In: *R. Studer, editor, Proc. 2nd GI-Workshop on Information Systems and Artificial Intelligence (IS/KI), Ulm, Germany, February 1992, IFB 303, Springer Verlag, Heidelberg.*

Generic Update Operations Keeping Object-Oriented Databases Consistent

Christian Laasch, Marc H. Scholl
Department of Computer Science, Information Systems - Databases
ETH Zentrum, CH-8092 Zürich, Switzerland
e-mail: {laasch, scholl}@inf.ethz.ch

Abstract

One of the objectives of ooDBMSs is to use type-specific methods for manipulating objects, in order to maintain the consistency of the database. This is, however, little help for the method implementor as far as the model-inherent constraints are concerned. We propose a set of *generic update operations* that maintain integrity constraints such as types, class memberships, subtype-, subclass-relationships, and class predicates. The 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.

Keywords: Object-Oriented Databases, Updates, Views, Integrity Constraints, Object Model

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). They typically ignored the maintenance of integrity constraints, i.e., referential integrity and uniqueness of a primary key. Since object-oriented data models offer a much broader scope of built-in semantics (such as the subclass relationship), 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 guarantee consistency during modifications.

We present an approach toward defining such a collection of generic update operators (in the context of our object model COCOON [SS90a]). Among the model-inherent integrity constraints are the typing, class membership, subtype- and subclass-relationships, class predicates, variables, and views.

The key idea that leads to update operations keeping the database consistent is to separate the sphere of constraints into types and classes where classes internally represent necessary and sufficient predicates. The effect of update operations is then captured either by 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 (like KL-ONE [BS85], BACK [NP90, PSKQ89]) and a strong type system form object-oriented programming languages (e.g. [Mey88, AB87]).

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 object preserving semantics of the query language operators. Section 3 presents the generic update operations and gives their semantics in terms of state transformations. Section 4 shows the representation of classes by predicates, such that automatic classification of objects guarantees subclass and class membership consistency. In Section 5 we discuss update semantics for subschemas. A comparison to related work is presented in Section 6 before we conclude in Section 7.

2 The COCOON Model

The COCOON model as described in [SS90a, SS90b] consists of objects and functions (see also [WLH90, Day89]), but separates types from classes: Types include all compile-time information, whereas classes represent collections that vary over time. COCOON is a core object model, meaning that we focus on the essential ingredients necessary to define a set-oriented query and update language. Therefore, for instance, tuples as a type constructor are excluded from the core.

2.1 Basic Concepts

Objects are instances of abstract object types (AOTs). They can be manipulated only by means of their interface, a set of functions.

Data are instances of concrete types (such as numbers, strings) and constructed types (such as sets). The distinction from objects is equal to [Bee89].

Functions are described by a name and signature, they are the interface operations of type instances. The implementation is specified separately. We use the term *functions* in the general sense including *retrieval functions* as well as *methods*, that is, functions with side-effects. According to generic update operations we consider only stored retrieval functions, which are uniform abstractions of "attributes" and "relationships" of classical data models, since directly updating derived or computed functions requires type-specific methods. Indirect updates (i.e., updates to values used in the derivation) are automatically propagated upon evaluation. A capability to express more semantics in the model is the feature of defining inverses of functions that are enforced by the system during updates.

Types describe the common interface of all instances of that type. That is the set of applicable functions in case of abstract object types. So, the definition of a type normally consists of two parts: a set of functions and a type name.[1] The following example defines a type *PersonT* with three functions *name, age,* and *children*:

$$\begin{aligned}\textbf{type } \mathit{PersonT} \textbf{ isa } \mathit{ObjectT} = \quad & \mathit{name} : \textbf{string},\\ & \mathit{age} : \textbf{integer},\\ & \mathit{children} : \textbf{set of } \mathit{PersonT};\end{aligned}$$

The function *children* illustrates the use of **set** as a type constructor. In general, types can either be atomic (data types and abstract object types) or constructed. We allow the application of two type constructors: set and function (e.g., children is an instance of the constructed type "functions from *PersonT* to **set of** *PersonT*").

As we will see later, queries can dynamically produce new types. Those are unnamed, but their set of functions can be derived from the query by standard type inference.

Subtyping. If a type is defined as a *subtype* of another then every instance of the subtype is also an instance of its supertype. This is called *multiple instantiation*. The definition of the subtype relationship ($\preceq$) can be divided into two parts: First subtyping between atomic types and secondly between constructed ones. The subtype relationship between constructed types can be derived as follows (see also [MCB90]):

$$\textbf{set of } T_1 \preceq \textbf{set of } T_2 \Longleftrightarrow T_1 \preceq T_2 \text{ for sets, and}$$

$$T_1^{dom} \rightarrow T_1^{rng} \preceq T_2^{dom} \rightarrow T_2^{rng} \Longleftrightarrow T_1^{rng} \preceq T_2^{rng} \wedge T_1^{dom} \succeq T_2^{dom} \text{ for functions.}$$

[1] In this paper, we write a ...T at the end of an identifier to make clear that it is a type, and a ...C for classes.

The subtyping relation between abstract object types is defined by the inclusion of the applicable function sets. For example, assuming the following additional type definition

type *EmployeeT* **isa** *PersonT* = *salary* : **integer**,
courses : **set of string**;

the *EmployeeT* type is a subtype of the type *PersonT*, since the applicable functions of *PersonT* ({*name, age, children*}) are included in the function set of *EmployeeT* (which is {*name, age, children, salary, courses*}). Subtyping defines a partial order $\preceq$ on abstract object types, forming a 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 *ObjectT* where no user-defined function is applicable (therefore, all instances of defined types in the database are also instances of *ObjectT*), 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 and Views are strictly distinguished from types in the following sense (see also [ACO85, Bee89]): Types are interface specifications (a collection of functions), whereas classes are containers for objects of some type (type extents). Each class or view, C, represents a (typed) set of objects and associates a type, the *member_type*(C), to all objects in the set *extent*(C). The extent of a class includes all objects that are instances of the member type and fulfill class-specific properties. For example:

class *PersonC* : *PersonT* **some** *ObjectC*;
class *YoungC* : *PersonT* **some** *PersonC* **where** *age* < 30;

The member types of both classes is the type *PersonT*. The predicate given for class *YoungC* is a constraint that all members of *YoungC* have to be persons younger than 30. The keyword **some** indicates that this is a necessary, but not sufficient, condition for members of *PersonC* to become members of *YoungC*. 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 denote classes specified with the keyword **all** / **some** as all-classes / some-classes, respectively. Notice that, unlike e.g. [D+90, Kim89, SÖ90], we define the extent of a class to include the members of all its subclasses. Views that are defined by queries of arbitrary complexity can be regarded as a special kind of classes, because their extent is sufficiently specified and their type can be derived[2].

Classes represent polymorphic sets: Members may be instances of several other (sub-)types particular subtypes in addition to the member type. Type checking of our language always refers to the unique member type, though. Due to the separation of types and classes, there may be any number of classes for a particular type.

Subclassing. There are several choices as to 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, because they are often correlated, but they need not. We will speak of a subclass relationship $C_1 \sqsubseteq C_2$, iff for the two classes it is true that *member_type*(C_1) $\preceq$ *member_type*(C_2) **and** *extent*(C_1) $\subseteq$ *extent*(C_2). Usually, at least one of the ordering relationships will be proper. Continuing the example:

class *PersonC* : *PersonT* **some** *ObjectC*;
class *YoungC* : *PersonT* **some** *PersonC* **where** *age* < 30;
class *EmployeeC* : *EmployeeT* **some** *PersonC*;

The class *YoungC* is a subclass of *PersonC* with the same type, but a subset of the objects, whereas *EmployeeC* is a subclass associated with a subtype of *PersonT* (and probably—but not necessarily—also a subset).

Variables. In order to be able to refer to objects and results of previous algebra expressions, we allow the use of variables instead of making the object identity explicit. Variables are used as temporary names ("handles") for instances of any type, i.e., data (e.g. integer), objects, sets of any type or functions. They have to be declared with their type in the database sublanguage—either explicitly or by a query—, such that compile-time type checking applies to variables too. For example,

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

var $Kids$: **set of** $PersonT$,
var My_Child : $PersonT$,
var $Likes\_Cats$: $PersonT \rightarrow$ **bool**

declares variables of type **set of** $PersonT$, $PersonT$, and $PersonT \rightarrow$ **bool**, respectively. Hence, the set variable, for instance, can keep the result of a selection on the database class $PersonC$; the object variable, for example, can be assigned the result of an object creation of type $PersonT$ in order to refer to the new object later (the operators are shown in the next sections):

$Kids :=$ **select** $[P](PersonC)$;
create $[PersonT](My\_Child)$;
set $[name :=$ 'Susie'$](My\_Child)$;

In the example, the query defines $Kids$ to have as value a subset of the persistent objects from the input class. Then a new object of type $PersonT$ is created and assigned to the object variable My_Child. After that, the name function for the new object is set to the value 'Susie'.

Variables are crucial w.r.t. consistent update semantics, since they introduce the notion of assignments (which in turn is typically associated with a copy semantics). Essentially, variables introduce a second object space in addition to the (persistent) database: the transient objects in an application program. The idea of a snapshot (taken by an assignment to a variable) is not easily combined with object identity (i.e., sharing): if the value of a variable contains a shared object, updates to that object *have to* be reflected in the variable.

2.2 Generic Query Operations

We use a set-oriented algebra, where the inputs and outputs of the operations are sets of objects. Hence, query operators can be applied to extents of classes, set-valued function results, query results, or set variables. The effects of each operator are described separately for type and extent. (Only **union**, **intersect**, and **pick** have an effect on both.)

Selection (**select** $[P]$(*set-expr*)) returns a subset of the input set of objects, namely those satisfying the predicate P. The type of the set is unchanged, it is *type*(*set-expr*).

Projection (**project** $[f_1, ..., f_n]$(*set-expr*)). The output of a projection is a set with a usually new type, a supertype of the input type: fewer functions are defined, namely only those listed in the projection. All objects of the input set are also elements of the output set (*object preservation*).

Extend (**extend** $[f_1 := expr_1, ..., f_n := expr_n]$(*set-expr*)). Projection eliminates functions, extend defines new derived ones. Obviously, each function name f_i must be different from all existing functions for the type of the input. The expression $expr_i$ can be any legal arithmetic-, boolean-, or set-expression. The result set contains exactly the same objects as the input, but a new type, a subtype of the input type, is associated to it (all the old functions plus the new ones are defined on it).

Pick (**pick** (*set-expr*)) is provided to convert a singleton set into the only element (i.e., drop the spurious set braces). The result type is the element type of the set.

Set operations. As the extent of classes are sets of objects, we can perform set operations as usual. With a polymorphic type system, we need no restrictions on operand types of set operations (ultimately, they are all objects). The result type, however, depends on the input types: for the union it is the lowest common supertype (in the lattice) of the input types. The intersection results in the greatest common subtype; finally, difference operation yields a subset of its first argument with the same type.

These are the basic *object preserving* query operators of our algebra. Other operators, such as join can be derived from them. The complete algebra, including operators for generating sets of tuples as query results to communicate with value-oriented environments, is described in [SS90b, SS90a].

3 Generic Update Operations

This section gives the semantics of our generic update operators. We will first sketch the systematics of the operators we offer. Then we will describe how the *state* of a database is formally defined. Finally, we show how each of the generic update operators affects this state.

3.1 Systematics of Update Operators

All elementary update operations are applied to single objects instead of sets. However, in order to apply set-oriented updates we provide a descriptive iterator (**apply_to_all**[*upd-op*](*set-expr*)) that takes one update operation as a parameter that is executed for each element of a set (e.g., a query result).

Our generic update operators can be divided into three groups, according to the three modeling concepts: variables (including functions), types, and classes.

- Assignments (:= and **set**) for changing values of variables and functions (Section 3.3).
- Operations for object evolution: besides being created and deleted, objects might also **gain** or **lose** types (Section 3.4).
- Operations (**add** and **remove**) for manipulating the extents of classes (Section 4.4).

In contrast to the first two groups, the third one contains no elementary operations. This is, because classes are formalized by using functions that are applicable to instances of an abstract object type representing classes. Therefore the operations for manipulating the extent of classes can be expressed in terms of elementary operations.

3.2 Formalization of the Database State

Update operations are transformations from one database *state* to another. This section describes how we formalize the notion of a database state.

Common to all different possibilities of formalizing the state of a database, is the fact that all information concerning the current state of the data level is represented. This is in our model, information about instance relationships, and the values of functions and variables. The class membership is excluded from the state, because classes can be modeled by types and functions (see Chapter 4). According to our object-function approach, we do not model an internal state of an object directly. Rather, all information on objects is contained in the instance relationships and the function and variable mappings.

Formally, function names are considered as variables over function types. Since also the active domain of abstract object types, which defines is the current set of instances, can be regarded as variables, we can represent variables, functions, and abstract object types all in the same way: by variables (or external names) to which the database state associates a current value.

That is, the database state is defined by a function σ that maps variables to values. The type of variables can be described by a function A yielding a type expression for each variable.

For example let the variable *person* of type $PersonT$ (i.e., $A(person) = PersonT$) denote an object p_3 (i.e., $\sigma(person) = p_3$). Besides this variable there might be a function *name* with $A(name) = PersonT \rightarrow$ **string**. The value of the variable (resp. function) *name* can be described by a set of pairs that consists of an object of type $PersonT$ as the first component and a string as the second. For example, the current value of the function *name* might be the following set:

$$\{\langle p_1, \text{'Smith'}\rangle, \langle p_3, \text{'Miller'}\rangle\}$$

Note that this set is one instance of the type $PersonT \rightarrow$ **string**, the set of all possible functions mapping persons to strings.

The active domain of any abstract object type is obtained as the value of function σ applied to the type. For example, in our current state $\sigma(PersonT) \supseteq \{p_1, p_3\}$. We do not explicitly maintain active domains of constructed types, because these can be derived from the active domains of the component types. The state function σ itself can also be described as a set of pairs. For example, our current state includes the following:

$$\sigma = \{\langle person, p_3\rangle, \langle PersonT, \{p_1, p_3, \ldots\}\rangle,\ \langle name, \{\langle p_1, \text{'Smith'}\rangle, \langle p_3, \text{'Miller'}\rangle, \ldots\}\rangle, \ldots\}$$

Since our model includes variables as part of the interface between the database and an application program, the state describes not only persistent information (the database), but also information that might be transient (like variables of a program). In order to differentiate between persistent and transient objects, we state that all members of the predefined class $ObjectC$ are persistent. Objects that are not member of this class are transient ones. Therefore persistent classes (i.e., classes representing persistent objects) are

subclasses of the class $ObjectC$. Due to this definition of persistence, we do not need additional operations for making objects persistent or transient, since we can make use of the operations **add** and **remove** that add or remove an object to or from the extent of a class, respectively. Thus, the semantics of update operations can be defined independently of persistence.

The semantics of update operations can now be defined by specifying the new value of the state function σ depending on the update and on the old state. We will not present here the full details of the formalization, the interested reader is referred to [SLR+92].

3.3 Assignments

Assignments to Variables

The value of variables can be explicitly modified by an assignment. As usual for object-oriented languages, the inferred type of the right-hand side expression can be a subtype of the variable's type. For example, according to the state above the assignment

$$person := \textbf{pick}\ (\textbf{select}\ [name = \text{'Smith'}](PersonC));$$

has the following effect on the state: the pair mapping the variable *person* to the value p_3 is substituted by the pair $\langle person, p_1 \rangle$, since the expression in the assignment evaluates to p_1.

In general, the state σ is changed by an assignment $v := expr$ just for the variable v; the new σ applied to the variable v results in the value of expression *expr*.

Since functions are also regarded as variables, it is possible to assign sets of pairs that are produced by appropriate expressions 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

Continuing the example from above, we change the name of the person p_1 by a partial assignment to the *name* function:

$$\textbf{set}\ [name := \text{'Jones'}](person).$$

Here, the state is changed as follows: the new σ function is the same for all arguments except *name*. The value of $\sigma(name)$ is the same set of pairs as before, except that the pair $\langle p_1, \text{'Smith'} \rangle$ is replaced by $\langle p_1, \text{'Jones'} \rangle$.

In general, a partial assignment

$$\textbf{set}\ [f_1 := expr_1, ..., f_n := expr_n](obj)$$

affects the state σ in the following way: For all functions f_i $(i = 1 \ldots n)$, the new function $\sigma(f_i)$ applied to the object *obj* results in $expr_i$. All other function values of f_i as well as other variables remain the same as before the update (that is, the same as in the old σ function).

3.4 Operations for Object Evolution

3.4.1 Object Creation

The creation of an object by **create** $[T](v)$ instantiates type T and assigns the new object to the variable v (whose type has to be T or a supertype thereof). Notice that object creation involves "invention" of new OIDs, that is, the object (OID) assigned to v has to be different from all existing ones.[3] Note that, since the create operation does not affect the membership in the class $ObjectC$, the created object is not automatically persistent. If so required, it has to be included into a persistent class afterwards.

In general, **create** changes the database state by deriving a new value of the σ function from the old one as follows:
the active domains of all types T' that are supertypes of T (and T itself) now include the new object (that

[3] 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 all existing or ever existing ones".

is, $\sigma(T') := \sigma'(T') \cup \{\sigma(v)\}$, where σ' represents the value of σ before the update). Therefore, the newly created object is made an instance of T and all its supertypes to maintain the subset semantics of the subtype relationship.

We intentionally combined object creation with the instantiation of a type even though the **gain** operation could be used for the type assignment, if we wanted to be minimalistic. If, however, creation and instantiation are split into two operations **new**(v) and **gain** $[T](v)$, e.g.,

$$\textbf{new}(newobj); \textbf{gain}\ [PersonT](newobj)$$

a problem arises w.r.t. (static) type checking: since **new**$(newobj)$ creates an object of type $ObjectT$ the variable $newobj$ also has to be of that type. Even though **gain** makes the new object an instance of $PersonT$, the (static) type of the variable $newobj$ remains $ObjectT$. Therefore we have to change the static type associated with the variable in order to allow use of $PersonT$-operations on $newobj$ by a "cast" operation. We combined the three statements into the **create** operator.

3.4.2 Dynamically Acquiring More Types (gain)

The **gain** operation is used to dynamically establish new instance–type relationships between objects and types. It does not change any values except some active domains. Therefore, **gain** $[T](obj)$ changes the state σ just by including obj in the values of $\sigma(T')$ for all supertypes T' of T (including T itself). See the example above.

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 obj.

3.4.3 Dynamically Loosing Types (lose)

In contrast to the **gain** operation, **lose** deletes instance–type relationships. The effect of the operation **lose** $[T](obj)$ is that all functions that are defined on the type T or a subtype of T are no longer applicable to the object obj. As a consequence, we have to remove each occurrence of the object obj from variables, sets and functions, if they are related to T or a subtype. Before we go into the details, let us look at an example:

Assume that there are three variables $employees$, $persons$, and p with the following type declarations: $A(employees) =$ **set of** $Employee$, $A(persons) =$ **set of** $PersonT$, and $A(p) = PersonT$. Let the values of the variables be defined by the expressions:

$employees :=$ **select** $[salary > 100K](EmployeeC)$;
$persons :=$ **project** $[name, age, sex](employees)$;
$p :=$ **pick** (**select** $[salary > 100K \wedge name =$ 'Smith'$](EmployeeC)$;

Suppose that p holds an object that is also element of both set variables. The difference between $employees$ and $persons$ is just on the type level, the object sets represented by them are the same. Smith's retirement by

$$\textbf{lose}\ [EmployeeT](p)$$

has the consequence that the object representing Smith is removed from sets that are associated with the type $EmployeeT$. Therefore, the object sets denoted by $persons$ and $employees$ become different, since Smith is still an element of $persons$, but not of $employees$ anymore.

This 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.

In general, the state σ' after an operation **lose** $[T](obj)$ can be derived in two steps. First, the object obj is excluded from the active domain of type T and all its subtypes. Secondly, the new values of the variables (including functions) are specified by the following recursive definition that goes back to the new active domain:

$$\sigma'(v) := \begin{cases} \sigma(v) & v :: T \textbf{ and } T \preceq ObjectT \textbf{ and } v \in \sigma'(T) \\ \bot & v :: T \textbf{ and } T \preceq ObjectT \textbf{ and } v \notin \sigma'(T) \\ \bigcup_{v' \in v} \sigma'(v'); & v :: \textbf{set of } T \\ \{\langle x, \sigma'(v(x))\rangle \mid x \in \sigma'(T_1)\} & v :: T_1 \rightarrow T_2 \\ \sigma(v) & \textbf{otherwise} \end{cases}$$

The idea of the derivation is to use the structure of types in order to reduce the problem of specifying the new value of functions and sets to easier cases. This separation according to the type structure of variables is realized by the different cases.

The first and last cases serve as bases: In the last case nothing changes because no objects are involved. In the first case, however, v denotes an object of type T whose instance relationship is checked. If this object still belongs to the type, the value of v remains the same. Otherwise, the second case, it is replaced by the null value ($\bot$).

If the variable denotes a set, the value of each element of this set must be derived recursively. The new value is constructed by the union over all elements. [4]

Analogously, the values of functions are also checked recursively. All function values that are no longer instances of the range type are substituted by the null value[5]. Notice that this can also be applied for constructed range types, such as for the function *children*. In this case, the recursive derivation is evaluated for all children of each person. Notice that we use the recursion instead of, for example, the difference between sets and the active domain, since **set of** is a type constructor (that also allows to create sets of sets) and we do not represent the active domain of constructed types.

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

$$\textbf{destroy } (obj) \stackrel{\text{def}}{=} \textbf{lose } [ObjectT](obj)$$

There is a choice how to specify the **lose** operation. In the derivation above we changed not only the range of functions, but also the domain. Since destroying an object changes all functions that were defined for it, the identifier does no longer occur in any function: I.e., no pair with the identifier as the first component remains in the set of function mappings in the state σ. Therefore, we can reuse the identifiers of deleted object. On the other hand, we could also leave the domain of functions unchanged, because due to the type-checking the application of removed functions is not possible anyway: I.e., the set of pairs that represents the functions in the state σ is not changed. Then we cannot reuse of object identifiers, but we could make use of already specified function values in case that an object gains a lost type back again.

4 Modeling Classes and Views

Now that we have given the semantics of update operators, let us see how they maintain consistency w.r.t. classes, views, and the subclass relationship. Even though the class is a central concept in a database schema, formally it is a derived concept that can be defined using objects and functions. We introduce a new abstract object type $ClassT$ whose instances represent classes and that includes the following functions: *cname, member_type, suffp, pmemb,* and *bases*.

The function *cname* returns the name of a class, and *member_type* the type that is associated to the class members. The semantics and the values of the functions *suffp, pmemb,* and *bases* are described in the next subsections, where also is shown that the extent of a class as well as the subclass relationship can be derived from these functions.

Classes can be constrained by class predicates, which might be necessary (some-classes) or necessary and sufficient (all-classes). For some-classes, the predicate is an integrity constraint on the members. The set of members for an all-class, however, is conceptually determined by the class predicate. The same is valid for views that are sufficiently defined by queries. In order to deal with all three kinds of classes (all-, some-,

[4] The union ignores null values as elements.

[5] Similarly to sets, if in a function value the only occurrence of x is in a pair $\langle x, \bot\rangle$, this is semantically equivalent to the function being undefined for argument x.

and, view-) 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.

4.1 Constructing Necessary and Sufficient Class Predicates

The idea is to represent the extent of a class by a necessary and sufficient predicate *suffp* that decides the class membership of objects. Therefore, the current extent of a class C can be derived on demand: It includes those instances of the member type of C that fulfill the predicate *suffp*:

$$extent(C) \stackrel{\text{def}}{=} \textbf{select } [suffp(C)](\sigma(member\_type(C)))$$

Formally, since the value of the function *suffp* is a predicate, the range of *suffp* is itself a function $ObjectT \rightarrow$ **bool** [6].

The values of the predicate *suffp* can be derived as follows:

Views: For views the necessary and sufficient conditions are already specified. Considering a view definition

define view V **as** *set-expr*

the value of *suffp*(V) yields true for all objects in the defining expression, i.e., in lambda notation

$suffp(V) := \lambda x.x \in$ *set-expr*

where x is a variable of type $ObjectT$.

All-Classes: Similarly to views, the definition of all-classes already specifies the necessary and sufficient conditions. In case of the class definition

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

the value of the function *suffp* is:

$$suffp(C) := \lambda x.\ p\ (x) \bigwedge_{i=1}^{n} suffp(C_i)$$

Notice that each all-class can also be defined as a view by applying the following operations

project $[T]$(**select** $[p](C_1 \cap ... \cap C_n)$).

Therefore we do not need to regard all-classes different from views. However, we use the kind of the predicate shown above, because it is similar to what we will derive for some-classes anyway.

Some-Classes: In order to obtain necessary *and sufficient* predicates for some-classes, we have to represent the information about class membership given by the user in terms of the operations **add** and **remove**. The idea is to collect for each some-class, those objects in a set that have been added explicitly. 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 that auxiliary set. For a some-class C, the function $pmemb(C)$ returns this set of "potential members" of C. Therefore the sufficient class predicate of a class C defined by

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

looks as follows:

$$suffp(C) := \lambda x.x \in pmemb(C) \wedge p\ (x) \bigwedge_{i=1}^{n} suffp(C_i)$$

Notice how the subclass relationship (i.e., set inclusion among class extents) is maintained automatically: the class predicate requires membership in all superclasses. Removing objects from a class C can be achieved by removing them from the set $pmemb(C)$ and this automatically propagates to all subclasses. Adding objects to class C includes them in all *pmemb*-sets of class C and its superclasses.

[6] The domain of the predicate is the predefined type $ObjectT$ instead of the member type of each class, because also the (meta-)function *suffp* must be *statically* typed.

Examples. As a first example assume that we are interested besides the class $PresonC$ just in a class representing my friends that live in Zurich:

var $sue, joe : PersonT$;

class $PersonC : PersonT$ **some** $ObjectC$;

class $My\_ZH\_FriendC : PersonT$ **some** $My\_FriendC$ **where** $lives\_in =$ 'ZH';

Suppose the variables joe and sue denote two persons, Joe who lives in Zurich and Sue living in Geneva. The update operation

add $[joe](My\_ZH\_FriendC)$;

adds Joe to the set $pmemb(My\_ZH\_FriendC)$ and thus includes Joe in that class. However, consider the same update for Sue:

add $[sue](My\_ZH\_FriendC)$;

Obviously Sue will not become a member of $My\_ZH\_FriendC$, since she does not satisfy the (necessary) class predicate. Nevertheless, it is safe to include her in the set $pmemb(My\_ZH\_FriendC)$. In case she afterwards moves to Zurich, she will be included as a class member automatically.[7]

The semantics illustrated by the second example carries over to combinations of some-classes: Consider the class $My\_Reachable\_FriendC$ that denotes friends living close by (i.e., reachable in a short amount of time on foot by car or by plane). We assume that this cannot be specified by a predicate, therefore, we use a some-class.

class $My\_FriendC : PersonT$ **some** $PersonC$;

class $My\_Reachable\_FriendC : PersonT$ **some** $My\_FriendC$;

Assuming Sue is member of $My\_Reachable\_FriendC$; because of the subclass relationship, she is also member of $My\_FriendC$, i.e., internally she is contained in the set $pmemb(My\_Reachable\_FriendC)$ as well as in $pmemb(My\_FriendC)$. Removing her from the class $My\_FriendC$ by

remove $[sue](My\_FriendC)$

results in removing her from $pmemb(My\_FriendC)$. Since $My\_Reachable\_FriendC$ is defined as a subclass of $My\_FriendC$ she also disappears from $My\_Reachable\_FriendC$. However, we keep her in $pmemb(My\_Reachable\_FriendC)$, because this information was not explicitly changed. Therefore after the execution of the operation

add $[sue](My\_FriendC)$

she will again be member of $My\_FriendC$, and also of $My\_Reachable\_FriendC$. Again, this choice of update semantics is consistent with our introduced view update semantics. The other choice would have been to remove Sue from $pmemb(My\_Reachable\_FriendC)$ also.

4.2 Subclass Relationship

If we derive new classes by view definitions, we also want to position them in the class hierarchy in order to provide users with an appropriate picture of the enlarged database schema. This classification is based on the partial reflexive order between classes ($\sqsubseteq$) that is defined as follows:

$$subc \sqsubseteq supc \stackrel{\text{def}}{=} (member\_type(subc) \preceq member\_type(supc)) \wedge (suffp(subc) \Rightarrow suffp(supc))$$

Notice that the explicit separation of subtype and subset relationship alleviates the problem of deciding whether a class c_1 is a subclass of c_2 or not. If the predicate

$member\_type(c_1) \preceq member\_type(c_2)$

is not true, there is no need to check whether the predicate $suffp(c_1)$ subsumes $suffp(c_2)$. Therefore the predicate subsumption, which is in general undecidable, is not to be checked in any case.

Because predicate subsumption is undecidable in general (and remains so even in quite restricted cases [Neb90, SS89]), we use an incomplete decision procedure [Ngu91] for positioning a class in a class hierarchy (resp. testing the predicate subsumption). The predicate is decomposed into conjunctive normal form and only those conjuncts that are known to be decidable are used, the others are ignored. This procedure guarantees that the determined position is not wrong. However, there may be cases where the class could have been placed further down the hierarchy.

[7] Another choice would have been to reject this **add** operation. We choose the "optimistic" solution, because it is consistent with the view update semantics we present below.

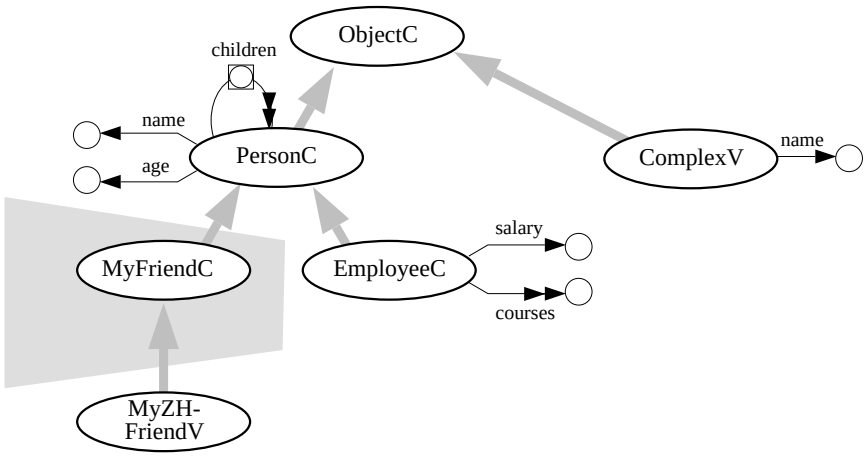

Figure 1: Subschema does not include the class in the shaded area

4.3 Updating the Extent of Views

Till now, **add** and **remove** were applied only to some-classes. Because the extents of views and all-classes are sufficiently defined, it seems unnecessary to apply **add** and **remove** to them. However, things change if we take subschemata into consideration. A subschema (or external schema) is a part of the global conceptual schema. It consists of a set of classes and views that fulfills certain closure properties (like, the ranges of functions included in the subschema must also be included) that are not discussed here any further (see e.g. [AB91, TYI88]). Applications working on subschemata that contain only parts of the class hierarchy might need operations to change the extent of all-classes or views.

For example, consider a global schema consisting of the following classes and views:

class $PersonC : PersonT$ **some** $ObjectC$;
class $My\_FriendC : PersonT$ **some** $PersonC$;
define view $My\_ZH\_FriendV$ **as select** $[lives\_in = \text{'ZH'}](My\_FriendC)$;

Assume the subschema of an application works on the class $PersonC$ and the view $My\_ZH\_FriendV$, but excludes the class $My\_FriendC$ (see Fig. 1). Because the some-class $My\_FriendC$ is hidden in the subschema definition, there is no way to add persons into the view $My\_ZH\_FriendV$ until now. In order to handle such cases we extend the semantics of the operations **add** and **remove** such that applying these operations to the view $My\_ZH\_FriendV$ becomes possible.

In case of selection views (also extend- and intersection-views) the semantics of the **add** operation as described above can be applied, since adding an object to the view results in adding it to all superclasses of the view, i.e., also the class the view is defined on. However, the semantics for removing an object is not applicable, since removing it from the view but not from the class the view is defined on would contradict the view definition. In case of union- or projection-views, that result in superclasses of the base classes, also the **add** operation is not applicable, since adding objects to a projection view would make the extents of the view and the underlying class differ [SLT91].

The problem is caused by trying to use the *pmemb* function for all-classes and views. Since their extent, however, is completely determined by the class predicates, inconsistencies are bound to arise. Instead, we have to explicitly maintain the relationship between a view and its base class(es). This can then be used to apply the operations **add** and **remove** to the base classes of this view. Therefore the extent of a view remains consistent to its definition, i.e., the anomalies sketched above cannot arise.

In order to get an handle which *pmemb*-sets of classes must be changed in case of adding or removing objects to or from views, we use the meta function *bases* that yields a set of some-classes. The value of *bases* can be derived by induction as follows: We define $bases(c) = \{c\}$ for all some-classes c as an anchor. The following recursive predicate derives the *bases* of all-classes and views that are defined by an expression e:

$$bases(e) := \begin{cases} e & e is a some-class \\ bases(e') & e = \textbf{project}\ [...](e'), \textbf{extend}\ [...](e'), \textbf{select}\ [...](e') \\ bases(e') \cup\ bases(e'') & e = e' \cap e'', e' \cup e'' \end{cases}$$

Projection views as well as views defined by the **extend** operation denote exactly the same set of objects as the expression they are defined on, since the operations only change the associated type of that set. Therefore **add** and **remove** should be propagated straightforwardly. The same is true for selection views. Adding and removing objects to/from union and intersection views propagates to both defining expressions. The problems arising especially with that default settings for intersection and union views are described in [SLT91] in more detail. Notice that this procedure captures also view definitions on views.

Considering the above example, the following values of the *bases* function can be derived:

$bases(PersonC) = \{PersonC\}$;
$bases(My\_FriendC) = \{My\_FriendC\}$;
$bases(My\_ZH\_FriendV) = \{My\_FriendC\}$;

If we add the following definitions to the schema

class $EmployeeC : EmployeeT$ **some** $PersonC$;
define view $ComplexV$ **as**
project $[name](EmployeeC\ \cap My\_ZH\_FriendV)$;

the value of $bases(ComplexV)$ is derived by the union of the $bases(EmployeeC)$ (which is $\{EmployeeC\}$) and the *bases*-value of the view $My\_ZH\_FriendV$ (which is $\{My\_FriendC\}$). That is, the bases of a view might contain classes that are neither sub- nor superclasses (see Fig. 1).

4.4 Add and Remove as Derived Update Methods

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 algebra, but derived as follows[8]:

$$\textbf{add}\ [e](class) \stackrel{\text{def}}{=} \textbf{apply\_to\_all}\ \Big[\textbf{set}\ [pmemb := pmemb\ \textbf{union}\ \{e\}](c)\Big] \Big(c : \textbf{select}\ [\ \emptyset \neq \textbf{select}\ [x \sqsubseteq c](x : bases(class)) \textbf{ and } bases(c) = \{c\}](c : ClassC)\Big)$$

$$\textbf{remove}\ [e](class) \stackrel{\text{def}}{=} \textbf{apply\_to\_all}\ \Big[\textbf{set}\ [pmemb := \textbf{select}\ [x \neq e](x : pmemb)](c)\Big] \Big(c : bases(class)\Big)$$

The **add** operation is defined by applying the **set** operation that includes the object denoted by e into 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 some class included in $bases(class)$. 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 $bases(class)$.

Continuing the example, if we add a person not living in Zurich to $My\_ZH\_FriendV$, he/she becomes member of the class $My\_FriendC$, but not of the view until he/she fulfills the selection's predicate. Removing a person from the view $ComplexV$ is propagated to the classes $PersonC$ and $EmployeeC$.

5 Related Work

In contrast to object-oriented programming languages (like C++, Eiffel, 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

[8] The class $ClassC$ belongs to the meta level and represents all classes of the database

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** $[ObjectT](obj)$ that destroys all occurrences of an object from **remove** $[obj](ObjectC)$.

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. 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.

Knowledge representation systems such as BACK [PSKQ89] and KRISYS [DLM90] allow to specify a wide variety of integrity constraints, but lack powerful update mechanisms. They allow to add information by "telling" new facts. However, because deletions can produce ambiguities, they are either forbidden (e.g., in case of [PSKQ89]) or subject to several restrictions (in case of [DLM90]).

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. Constraints on classes are used as membership criteria rather than integrity 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. Essentially, set variables are treated as temporary classes. Clearly, the semantics presented here need not be the ultimate solution. Several points have been identified, where we could take other choices. The main contribution is that we (i) proposed a complete and consistent update semantics and (ii) we identified this decision space.

Besides the efficient implementation of the proposed operations, future work will focus on set-oriented updates. In general, set-oriented updates are hard to define consistently, because interdependencies, e.g., between qualifying predicates and updates, or between two updates in the set, often lead to non-deterministic semantics. With only one single update operation in the **apply_to_all** iterator, we can easily define consistent semantics by proceeding in three steps: (i) identify all objects that are going to be updated; (ii) evaluate any retrieval expression in the update statement (in the old database state); (iii) apply the actual update to the objects, one at a time. Because steps (i) and (ii) both refer to the unique initial state before all updates, there can be no ambiguity (order-dependence in step (iii)). This is no longer true, if we allow update sequences in **apply_to_all**. We are working on the exact restrictions that are necessary in that latter case.

References

[AB87] M.P. Atkinson and O.P. Buneman. Types and persistence of database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[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 Theory*, 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.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[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.

[DLM90] S. Dessloch, F.-J. Leick, and N. M. Mattos. A state-oriented approach to the specification of rules ans queries in kbms. Technical Report 4/90, ZRI, University of Kaiserslautern, July 1990.

[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.

[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, November 1990.

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

[Neb90] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43:235–249, 1990.

[Ngu91] H.-M. Nguyen. Classifying classes and views in class hierarchies. master thesis, Department of Computer Science, ETH Zurich, July 1991. In German.

[NP90] B. Nebel and C. Peltason. Terminological reasoning and information management. In D. Karagiannis, editor, *Information Systems and Artificial Intelligence: Integration Aspects*, Ulm,Germany, March 1990. LNCS 474, Springer Verlag, Heidelberg.

[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, sep 1989.

[SLR+92] M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, and M. Tresch. The COCOON object model. Technical report, ETH Zürich, Dept. of Computer Science, 1992. In preparation.

[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. Technical Report TR 90–11, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, April 1990. To appear in ACM TOIS.

[SS89] M. Schmidt-Schauß. Subsumption in KL-ONE is undecidable. In *Proc. First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 421–431, Toronto, Ont., 1989.

[SS90a] 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.

[SS90b] M.H. Scholl and H.-J. Schek. A synthesis of complex objects and object-orientation. In *Proc. IFIP TC2 Conf. on Object Oriented Databases (DS-4)*, Windermere, UK, July 1990. North-Holland. To appear.

[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.