.e., referential integrity and uniqueness of a primary key. Up to now, there is no satisfactory solution for different update policies of several foreign key relationships. Since object-oriented data models offer a much broader scope of built-in semantics, there is an even stronger need for generic update operators that account for these semantics. If methods are the only way to guarantee consistent database updates, the method implementor has to take care of all integrity constraints. Moreover, changing the schema, for example adding new constraints, requires additional checks on all methods. We argue that it is crucial that advanced database models offer not only more capabilities to statically *specify* semantics but also offer update operations that dynamically *maintain* consistency during modifications.

In addition, we provide clean semantics for em set-oriented (or bulk) updates. Set-orientation of updates in object-oriented models is important, because of the following reasons: First, if bulk-updates are applicable to query results as well as to named collections (such as classes and function results), the gap between one-object-at-a-time updates and set-oriented queries is eliminated. Secondly, set-oriented updates allow for more efficient update processing (e.g., parallel execution of updates), since the system gets more flexibility for execution.

However, in contrast to the relational model it is not sufficient that elementary update operations are set-oriented, if methods, which usually consist of sequences of elementary update operations, are provided as means to express complex updates that respect application-specific integrity constraints. In order to allow for the set-oriented application of methods, we separate set-orientation from the elementary operations by providing an "iterator". In contrast to other approaches, our iterator allows to apply sequences of updates in a set-oriented way with *deterministic* semantics [LS93a].

In this paper, we present an approach toward defining such a collection of generic update operators in the context of our object model COCOON [SS90]. Discussing update operations in the context of a particular model is crucial, because of their interaction with the integrity constraints that are provided by the built-in semantics. On the one hand, update operations should not yield inconsistent database states, which do not fulfill all integrity constraints expressed in the schema definition. On the other hand, the update operations have to be powerful enough such that all possible, consistent states can be reached. However, many object models, especially those influenced by semantic data models, provide powerful capabilities to express semantics, but only allow for rather rudimentary updates; i.e., elementary update operations might result in inconsistent database states and rollback at the end of transaction is the approach to enforce consistency, if it was not satisfied by the last elementary update operation.

As an example for the mismatch of capabilities to express semantics and update operations let us consider two function f and g that map instances of type A to instances of B and vice versa:

define function $f: A \rightarrow B$ **not null**;
define function $g: B \rightarrow A$ **not null**;

If both functions are specified as total mappings, which is allowed in most models, an empty database state cannot be populated without inconsistent intermediate states, because none of the models provides the simultaneous creation of objects belonging to different types. Thus, either the update operations are not *adequate*, because consistent database states cannot be reached (by integrity-preserving updates), or the update operations are not *closed* against the integrity constraints, if inconsistent states can be reached (e.g., after the creation of the first instances of A or B).

The usual approach is to use update operations that are not closed, and to defer checking of integrity constraints until end of transaction. Thus, there is no difference between modell-inherent and application specific integrity constraints. In our opinion, however, sematically powerful models should allow for different levels of constraints: (1) modell-inherent constraints that are always fulfilled, (2) method specific constraints, which, for example, can be expressed by pre- and post-conditions and are guaranteed outside the method, (3) and more general constraints that can only be checked at end of transaction.

In order to provide an adequate set of update operations that is closed against the integrity constraints expressed in the database schema, we follow a conservative approach allowing for fewer capabilities to express semantics, but providing rather powerful update operations that keep databases consistent. Thus, the built-in semantics of the COCOON model contains typing constraints, class membership, subtype- and subclass-relationships, class predicates, views, and inverse functions. The key idea for change management in object-oriented models is to separate the model-inherent semantics into constraints e.g., (sub-)typing, and derivation rules like view definitions and derived functions. The effect of update operations is then captured either by their definition that guarantee manipulating the association of objects to types or by re-evaluating these class predicates. That is, we integrate the techniques of automatic classification known from knowledge representation systems and a strong type system form object-oriented programming languages.

Two additional aspects of change management, which are motivated by updates on the schema level instead of the object level, will not be covered by this paper: Schema evolution and object migration. Schema evolution as means to provide new (additional) requirements can be accomplished by update primitives that are applied to the schema level [TS92]. If the schema level itself is represented by elements of the model (i.e., by objects and functions), we can exploit the generic update operations for the construction of schema-evolution primitives [SST92]. These can be used on the one hand in knowledge-based applications that require flexible schemas, on the other hand for integration of multiple (heterogeneous) databases [TS93, ST94]. Object migration describes the consequences of schema evolution on the instance level [RS94]. This is, for example, restructuring of instances within a database, or more sophisticated mappings between different models in case of the integration of heterogeneous databases.

The organization of the paper is as follows: in Section 2 we review the concepts of the COCOON object model. We summarize the basic terminology and the capabilities to express integrity constraints. Section 3 presents the semantics of generic update operations. In Section 4 it is shown how classes, class predicates, views and inverse functions are represented, such that automatic classification of objects guarantees these constraints. This representation also allows for the application of update operations to views. A comparison to related work is presented in Section 5 before we conclude in Section 6.

2 Basic Concepts

COCOON is a so-called "object-function" model, similar to IRIS [WLH90] or DAPLEX [Day89], for example. The main constituents of the COCOON model are: *objects*, *functions*, *types*, and *classes*.

Objects are instances of *abstract object types* (AOTs, see below). They are pure abstractions, in the sense that none of the descriptive information "associated with" an object is considered to be "part of" the object in any sense. Rather, *functions* (or methods, see below) are used as the uniform abstraction of stored fields, computed attributes, and relationships.

Data (or **Values**) are distinguished from objects, similar to [Bee89]. They are instances of concrete *data types* (see below).

Functions model the AOT-specific operators. They are the abstraction of side-effect free retrieval functions, called *properties*, and update *methods* with side-effects, that is, the AOT-specific operators. Properties are further separated into *stored properties* (attributes) and *computed properties*. The later ones are either *derived* by a COOL expression (derived property) or using any *foreign* programming language (e.g., Modula-2, C++). We do not want to distinguish stored and computed (derived) functions without side-effects here, since for many situations, it is unimportant whether a function value is stored or computed, e.g. for queries. Thus, we consider this terminology as a higher level of data independence to hide implementation facts from the logical database schema.

All functions are described by a name and signature (domain and range type). They are the interface operations of type instances. The implementation of functions is specified separately. The point behind our more abstract view is that we want to leave it up to the process of physical database design to make the decisions about what to store and what to derive (of course, there are restrictions on these decisions, so we can mainly decide to materialize derived functions trading update effort for retrieval speed).

In order to express general relationships, functions may be set-valued. Furthermore, two functions may be defined as being inverses of each other. Of course, these integrity constraints are enforced by the system during updates. The example,

define function *name* : *person* → *string*;
define function *age* : *person* → *integer*;
define function *address* : *person* → *city*;
define function *empl* : *employee* → *company* **inverse** *staff*;
define function *staff* : *company* → **set of** *employee* **inverse** *empl*;

defines first three functions with domain type *person*, where *name* and *age* have primitive range types (*string, integer*), and the third one has an abstract range (*city*). The last two functions are inverses of each other, such that the constraint $x \in$ *staff(empl(x))* always holds.

Types are separated into (concrete) *data types* and (abstract) *object types*. Data types are either *primitive* (e.g., integer, real, string, boolean)[1] or *constructed* (e.g., set, tuple, function). Non-constructed types, namely primitive and abstract ones, are called atomic types.

Object types describe the common interface of all instances of that type, the set of applicable functions. So the definition of an object type usually consists of two parts: a type name and a set of functions.[2] The signature of the included functions can be defined separately, using **define function**, or together with the type definition. The example,

define type *person* **isa object**= *name, age, address* ;
define type *city* **isa object**= *cname* : **string**,
zipcode : **integer**,
population : **integer**,
has_comp : **set of** *company* **inverse** *location* ;
define type *company* **isa object**= *ident* : **string**,
location : *city* **inverse** *has_comp* ,
branches : **set of** *company* ,
staff ;

defines three object types. The interface of the type *person* has three applicable functions, *name, age*, and *address*. In contrast, some of the functions of the types *city* and *company* are defined together with the type.

Types serve several purposes: (i) they represent a "repository" of possible values (called the *domain*, an intensional notion of type); (ii) they are used by the compiler for type checking (i.e., assuring that only "(type) valid" expressions are ever executed.

[1] In other OODMs, primitive types are also called printable, literal, or base types.

[2] As we will see later, queries like projections can dynamically produce new types. These are unnamed, but their set of functions can be derived from the query by type inference rules.

For example, we would not allow to compute the square root of a string. Finally, (iii) types are often also used as containers (collections) for those values of that type, which are currently "in use" in the database (an extensional notion). Notice that we do *not* need this aspect of types in general, but sometimes use it for *object* types (where this will be called the "active domain" of the type).

Subtyping allows a new type to be defined as a subtype of an existing one. Every instance of a subtype is also an instance of its supertypes. This is called *multiple instantiation*. Abstract object types are arranged in an inheritance hierarchy (actually, due to multiple inheritance, not a strict hierarchy). The subtype hierarchy is essentially defined by the superset relationship between the sets of functions (*fcns*) defined on the subtype as compared to its supertype(s):

$$t \preceq t' \iff \mathit{fcns}(t) \supseteq \mathit{fcns}(t')$$

A subtype inherits all the functions of all its supertypes and (usually) adds new ones.

define type *employee* **isa** *person* = *salary* : **integer**,
ssecno : **integer**,
empl ;

The new object type *employee* is defined as a subtype of *person*. Hence, it inherits its functions and adds new ones, *salary, ssecno*, and *empl*, which has been previously defined. Each instance of type *employee* is also an instance of *person*. This reflects the second aspect of subtyping, namely the set inclusion between the associated domains (set of all possible instances).

Subtyping between abstract object types defines a partial order, which can internally be extended by unnamed types such that there is a type for each element of the powerset of function names. The AOTs together with these unnamed types form a *type lattice*, such that for any two types their lowest upper bound and greatest lower bound is always defined. The top element of the lattice is the most general type **object** where no user-defined function is applicable (therefore, all instances of defined abstract types in the database are also instances of **object**), the bottom element is the type ($\perp$) that is associated with the set that includes all functions.

We allow multiple inheritance, that is, types may have more than one supertype. We assume that naming conflicts have already been resolved (for instance, by prefixing function names with type names).

Classes are strictly distinguished from types in the sense that types are interface specifications (a collection of functions), whereas classes are *containers* for objects of some type, see also [ACO85, Bee89]. So, each class, C, represents a (typed) set of objects and associates the member type, $\mathit{mtype}(C)$, to the objects in the set $\mathit{extent}(C)$. The extent of a class is manipulated (so as to include or exclude member

objects) either explicitly by corresponding operations or implicitly by membership predicates on objects' properties. For example:[3]

define class *Persons* : *person* ;
define class *Cities* : *city* ;
define class *Companies* : *company* ;

The three classes *Persons, Cities*, and *Companies* are defined as containers for objects of type *person, city*, and *company*, respectively. So, for example, *mtype(Persons)* = *person*.

Classes represent polymorphic sets: Members may be instances of several other types, particularly subtypes, in addition to the member type. Type checking in the COOL language always refers to the unique member type of a class, since classes are the primary operands of COOL operations. Due to the separation of types and classes, there may be any number of classes for a particular type.

As the result of the design process it is most likely to obtain one class per type. However, it is possible to define several classes over the same type, that is, to distinguish between multiple collections of objects over that type. On the other hand, a type T might serve the only purpose to "factor out" commonalities of the subtypes, such that only collections of the subtypes, but not of T itself are explicitly maintained in a database (cf. opaque types, abstract classes in some OOPLs).

Unlike in many other models, we do not explicitly maintain "type extents" (active domains), that is, the sets of *all* instances for each type. Rather, classes are used as named, user-defined containers. However, we can always derive the active domains of a type T by a query on class *Objects* selecting those members that are of type T.

Subclassing. There are several choices how to define a subclass relationship. Depending on whether the member types of two classes are the same or one is a *subtype* of the other, and depending on whether the extent of one class is a *subset* of the extent of the other. That is, we have two known relationships to consider: subtype and subset. We will always distinguish carefully which one of them holds, since they are often correlated, but they need not be. We will speak of a subclass relationship $C_1 \sqsubseteq C_2$, iff for two classes the following predicate is true in any database state:[4]

$$mtype(C_1) \preceq mtype(C_2) \textbf{ and } extent(C_1) \subseteq extent(C_2).$$

Usually, at least one of the ordering relationships ($\preceq$ or $\subseteq$) will be proper. Continuing the example:

define class *Persons* : *person* **some** *Objects*;
define class *Employees* : *employee* **some** *Persons*;
define class *Youngsters* : *person* **some** *Persons* **where** *age* < 30;

[3] As a naming convention for this paper, type identifiers start with a small letter and are in singular, whereas class names are capitalized and in plural.

[4] A more precise definition of the subclass relationship is given in Section 4.1.

Notice that the definition of *Persons* given earlier is equivalent to this one; *Objects* is the predefined root of the class hierarchy. *All* objects are contained in *Objects* (that is, for the **object**type, we *do* have an active domain). The class *Employees* is defined as a subclass of *Persons* with a more specific member type *employee*. The objects in *Employees* are some (possibly all) of the objects of class *Persons*: i.e., the subtype relationship between the member type of *Employees* and *Persons* is proper, and the extent of *Employees* is a subset of *extent(Persons)*.

The class *Youngsters* is also defined as a subclass of *Persons*. But here, the member types of both classes *Persons* and *Youngsters* are the same: *person*. The predicate given for class *Youngsters* is a constraint that all members of *Youngsters* have to be persons younger than 30 (that is, typically *Youngsters* will be a proper subset of *Persons*). The keyword **some** indicates that it is a necessary, but not sufficient, condition for members of *Persons* to become members of *Youngsters*. Changing the keyword **some** to **all** would indicate a necessary *and* sufficient condition: in this case the DBMS would automatically classify persons into the subclass, if the predicate evaluates to true.

In the remainder, we call classes specified with the keyword **all** (**some**) as all-classes (some-classes), respectively. Notice that, unlike e.g. [D$^+$90, Kim89, SÖ90, SRH90], we define the extent of a class to include the members of all its subclasses.

Views are derived classes, where the member type and the extent are defined implicitly by a query expression[SLT91]. Thus, the extent of a view is usually not stored explicitly, but rather computed from the view-defining query.[5]

Views provide a specialized interface to some base objects. A user or an application programmer usually works only with a small portion of the global schema, a *subschema*, that is particularly tailored for the task performed. Such a subschema consists of a collection of base and/or view classes together with the functions defined on them.

Our view mechanism simply allows arbitrary queries to serve as view definitions, exactly as in relational DBMSs. The crucial property of the query operations is their object-preserving semantics [SS90], such that the query does not result in copies of the input objects, but the object identities are preserved. As view definitions following query operations are provided: selections **select** $[P]$(*set-expr*) returning subsets of the input sets of objects, namely those satisfying the predicate P; a projection **project** $[f_1, ..., f_n]$(*set-expr*) yields the input set of objects with a usually new type, a supertype of the input type, which contains fewer functions; extension (**extend** $[f_1 := expr_1, ..., f_n := expr_n]$(*set-expr*)) that also contains the set of objects, but allows for the application of additional functions; and set operations. For example:

define view *EmplView* **as project** [*name,empl*] (*Employees*) ;
define view *YoungEmpls* **as** *Youngsters* **intersect** *Employees* ;

[5] The capability of materializing views is not considered in this paper.

The view *EmplView* represents the same objects as the class *Employees*, but only the two functions *name, empl* are applicable. The others, namely *age, address, salary*, and *ssecno* are hidden from the user. The second view illustrates that the set of objects can be restricted: *YoungEmpls* is defined as those objects that are in both classes, *Youngsters* and *Employees*, at the same time. Because the objects in the view belong to both classes, all functions that are defined on either of the respective member types can be applied.

Views can be used as arguments for queries and updates, just as ordinary classes can. We will see that, in contrast to the relational model, only a few restrictions have to be imposed. In fact, all update operators have the same effect as if they were applied to the base class, because the views' extents are derived from them.

Variables. In order to be able to refer to objects and results of previous algebra expressions, we allow the use of variables. Variables are used as *temporary names* ("handles") for instances of any type, i.e., data (integer, ...), objects, sets of any type, or functions. They have to be declared with their type in the database language, such that compile-time type checking applies to variables too. For example,

var *john* : *person* ;
var *BigCities* : **set of** *city* ;

declares variables of type *person*, and **set of** *city*, respectively. Hence, the object variable, for example, can be assigned the result of an object creation into class *Persons* in order to refer to the new object later; the set variable can keep the result of a selection on the database class *Cities*:

john := **new** (*person*) ;
BigCities := **select** [*population* > 1000000] (*Cities*) ;

In the example, a new object of type *person* is created, inserted into the class *Persons*, and assigned to the object variable *john*. Then, the query defines *BigCities* to have as value a subset of the persistent objects from the input class *Cities*.

3 Update Operations

In OODBMSs updates are usually carried out by type-specific methods that are defined by the type implementor. These methods provide a means to offer integrity-preserving updates to *clients*, but there is no support for the method *implementor* as far as integrity preservation is concerned. Therefore, COOL provides generic update operations that maintain model-inherent integrity constraints, and make the implementation of methods simpler. Additionally, applications might make direct use of these update operations. Therefore, object manipulations that involve the generic modeling facilities of COCOON, do not necessarily need to be coded into

methods over and over again. Rather, applications can use our generic operators directly where appropriate.

The update operators can be divided into four groups, the first three of which are defined according to the three modeling concepts (variables including functions, types, and classes): Assignments, operations for object evolution, and operations for manipulating the extents of classes and views. The last group are sequences of update operations that allow to keep integrity constraints consistent that are not covered by the modeling capabilities of the OODBMS: type-specific methods.

In the first place, all update operations are applied to single objects so as to obtain simple semantics. However, in order to apply set-oriented updates we provide a descriptive iterator (**apply**[*upd-seq*](*set-expr*)) that takes a sequence of update operations *upd-seq* as a parameter, and executes it for each element of a set *set-expr* (e.g., a query result) with deterministic semantics [LS93a]. Thus, we can not only apply single generic update operations in a set-oriented fashion, but also update sequences or type-specific methods.

3.1 Assignments

Assignments to Variables ($v := e$). The value of a variable v can be explicitly modified by an assignment. As usual for object-oriented languages, the inferred type of the right-hand side expression e can be a subtype of the variable's type. For example, the assignment

$p :=$ **pick** (**select** [$name =$ 'Smith'](*Persons*));

has the effect that (after the execution of the assignment) the variable p denotes an object with the name *'Smith'*.

Since functions are also regarded as variables, it is possible to assign a set of function pairs, which is produced by an appropriate expression, to functions. The effect would be that the function is redefined for all arguments at the same time. Typically, however, we want to redefine function values only for particular arguments. This is achieved by the following.

Partial Assignments to Functions (**set** [$f := e'$](e)).
The operation **set** changes the function f such that the function application $f(e)$ results in the value of the expression e'.

Continuing the example from above, we change the name of the person denoted by the variable p by a partial assignment to the *name* function from *'Smith'* to *'Jones'*:

set [$name :=$ 'Jones'](p).

Notice, however, that objects denoted by variables might change their function values, even though the variables are not used. This is due to object sharing, one of the most important properties of object-oriented models (see also object-oriented programming languages such as Eiffel [Mey88]). Consider the following example: Let the variable p denote a person, whose name is *'Jones'*; i.e., $name(p) =$ *'Jones'*. The execution of the two COOL statements

$q := p$;
set $[name :=$ 'Miller' $](q)$
yields a state, where p and q denote the same object, whose name is *'Miller'*. That is, the value of $name(p)$ changed from *'Jones'* to *'Miller'*, although the variable p was not changed explicitly. Let us now extend this example by considering the following assignment to the set-valued variable *jonesset*:

$jonesset :=$ **select** $[name =$ 'Jones' $](Persons)$

If we suppose that this operation was executed before the above operations, the variable *jonesset* contains all persons with the name *'Jones'*, especially the object denoted by the variables p and q. The "usual" semantics of such an assignment in databases is the snapshot semantics, that the query is evaluated once and the result is assigned to the variable. This distinguishes set-variables from database views, which are reevaluated each time the view is used. However, the above **set** operation propagates to the elements in the variable *jonesset* in the same way as to p: I.e., after the **set** operation the variable *jonesset* still denotes the same set of objects, but their names might be different from *'Jones'*. Thus, in object-oriented models the snapshot semantics of assignments interacts in a subtle way with object sharing.

3.2 Operations for Object Evolution

Object Creation (**new** (T)).
The creation of an object by **new** (T) instantiates type T and can be used, e.g., in an assignment to a variable whose type has to be T or a supertype thereof. Notice that object creation involves "invention" of new OIDs, that is, the object (OID) has to be different from all existing ones. [6]

In general, **new** (T) changes the database state such that the active domains of all supertypes of T are extended by the newly created object. Thus, by making the created object an instance of T and all its supertypes, the subset semantics of the subtype relationship is maintained.

For example, by the operation **new** (*employee*) we instantiate an object of type *employee*, which is also an instance of type *person* because of the subtype relationship between the two types.

Dynamically Acquiring More Types (**gain** $[T](e)$).
The **gain** operation is used to dynamically establish new instance-type relationships between an object denoted by e and a type T. It does not change any values of variables or functions, but it adds the object denoted by e to the active domains of all supertypes of T.

The motivation for dynamic type changes is the longevity of objects. In contrast to programs, where data are valid only until the end of program, information stored

[6] Since we do not show OIDs to users, we do not have to insist on not reusing OIDs. Therefore, we do not have to keep track of OIDs of deleted objects.

in databases is usually valid over a couple of years, such that the "role" of objects might change. For example the operation **gain** is needed in the well-known hire/fire scenario, when a person becomes an employee. In this case all functions that are applicable to employees should be applicable to the hired person, too. Therefore, the person denoted by the variable p has to become an instance of the type *employee* by the following operation:

gain $[employee](p)$.

A possible extension of the semantics could be to add the specification of default values for functions that now become applicable. Currently, none of the new functions gets a value, that is, they are all undefined ($\perp$) for the object e.

Dynamically Losing Types (**lose** $[T](e)$).

In contrast to the **gain** operation, **lose** deletes instance-type relationships. This is, the object denoted by e is removed from the active domains of T and all its subtypes. The effect of the operation is that all functions that are defined on the type T or a subtype of T are no longer applicable to the object denoted by e. Additionally, since COOL is a strongly-typed language and objects might be shared, the semantics of the **lose** operation has to be defined with respect to all typing constraints of variables and functions. As a consequence, we have to remove each occurrence of the object denoted by e from variables, sets and functions, if they are typed as T or a subtype of T.

In order to illustrate this requirement, let us discuss the following example. Assume that there are three variables, *empset* of type **set of** *employee*, *pset* of type **set of** *person*, and p of type *person*. Let the values of the variables be defined by the expressions:

$empset :=$ **select** $[salary > 100K](Employees)$;

$pset :=$ **project** $[name, age](empset)$;

$p :=$ **pick** (**select** $[salary > 100K$ **and** $name =$ 'Miller'$](Employees)$;

The variable p holds an object that is also element of both set variables. The difference between *empset* and *pset* is just on the type level, the object sets represented by them are the same. Miller's retirement by

lose $[employee](p)$

has the consequence that the object representing *'Miller'* is removed from the active domain of the type *employee*. Additionally, since functions that are defined on a subtype of *employee* must not be applied to the object that represents *'Miller'*, the object is also removed from the set denoted by *empset*, because the elements of this set have to be instances of type *employee*. Otherwise run-time errors would arise that are similar to using dangling references in programming languages. Therefore, the sets denoted by *pset* and *empset* become different, since *'Miller'* is still an element of *pset*, but not of *empset* anymore.

Therefore the deletion of instance relationships must be propagated to all constructed values, like sets, tuples, and functions, that are defined on the respective type. This propagation is defined by the mapping of values containing invalid in-

stances onto those that are valid after the update These proposed semantics guarantees that static type checking is sufficient despite operations that change types of objects dynamically. More intuitively, this kind of semantics implements the point of view that type information is part of the constraints that every valid database state has to fulfill. Once such constraints fail to hold, the state is changed by removing the objects from variable values. A formal definition of the semantics is given in [LS93b].

As already mentioned above, we need no explicit **destroy** operation, since its functionality is subsumed by the **lose** operation as follows:

$$\mathbf{destroy}\,(e) = \mathbf{lose}\,[\mathbf{object}](e)$$

This is, an object denoted by e is destroyed, if it loses its most general type *object*.

3.3 Manipulating the Extents of Classes and Views

The operation $\mathbf{add}\,[e](C)$ is provided to add an already existing object denoted by e into the extent of the class C. According to the subset-semantics of the subclass relation, the object also becomes a member of C's superclasses.

For example, by the operation

$$\mathbf{add}\,[p](\mathit{Employees})$$

the object denoted by p becomes a member of the class *Employees*. Due to the subclass relationship, the object is also added to the class *Persons*.

An object denoted by e can be removed from the extent of the class C by the operation $\mathbf{remove}\,[e](C)$. This operation has no effect on the existence of the object, but only on the membership relations between the object and classes. That is, the object is not only removed from the extent of the class C, but also from those of C's subclasses, since these have to be subsets of C's extent.

Continuing the above example, the object denoted by p can be removed from the class *Employees* by the following operation:

$$\mathbf{remove}\,[p](\mathit{Employees})$$

If there are subclasses of *Employees*, p would also be removed from their extents. However, the object remains in the extent of the class *Persons*.

The detailed semantics of the operations **add** and **remove** with respect to class predicates and views are defined in Section 4.2.

4 Maintenance of Integrity Constraints

The model-inherent integrity constraints are separated into the following two groups:

- **Constraints** like typing constraints and the subtype relationships;
- **Derivation Rules** such as class predicates, view definitions, subclass relationships and inverse functions.

Constraints are preserved due to the definition of the update operations such that no run-time type errors (like dangling references or invalid occurrences of objects in constructed values) occur. A definition of the update operations using the approach of denotational semantics can be found in [LS93b]. Derivation rules are formalized by the mapping of classes, views and inverse functions with their respective operations onto the basic model.

4.1 Modeling Classes and Views

Even though "class" is a central concept in a COCOON database schema, it is a derived concept in terms of our formalization, that can be defined using objects and functions. There is only a slight extension of the formal level for static type-checking.

4.1.1 Introducing meta classes

We introduce a new object type *class* whose instances represent classes and that includes the following functions representing the class properties. Notice that we use COOL statements that are already mapped onto the formal level:

$$\begin{array}{lll} \textbf{define type } class \textbf{ isa object} = & cname & : \textbf{string}, \\ & bases & : \textbf{set of } class, \\ & suffp & : \textbf{object} \rightarrow \textbf{bool}, \\ & pmemb & : \textbf{set of object}; \end{array}$$

As top element of the class hierarchy we require the existence of class *Objects*, which represents all existing objects (i.e., the active domain of type **object**). Additionally, we provide the meta class *Classes*, whose extent represents all classes (including itself).
The following COOL statements initialize the class hierarchy:

$$\begin{array}{ll} \text{[Init]} & \textbf{define var } Classes, Objects : class; \\ & Classes := \textbf{new}\,(class); Objects := \textbf{new}\,(class); \\ & \textbf{set}\,[cname := \text{'Classes'}](Classes); \\ & \textbf{set}\,[cname := \text{'Objects'}](Objects); \\ & \textbf{set}\,[bases := \text{'Classes'}](Classes); \\ & \textbf{set}\,[bases := \text{'Objects'}](Objects); \\ & \textbf{set}\,[pmemb := \{Classes, Objects\}](Classes); \\ & \textbf{set}\,[pmemb := pmemb(Classes)](Objects); \\ & \textbf{apply}\,[\textbf{set}\,[suffp := x.x \in pmemb(c)](c)](c : Classes) \end{array}$$

Notice that the function *mtype* is not contained as a class property because we are interested in a language that allows for static type checking. However, if we have to apply a function, which may vary over time, in order to get the member type of a class, static type checking is not possible. Therefore the function *mtype* belongs to the typing environment akin to the declarations of variables and functions [SLR+93].

4.1.2 Representation of class predicates

As already mentioned the extent of classes can be constrained by necessary or necessary and sufficient predicates. In order to deal with some-classes, all-classes, and views uniformly, we "complete" the class predicate of some-classes to become necessary and sufficient. Then, maintaining consistency w.r.t. class membership during updates can easily be achieved by the fully determining predicate: the extent of a class (or view) is derived as all instances of the member type that fulfill the necessary and sufficient class predicate *suffp*. Therefore, the extent of a class C is defined by the following COOL query[7]:

$$extent(C) = \textbf{select}\,[suffp(C)](\textbf{adom}\,(mtype(C)))$$

4.1.3 Definition of class objects

In order to represent COCOON classes by objects, each definition of a class C results in sequences of update operations on the schema level. The following statements are common to all kinds of class definitions:

[Common] **define var** $C : class$;
$C := \textbf{new}\,(class)$;
set $[cname :=$ 'C'$](C)$;
set $[pmemb := pmemb(Classes) \cup \{C\}](Classes)$;

An object representing the class is generated, gets its classname assigned, and becomes a member of the (meta-)class $Classes$.

In order to obtain necessary *and sufficient* predicates for some-classes, we have to internally keep track of objects that have explicitly been added by the user (and not removed since). We collect, for each some-class, those objects in a set that have been added in the function *pmemb*. If applied to a some-class C, it returns the set of "potential members" of C (which is initialized by the empty set). Thus, the sufficient predicate for a some-class is the conjunction of the necessary condition (if any) and the test whether an object is included in $pmemb(C)$. The set of base classes, *bases*, which is needed for the view update mechanism, is the singleton set including the class itself. Thus, the **some** class definition

define class C $: T$ **some** $C_1, ..., C_n$**where** p

consists of the above [Common] statements plus the following ones:

[Some] **set** $[pmemb := \emptyset](C)$;
set $[bases := \{C\}](C)$;
set $[suffp := x.\ x \in pmemb(C)$ **and** $p(x) \bigwedge_{i=1}^{n} suffp(c_i)](C)$;

[7] The operation **adom** returns the active domain of an object type that can be derived by the selection on the class *Objects* where the instance relationship is checked.

In case of **all** classes, the set of base classes is the union of the base classes of the superclasses, and the sufficient predicate is defined by the predicate p and the class membership in all superclasses. Therefore, the **all** class definition

define class C $: T$ **all** $C_1, ..., C_n$**where** p

is mapped onto the [Common] statements plus the following operations:

$$\begin{aligned}[\text{All}] \qquad & \textbf{set}\,[\mathit{bases} := \bigcup_{i=1}^{n} \mathit{bases}(C_i)](C); \\ & \textbf{set}\,[\mathit{suffp} := x.\ p(x) \bigwedge_{i=1}^{n} \mathit{suffp}(C_i)](C);\end{aligned}$$

In case of a view definition

define view C **as** e

the predicate *suffp* is the membership in the defining expression:

$$[\text{View}] \qquad \textbf{set}\,[\mathit{suffp} := x.\ x\ \in\ e](C);$$

The *bases* of the view can be derived by the following recursive predicate:

$$\mathit{bases}(e) := \begin{cases} e & \text{if } e \text{ is a some-class,} \\ \mathit{bases}(e') & \text{if } e = \textbf{project}\,[\ldots](e'), \\ & \text{or } e = \textbf{extend}\,[\ldots](e'), \\ & \text{or } e = \textbf{select}\,[\ldots](e'), \\ \mathit{bases}(e')\ \cup\ \mathit{bases}(e'') & \text{if } e = e'\ \textbf{union}\ e'', \\ & \text{or } e = e'\ \textbf{intersect}\ e''. \end{cases}$$

4.1.4 Class hierarchy

The partial reflexive order between classes ($\sqsubseteq$) is defined by the conjunction of the subtype relationship between the member types and the predicate inclusion between the necessary and sufficient predicates:

$$C_1\ \sqsubseteq\ C_2\ \stackrel{\text{def}}{=}\ \mathit{mtype}(C_1)\ \preceq\ \mathit{mtype}(C_2)\ \wedge\ \mathit{suffp}(C_1)\ \Longrightarrow\ \mathit{suffp}(C_2)$$

4.2 Mapping the Update Operations add and remove

Since classes are modeled by using objects and functions, we can specify the semantics of the operations **add** and **remove** in terms of the elementary operations. That is, they are not elementary operations of our language, but derived as follows :

$$\begin{aligned}\textbf{add}[e](C) \quad &\equiv \textbf{apply}\,[\,\textbf{set}[\mathit{pmemb} := \mathit{pmemb}\ \ \textbf{union}\ \{e\}](y)] \\ &\qquad (y : \textbf{select}[\emptyset \neq \textbf{select}\,[x \sqsubseteq c](x : \mathit{bases}(C)) \\ &\qquad\qquad \textbf{and}\ \mathit{bases}(c) = \{c\}](c : \mathit{Classes})) \\ \textbf{remove}[e](C) &\equiv \textbf{apply}\,[\,\textbf{set}[\mathit{pmemb} := \textbf{select}\,[x \neq e](x : \mathit{pmemb})](c)] \\ &\qquad (c : \mathit{bases}(C))\end{aligned}$$

The **add** operation is defined by applying the **set** operation that adds the object e to the set $pmemb(c)$, for each class c contained in the result of the **select** operation. The predicate of the selection chooses all some-classes of the database (identified by the condition $bases(c) = \{c\}$) that are superclasses of a class included in $bases(C)$. Thus, adding objects to a class is propagated to all its superclasses. Removing objects from a class can be specified easier, because the propagation to the subclasses is carried out by the necessary and sufficient predicates. Therefore, the semantics of the **remove** operation is to take the object out of the *pmemb*-sets of the classes contained in $bases(class)$. In particular, this definition allows for the application of **add** and **remove** not only on some-classes but also on all-classes and views. This is motivated by considering applications, which work on subschemata that do not contain the complete class hierarchy (see e.g. [AB91, TYI88]). For a detailed discussion of the motivation and alternatives for the semantics of **add** and **remove** see [LS92].

4.3 Inverse Functions

In general, we conceive functions as binary relations (similar to IRIS [WLH90]). In case of single-valued functions, the first component of the relation denotes an argument of the function and the second component is the result of the function application. Set-valued functions can be mapped to binary relations by inserting a pair $\langle o, o' \rangle$ in the relation, if o' is an element of the set that results from the function application to o. The inverse of a function is interpreted as the same relation, with the components exchanged. Since, in general, a function may yield the same value for different arguments, the inverse function has to be defined as set-valued. Notice, however, that the key word **unique** could be used for single-valued and set-valued functions to indicate that the inverse function is single-valued.

4.3.1 Updating Inverse Functions

Because some updates are more easily expressed on one function, and others on the inverse, it is convenient to allow updates on either of two inverse functions. Of course, if one of them is updated, the changes have to be propagated to the other automatically.

For example, let us consider the inverse functions *works_for* that is applicable to employees and *staff* that is defined on companies:

$$\begin{aligned} works\_for &:: employee \rightarrow company \ \textbf{inverse} \ staff \\ staff &:: company \rightarrow \{employee\} \ \textbf{inverse} \ works\_for \end{aligned}$$

If a new company c is assigned to an employee e by the operation

$$\textbf{set}\,[works\_for := c](e)$$

the inverse function must be changed by adding e to *staff*(c). However, if the staff of a company gets exchanged, it is rather cumbersome to use assignments of the *works_for* function (to nullify the old staff and set the new ones). Instead, it should be possible to exchange c's staff by

$$\textbf{set}\,[\mathit{staff} := \{e_1, \ldots, e_n\}](c)$$

such that the *works_for* function returns c only for the employees e_1 through e_n. Thus, we provide a flexible update capability for inverse functions that is independent of any storage strategy, since typically only one function will be stored, the other one will be derived. Similar to the view update mechanism, we provide a default semantics for elementary update operations. In case that a more sophisticated semantics is needed in an application, one has to define appropriate methods using the elementary update operations.

The propagation of updates to inverse functions depends on whether the functions are single- or set-valued. Therefore, we have to consider the four cases for the three kinds of operations that can change a function: (i) partial assignments, (ii) global assignments, and (iii) **lose** operations that might remove instances from either domain.
Assume the following function definitions:

$$\textbf{function}\ f : T^f_{dom} \rightarrow T^f_{rng}\ \textbf{inverse}\ g$$
$$\textbf{function}\ g : T^g_{dom} \rightarrow T^g_{rng}\ \textbf{inverse}\ f$$

where the four cases result from different function ranges:

I	$T^f_{rng} \preceq T^g_{dom}$	$T^g_{rng} \preceq \{T^f_{dom}\}$
II	$T^f_{rng} \preceq T^g_{dom}$	$T^g_{rng} \preceq T^f_{dom}$
III	$T^f_{rng} \preceq \{T^g_{dom}\}$	$T^g_{rng} \preceq \{T^f_{dom}\}$
IV	$T^f_{rng} \preceq \{T^g_{dom}\}$	$T^g_{rng} \preceq T^f_{dom}$

Let us now discuss how the application of the three kinds of operations to the function f propagates to the inverse g [8].

(i) The effect of the operation $\textbf{set}\,[f := e'](e)$ on the binary relation is: first, delete all pairs that have e as their first component; then, in the cases I and II, insert the new pair $\langle e, e'\rangle$; otherwise (case III and IV) insert a pair $\langle e, e''\rangle$ for each $e'' \in e'$. In the cases II and IV the old tuple $\langle \hat{e}, e'\rangle$ is removed from the relation, such that g is a single-valued function. The effect to $\hat{e}$ is that $f(\hat{e})$ is set to undefined ($\perp$) in case II, and e' is removed from $f(\hat{e})$ in case IV.

Therefore the effect on g is as follows: (I) e is added to the set $g(e')$; (II) the value $f(g(e')$ is undefined for the old value $g(e')$, the new value of $g(e')$ is e; (III) $g(e'')$ contains e if and only if $e'' \in e'$; (IV) $g(e'')$ is e if and only if $e'' \in e'$, $g\hat{e})$ is undefined if it was e and $\hat{e} \notin e'$.

[8] Because the update capability is symmetric, we do not have to discuss the application of the operations to g.

(ii) The operation $f := x.e$ can be mapped to the **set** $[f := e'']$$(e')$ where e'' results from substituting x in e by e', which iterates over the instances of T^f_{dom}.

(iii) By the operation **lose** $[T^f_{dom}](e)$ the function f is not applicable to e anymore, because e is removed from the domain T^f_{dom}. Additionally, e is removed from function values $g(e')$ according to the derivation rule nv, because the range of g is either a subtype of T^f_{dom} or a subtype of $\{T^f_{dom}\}$. Therefore, the inverse constraint is fulfilled anyway.

5 Related Work

In general, most models allow to specify total functions and uniqueness constraints. Knowledge representation systems such as BACK [PSKQ89] and O^{2}FDL [MCB90] allow to specify additional integrity constraints such as number restrictions. However, these models lack powerful update mechanisms. The update semantics is usually restricted such that the whole transaction is aborted, if the last update operations yields an inconsistent database state. This approach might be appropriate in cases where application-specific constraints are concerned. However, in case of model-inherent constraints we propose more powerful update operations.

In contrast to object-oriented programming languages (like C++, Eiffel, and Smalltalk) our proposed operations capture the evolution of objects. That is, the set of associated attributes or methods might change over the life cycle of an object. These changes are propagated to all occurrences of an object. That is, the semantics is not just to change the pointer the operation is applied to. Thus, destroying objects deconstructs the object corresponding to the **delete** operation in C++ *and* changes the variables and functions pointing to this object like the **forget** operation of Eiffel. According to the distinction between persistent and transient objects we separate **lose** $[$**object**$](obj)$ that destroys all occurrences of an object from **remove** $[obj](Objects)$.

Similar operations for object evolution are also proposed in [RS91] and [FAB$^+$89]. However, in contrast to their informal presentations our operations are formally defined [LS93b] and the impact of dynamic type changes on the type system is analyzed.

The proposed update operations in the Galileo model [Ghe90] are similar to ours, because they also separate types from classes. Classes represent sets of (typed) objects and the subclass relationship is the set inclusion that is maintained if the class extent is changed. Since they provide neither views nor any class predicates, their classes correspond to some-classes without predicates in our model. They propose an operation *specialize* for the migration of objects. However, this operation is restricted by the constraint that the specialized type has to be a subtype of the current type of the object. Thus, that operation is more restricted than our **gain** operation, because of a more restricted type system. Additionally, the operation *specialize* cannot be type checked at compile–time, because the success depends on the current type of

an object. They don't provide an operation to restrict the type according to our **lose** operation.

In [BSKW91] it is shown that the view update problem can be alleviated, if the system knows about integrity constraints (that is, referential integrity in case of the relational model). Our approach allows updates through views, which represent a special kind of derived data (namely derived classes), because the implied integrity constraints are represented in the model. This is one advantage of our view mechanism in contrast to the one in O_2 [dSAD94], where only problems of updating imaginary classes are sketched, but no solution is provided. The general problem how of updating derived data (see [Abi88] for a survey on this problem in the context of deductive databases) is not captured here.

6 Conclusion

In this paper we presented a set of generic update operations for an object-oriented data model that includes class predicates (known from knowledge representation models) as well as variables (borrowed from programming languages). The semantics of these operations is defined such that the model-inherent integrity constraints such as typing, class membership, subtype- and subclass-relationship, and class predicates can be maintained automatically. Predicates on classes are used as membership criteria rather than constraints. That is, if changes to a member object makes the class predicate false for that object, it is removed from the class (instead of rejecting the update). Furthermore, the strict type system is also exploited to define consistent update semantics, particularly w.r.t. variables. The main contribution is that (i) the capabilities to express semantics in terms of the model must fit in well with the provided update operations and (ii) maintaining model-inherent integrity constraints alleviates the definition of methods, which are needed to express the application specific update operations.

Besides the efficient implementation of the proposed operations, future work will focus on the integration of general constraints in order to support application specific integrity constraints. That is, we are going to make use of constraint definition languages, which have been proposed in context of deductive databases, as means to express integrity constraints that are not already covered by the model in terms of class invariants and method-specific pre- and post-conditions.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In J. Clifford and R. King, editors, *Proc. ACM SIGMOD Conf. on Management of Data*, pages 238–247, Denver, Co, May 1991. ACM, New York.

[Abi88] S. Abiteboul. Updates, a new frontier. In M. Gyssens, J. Paredaens, and D. van Gucht, editors, *ICDT '88: 2nd Int. Conf. on Database The-*

ory, pages 1–18, Bruges, Belgium, September 1988. LNCS 326, Springer Verlag, Heidelberg.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[Bee89] C. Beeri. Formal models for object-oriented databases. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, pages 370–395, Kyoto, December 1989. North-Holland. Revised version appeared in "Data & Knowledge Engineering", Vol. 5, North-Holland.

[BSKW91] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 248–257, Denver, CO, May 1991. ACM, New York.

[D+90] O. Deux et al. The story of O_2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, March 1990. Special Issue on Prototype Systems.

[Day89] U. Dayal. Queries and views in an object-oriented data model. In R. Hull, R. Morrison, and D. Stemple, editors, *2nd Int'l Workshop on Database Programming Languages*, pages 80–102, Oregon Coast, June 1989. Morgan Kaufmann, San Mateo, Ca.

[dSAD94] C.S. dos Santos, S. Abiteboul, and C. Delobel. Virtual schema and bases. In *Advances in Database Technology - EDBT'94*, pages 81–94, Cambridge, UK, March 1994. Springer LNCS 779.

[FAB+89] D.H. Fishman, J. Annevelink, D. Beech, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, and W.K. Wilkinson. Overview of the iris dbms. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 10, pages 371–394. ACM Press, Addison-Wesley, New York, 1989.

[Ghe90] G. Ghelli. A class abstraction for a hierarchical type system. In S. Abiteboul and P.C. Kanellakis, editors, *Proc. Int. Conf. on Database Theory (ICDT)*, pages 56–70, Paris, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[Kim89] W. Kim. A model of queries for object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 423–432, Amsterdam, August 1989.

[LS92] C. Laasch and M.H. Scholl. Generic update operations keeping object-oriented databases consistent. In *Proc. of 2. GI Workshop Information Systems and Artificial Intelligence*, pages 40–55, Ulm, Germany, February 1992. IFB 303, Springer Verlag, Heidelberg.

[LS93a] C. Laasch and M.H. Scholl. Deterministic semantics of set-oriented update sequences. In *Proc. of the IEEE Conf. on Data Engineering*, pages 4–13, Vienna, Austria, April 1993.

[LS93b] C. Laasch and M.H. Scholl. A functional object database language. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Database Programming Languages (DBPL4)*, pages 136–156, New York City, August 1993. Springer, London.

[MCB90] M.V. Mannino, I.J. Choi, and D.S. Batory. The object-oriented functional data language. *IEEE Transactions on Software Engineering*, 16(11):1258–1272, November 1990.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, 1988.

[PSKQ89] C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK system revisited. Technical Report KIT-Report 75, Technical University of Berlin, Berlin, Germany, September 1989.

[RS91] J. Richardson and P. Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 298–307, Denver, CO, May 1991.

[RS94] E. Radeke and M.H. Scholl. Object migration in federated database systems. In *Proc. 3rd Int'l Conf. on Parallel and Distributed Databases*, Austin, TX, September 1994. Full version available as Technical Report.

[SLR+93] M.H. Scholl, C. Laasch, C. Rich, M. Tresch, and H.-J. Schek. The COCOON object model. Technical Report 192, ETH Zürich, Dept. of Computer Science, December 1992. Also available as Technical Report 93-02, University of Ulm, Dept. of Computer Science, February 1993.

[SLT91] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 189–207, Munich, Germany, December 1991. LNCS 566, Springer Verlag, Heidelberg.

[SÖ90] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented databases. *ACM Transactions on Office Information Systems*, 8(4):387–430, October 1990.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):125–142, March 1990. Special Issue on Prototype Systems.

[SS90] M.H. Scholl and H.-J. Schek. A relational object model. In S. Abiteboul and P.C. Kanellakis, editors, *ICDT '90 – Proc. Int'l. Conf. on Database Theory*, pages 89–105, Paris, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[SST92] M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In *Proc. Int'l Workshop on Distributed Object Management*, Edmonton, Canada, August 1992. Morgan Kaufmann.

[ST94] M.H. Scholl and M. Tresch. Evolution towards, in, and beyond object databases. In K. von Luck and H. Marburger, editors, *Management and Processing of Complex Data Structures, Proc. 3rd Workshop on Information Systems and Artificial Intelligence*, Hamburg, Germany, February 1994. Springer, LNCS 777.

[TS92] 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.

[TS93] 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)*, Taejon, Korea, April 1993.

[TYI88] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema virtualization in object-oriented databases. In *Proc. IEEE Data Engineering*, pages 23–30, Los Angeles, February 1988.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):63–75, March 1990. Special Issue on Prototype Systems.