Erschien zuerst in: *Proceedings of the 2. International Conference on Deductive and Object Oriented Databases. 1991*

Updatable Views in Object-Oriented Databases

Marc H. Scholl, Christian Laasch, Markus Tresch

ETH Zurich, Dept. of Computer Science
Information Systems – Databases
ETH Zentrum, CH–8092 Zurich, Switzerland
E-mail: <*lastname*>@inf.ethz.ch

Abstract

Object-oriented database systems (ooDBMSs) are supposed to offer at least the functionality available in commercial relational DBMSs of today. One important consequence of this is that they have to provide a separation of the global (conceptual) database schema from the external schema ("subschema") of a particular task. *Views* are a mechanism to realize this data independence. In addition, they also support multiple levels of detail, security and authorization, and interoperability in a heterogeneous environment. In a relational DBMS, views are defined by queries. However, they can not be freely updated. We describe concepts of an object model and query language that are necessary for object view definitions. We show that updating object views is much more feasible than in the case of relational views. The key property of a query language leading to this result is *object preserving operator semantics*. That is, in contrast to many previous object algebras, query results are sets of *existing objects* instead of data tuples or new objects. Consequently, we have to solve the classification problem: where to include the view in the type and class lattices.

Keywords: Object-Oriented Databases, Object Model, Object Algebra, Views, View Updates.

1 Introduction

Relational database systems provide a very simple, yet powerful view mechanism: any query, since it returns a relation, can be used to define a view, that is, an additional (derived, virtual) relation. Abstracting from the details of a particular query language, views are defined by a statement such as "**define view** *name* **as** *query*." In this paper, we show how the same can be done in an object-oriented database system. Relational views can not be updated freely, since it is often ambiguous how to trace view updates back to updates of base tuples. Views containing the key of their (one) underlying base relation can be updated. In an ooDBMS, since objects have an identity independent of their associated values, this problem disappears, if the query language preserves objects' identities. We discuss properties of an object-oriented query language that are necessary to allow for updatable object views.

There are the following four major reasons why to support view capabilities in an object-oriented database system: (1) size of the global schema (just for manageability we may need to split into smaller pieces); (2) authorization (show only relevant data, that is, restrict access by visibility); (3) integration in a heterogeneous environment (provide customized 'data models' for different clients); and (4) "upward compatibility" (views have proven useful in relational DBMSs, ooDBMSs are supposed to extend the functionality of RDBMSs).

In fact, there is large consensus that views should be definable in an ooDBMS. As ooDBMSs are a quite novel area of research, we refer to the literature on object-oriented data models instead of implemented systems. There is a constantly increasing number of such publications, however, only a few of them mention views at all. Some list view definition capabilities as an objective in their query

language design, but do not elaborate on the problem after describing the query language. Many of the proposed object-oriented query languages have an inherent problem: They either generate *new* objects or just data tuples (*values*) as query results. In the former case, views defined by these queries would not contain base objects, so updates do not propagate. In the latter case, no methods can be applied to query results, nor will updates make any sense. We will come back to these proposals in the discussion after we have presented our solution.

The key idea that leads to a fully flexible view definition capability of a query language is to make the operations *object preserving*: The objects contained in the result of a query *are the input objects* (some of them, in general). Object preservation is important for it makes updates propagate automatically: an update to an object in a view *is* an update of a base object. In the context of views there are two directions for update propagation:

1. Updates on base objects are reflected in views. This distinguishes views from snapshots, where copies of the base objects are taken that are kept separate and may change independently.
2. Updates on view objects should propagate to the underlying base objects where possible. This view update problem is known to be difficult in the relational context. Since there, "object identity" is tied to key attribute values, only views containing the key may be updated [GPZ88, SLW88].

Obviously, with object preserving query semantics, both directions of update propagation will follow naturally. We have shown earlier [SS90a] how to define an object preserving query algebra. It is particularly noticeable that object preserving semantics can be given to all query operators, including joins and complex expressions. Thus, there is no limitation of the expressive power queries used for view definitions. Here we will show that views defined by such a query language can almost freely be updated.

We present our results in the context of a particular object model, COCOON. Even though object models usually differ quite a lot in the particular terminology used or in several details, there is broad consensus on the high-level concepts that should be provided. The object model we use is fairly standard, so the results can be interpreted and adopted to other models as well. Besides object preservation, further important concepts of COCOON, which are exploited in our approach to view updates, are: *multiple instantiation* (objects may be instances of more than one type and they may change types dynamically), *multiple class membership* (objects may be members of several classes at the same time; class membership may also be changed dynamically), and *class predicates* (defining necessary—and sometimes also sufficient—constraints on members of superclasses for being a member of a subclass).

The organization of the paper is as follows: in Section 2 we review the concepts of the COCOON object model. First we summarize the basic terminology and the object preserving semantics of the query language operators. Then we focus on the generic update operations. Section 3 shows how to define views by queries, how to position the result types in the type lattice, and how class predicates determine the position of the view in the class hierarchy. In this paper, we do not give formal definitions, rather we use an example-driven presentation. However, we discuss the general problems and solutions after presenting representative examples. Section 4 compares our approach with some other recent work. Finally, Section 5 concludes with a summary of the key properties that led to our results.

2 An Object Model

Essentially, the COCOON model as described in [SS90a, SS90b] is an object-function-model (cf. [WLH90, DMB+87, Day89]). Its constituents are *objects*, *functions*, *types* and *classes*. The query language, COOL, offers object-preserving as well as object-generating generic query operators plus generic update operators. In this paper, we do not use object-generating query operators. The key objective in the design of COOL was its set-oriented, descriptive characteristics, similar to a relational algebra. In fact, COOL can be seen as an extension of our nested relational query language [SS86].

2.1 Basic Concepts

COCOON is a *core* object model, meaning that we focus on the essential ingredients necessary to define a set-oriented query language. For instance, tuples as a type constructor are excluded from the core. Basically, all we need is objects (concrete and abstract) and one type constructor, namely *set*. Other features can be added later due to the orthogonality of the language [SLR+91].

Objects are instances of abstract data types (ADTs). 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 similar to [Bee89].

Functions are described by a name and signature (i.e., domain and range types). Functions can be single- or set-valued, they are the interface operations of types. The implementation is specified separately (we do not show this here). Notice that we distinguish *retrieval functions* from *methods*, that is to say, functions with side-effects. Unless stated otherwise, we use the term *functions* in the general sense in the sequel. Retrieval functions are a uniform abstraction of "attributes" and "relationships" of classical data models. A useful feature is the capability of defining inverses of functions. This integrity constraint is enforced by the system during updates. For an example, see below under "types".

Types describe the common interface of all instances of that type, that is, the collection of applicable functions and their interaction. The latter gives the function's semantics, which can be specified in the form of axioms, for example. We do not consider this any further here. So, the definition of a type basically consists of two parts: a set of functions and a type name.[1] The following example defines a type *CompanyT* with four functions *cname, staff, locations,* and *open_new_branch*.

type *CompanyT* **is_a** *ObjectT* =
cname : string,
staff : **set_of** *EmployeeT* **inverse** *employer,*
locations : **set_of** *CityT,*
. . .
method *open_new_branch (CityT);*

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, e.g.

type *EmployeeT* **is_a** *PersonT = ... ;*

then every instance *e* of the subtype *EmployeeT*, is also an instance of the supertype *PersonT*. This is called *multiple instantiation*. Consequently, all functions defined on the supertype, *PersonT*, apply to the subtype, *EmployeeT*, too. Further, functions on the subtype may be more restricted than they are on the supertype (that is, their range may be a subtype of that defined in the supertype). Subtyping defines a partial order "$\preceq$" on types. Internally, the type system is completed to form a lattice of types (the powerset lattice of the set of all functions in the database schema), such that for any two types their least upper bound and greatest lower bound are always defined. The top element of the lattice is the most general type *ObjectT* (therefore, all instances of defined types in the database are also instances of *ObjectT*). 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. We strictly distinguish types from classes 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). A class *C* is a collection object (an instance of the metatype class). For each

[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.

class, there is a set of objects, called its extent (*extent(C)*). The elements of that set, called *members* of the class, are instances of a type, the *member_type(C)*. Due to multiple instantiation, individual members may be instances of several other types too. Even though classes represent polymorphic sets, type checking of our language always refers to the unique member type of the involved sets. A class definition specifies a class name, the member type, the names of superclasses, and an optional class predicate (for the latter two, see below, under subclasses):

class *EmployeeC : EmployeeT* **some** *PersonC;*

Due to the separation of types and classes, there may be any number of classes for a particular type (for instance, more than one as the result of selection operations, see below, or none, if we are not interested in maintaining an explicit extent of that type).

Subclasses. 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. As will be seen, these two often correlate, but they need not. Therefore, we will always distinguish carefully which one of them holds. 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. Consider the following example:

type *PersonT* **is_a** *ObjectT = ...;*
type *EmployeeT* **is_a** *PersonT = ...;*
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*. The predicate given for class *YoungC* is a constraint that all members of *YoungC* have to be younger than 30. 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. Notice that, unlike e.g. [D$^+$90, Kim89, SÖ90], we define the extent of a class *C* to include the members of all its subclasses.

2.2 Object Algebra

One of our primary goals in the model is a strongly-typed algebra that allows for *static type checking* [BTBO89]. This is achieved by using the type associated with a class to check whether the operations on the class members are legal. 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. These are collectively called <*set-expr*> in the sequel. Formally, a class name *C* as an argument is a shorthand for *extent(C)*.

Variables and Assignments. Due to the set-oriented style of the query language, objects are typically unnamed. However, variables can be used as temporary names ("handles") for objects. They have to be declared with their type, such that compile-time type checking applies to variables too. For example,

var *My_Chevy* : *AutomobileT;*

declares a variable *My_Chevy* of type *AutomobileT* that can, for example, be assigned the result of an object creation in order to identify the new object in subsequent (update) operations.

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, i.e., it is *type(*<*set-expr*>*)*.

Projection (**project** [A_1, ..., A_i] (<*set-expr*>)). The output of a projection is a set with a usually new type, a supertype of the input type, as less functions are defined on the output, namely only those listed in the projection. All objects of the input set are also elements of the output set (*object preservation*).

Extend (**extend** [<*fname*>:=<*expr*>,...] (<*set-expr*>)). Projection eliminates functions, extend defines new derived ones. Obviously, each given function name <*fname*> must be different from all existing functions for the type of the input. <*expr*> can be any legal arithmetic-, boolean-, or set-expression. The objects of the input set are preserved, that is, **extend** returns a set with the same objects as the input, but with a new type, a *subtype* of the input type (all the old functions plus the new ones are defined on it).

Set operations. As the extent of classes are sets of objects, we can perform set operations as usual. One notable point is the criterion for duplicate elimination (or equality determination): equality of abstract objects is what this usually called identity. So, this is the notion that is used in the set operations[2]. 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 difference operation yields a subset of its first argument with the same type; finally, intersection results in the greatest common subtype.

These are the basic *object preserving* query operators of our algebra. Other operators, such as join can be derived from them (cf. Section 3.5). The complete algebra includes an operator (**extract**) for generating sets of *tuples* as query results to communicate with value-oriented environments [SS90b]. Formally, we do not provide object generating *query* operators. However, new objects can be derived by combining a query with an update operator, **insert**, that generates the new objects (see [SLR+91] for details).

2.3 Update Operations

One of the main objectives of the object-oriented approach is to use only *type-specific* update operations, so as to guarantee consistency. We support this functionality in our model by *methods*: users can apply type-specific methods to update objects.

Update. As a first extension, like in the relational case, we want to be able to specify *set-oriented* updates; that is, identify a set of objects to be updated (e.g., by a query) and apply the update method to all qualifying objects without programming an explicit loop. This set-oriented update mode can help increase the performance of updates. We provide a descriptive iterator that takes the update method as a parameter:

update [*m*] (<*set-expr*>);

The semantics of this statement is obvious: method *m* is applied to all objects returned by the set expression. **Update** could also be called “apply_to_all”.

As a second extension to type-specific update operations, we provide a set of generally applicable *generic update operations*. Such generic update operations can be used by type implementors to define type-specific methods. The generic update operations include **insert** and **delete** to create and destroy objects (possibly in a set-oriented fashion), **add** and **remove** to add existing objects to a set or to remove them from it, respectively, and **set** to set functions to new values.

Insert takes as arguments a class *C*, with member type of, say, *T*, and assignments of values to functions defined on *T*. An instance of *T* is created and added to the specified class, the functions listed are set to the given values. The class predicate is checked. If violated, the insert is rejected by the integrity checker. The result of the **insert** operation is the newly created object. For example, we create a new instance in class *EmployeeC* by:

[2] This is the problem of determining uniqueness of objects as mentioned in [Kim89]. The other two problems mentioned there, heterogeneity and scope of classes, are treated differently in our model: classes are considered homogeneous (all member objects are instances of the member type), and the extent always includes all subclasses (see above).

js := **insert** [*name*:='John Smith', *address*:=..., ...] (*EmployeeC*);

Here, *js* has to be a variable of type *EmployeeT*. Notice that we do not create objects that are not members of any class.

Delete (<*set-expr*>) destroys all objects *v* returned by <*set-expr*>: they are removed from all classes and function values in the database.

Add and Remove have a weaker effect than insert and delete: they have no impact on the existence of objects. Rather, existing objects are added to or removed from a set. Both operations take two sets of objects (e.g., class names or queries) as parameters. The first set of objects is added to or removed from the second. For example, let *E* be a query returning some employee objects and *c* a variable holding a company object, we can make all employees in *E* work for *c* by:

add [*E*] (*staff*(*c*));

Notice that we applied the update to the *staff* function defined on companies, the function *employer* is updated automatically, since it is defined to be the *inverse* of *staff*.

Add and **remove** are also used to change class membership of objects dynamically. For example, if *p* is a person object, **add** [{*p*}] (*EmployeeC*) makes *p* a member of class *EmployeeC*, and thus an instance of *EmployeeT*. Conversely, **remove** [{*p*}] (*EmployeeC*) takes *p* out of *EmployeeC* (but leaves it in *PersonC*, and all other superclasses of *EmployeeC*, if any). Class predicates are checked upon **add** and **remove** operations. They may lead to 'rejection' of the operation: an explicit **add** will be ignored if the class predicate is not fulfilled, and an explicit **remove** will be ignored if the class has a sufficient class predicate that holds for the object. Generally, membership in a class with sufficient class predicate is not manipulated explicitly by **add**/**remove**, but only implicitly by changing function values used in the class predicate. For more details on the effects of updates in connection with class prediactes see [LS91].

Set assigns new values to functions. Let *e* be a variable holding an employee, then **set** [*salary*=3,000] (*e*) assigns a new salary to *e*. **Set** is used inside of **update** and **insert** and is usually written as an assignment (*salary*(*e*) := 3,000). Type implementors can specify a function-specific method (*set_salary*, in the example) to perform the assignment [FBC+87]. Arbitrary computations can be performed in such a method, e.g., to check some constraints, to update additional information, or even to refuse the update. For example, updating a derived function may be implemented as either changing the underlying values or refusing the update.

Nesting of Update Operations. Notice that the generic update operations may not only be applied to classes, but also to set-valued attributes and query results. Just as type-specific methods, these operations are applicable inside an **update**. For instance, we can **add** all employees making more than 50K to the staff of the CS department by the following operation:

update [**add** [**select** [*salary* > *50k*] (*EmployeeC*)] (*staff*)] (**select** [*name* = *'CS'*] (*DepartmentC*)).

The example works as follows: a subset of the department class is identified by the **select**ion (this may or may not be a singleton set here). The departments returned are **update**d by **add**ing to the set-valued function *staff* those employee objects returned by the "inner" **select**ion on the employee class. So, what happens is that function *staff* for the CS department(s) changes its value to include the result set of the inner selection. Notice in particular, that *no objects are copied*, since objects can be elements of an arbitrary number of sets without replication (object sharing!).

3 Views and Updates

Classes contained in the global schema are called *base classes*, objects in these classes are stored as *base objects*. Queries can be used to define additional (derived) *view classes*. The extent of views is usually

not stored explicitly, but rather computed from the view-defining query upon demand[3]. Views provide a specialized interface to some base objects. A user or an application program 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. Some consistency constraints (concerning closure) have to be enforced for subschemas, we do not elaborate on them here (see, e.g. [HZ90]). Our view mechanism simply allows arbitrary queries to serve as view definitions, exactly as in relational DBMSs:

define view <*name*> **as** <*query*>;

After the execution of this statement, <*name*> will appear as a (persistent) class of the database, just like base classes. The only difference is that the extent of the view is defined by the <*query*> expression, based on the extents of other (view and/or base) classes. Notice that views may, of course, be defined based on other views. For ease of discussion we will, however, call the underlying class(es) for each view 'base class(es)'.

Because views are just another kind of classes, they can be used as arguments of update operations, just as base classes can. 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 these.

3.1 Selection Views

Selection views create subclasses containing all objects satisfying the selection predicate. The type remains unchanged. As an example, we define a view of all companies having at least one branch in New York (see Fig. 1):

define view *NYCompV* **as**
  select [$\emptyset \neq$ **select** [*name* = 'New York'] (*locations*)] (*CompanyC*);

This is a well-defined selection, since "$\emptyset \neq$ **select** [*name* = 'New York'] (*location*)" is a valid predicate, which happens to include a (sub) query.

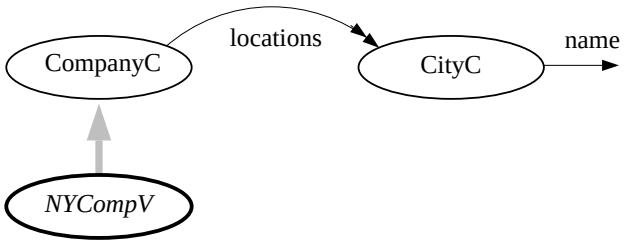

Figure 1. Selections create subclasses.

Insertion, addition, deletion or removal of an instance in the view *NYCompV* works on the base class *CompanyC*. Insertion of objects that do not fulfill the selection predicate lead to a problem (the same problem occurs with updates that change the truth value of the class predicate). There are two choices how to react: (i) reject such insertions/updates (since they violate some kind of closure property [Heg90, HZ90], or (ii) allow them by inserting into the base class/updating the object and determining "visibility" in the view by the selection predicate. Commercial relational DBMSs implement the second choice. Note that the problem is not due to ambiguities in the update translation, but rather one of what you *want to allow*. One might argue which way to go. The following example creates a new object *ThisComp* through the view (assuming variable *paris* holds a city object with name $\neq$ 'NewYork'):

ThisComp := **insert** [*cname := ..., locations:= { paris }, president:= ...*] (*NYCompV*);

[3] Even though views may be materialized for performance reasons, but then care must be taken to keep them consistent or to invalidate them.

If this view update is admitted, the classifying predicate of the selection view determines that the inserted object does not satisfy the condition for view membership. Thus, the new object is not visible in the view. It would, however, be inserted into the base class. Similarly, when some company *c* is updated by adding a New York branch office to it's *locations*, the update makes *c* a member of the *NYCompV* view. If the last branch office in New York of a *NYCompV* object is removed, the company is no longer a member of the view class.

In any case, to guarantee integrity after updates, the system has to automatically re-classify objects after updates: A selection view defines a derived class *V* as a subclass of its base class *B* by a predicate *P*. The effect is exactly the same as if the DB schema contained the following class definition (both, *B* and *V*, have as member type some type *T*):

define class *V*: *T* **all** *B* **where** *P*;

In this case, we clearly expect the system to guarantee *V*'s extent to be exactly that subset of objects in *B*'s extent satisfying predicate *P*. That is, even if we choose solution (i) for the above view update problem, we will have to change class membership automatically when function values are changed that affect class predicates. Therefore, we adopt the second choice and admit such insertions/additions into selection views. A more 'conservative' view update semantics can always be achieved by overriding the generic **add** and **insert** operators. Other modifications of objects in the view pose no problems, all methods from the base class are available.

3.2 Difference Views

A difference view is a subclass of the minuend and contains all objects that are not member of the subtrahend. The result type is that of the minuend, that is, the type of the subtrahend does not affect the operation at all. As an example, we can define companies having at least one branch in New York but do not produce automobiles:

define view *NYDiffCompV* **as**
NYCompV − (**select** ['Automobile' ∈ *products*] (*CompanyC*));

In general, there always exists a selection expression that yields the same result as the difference. Such selection expressions are applied to the minuend with the predicate that excludes objects contained in the subtrahend. In the example, we use a dummy variable *c* to formulate the query (similar to the **self** or **this** standard identifiers of other languages):

define view *NYDiffCompV* **as**
select [*c* ∉ (**select** ['Automobile' ∈ *products*] (*CompanyC*))] (*c* : *NYCompV*);

Because difference is a special kind of selection, we do not need to discuss updates any further, we already know how to update selection views.

3.3 Union Views

A union view is a superclass of the two classes constructing it. Its extent contains all members of the base classes, and its type is the intersection of the two sets of functions applicable to the bases classes (that is, the least upper bound in the type lattice). If a function *f* is defined in both base classes, the union type contains *f* with the union of the both ranges as range of *f*[4]. For example, we create a view of legal entities that contains companies and persons:

define view *LegalEntityV* **as** *PersonC* **union** *CompanyC*;

[4] As already mentioned, function names are assumed to be unique. Therefore, if a function name occurs twice, both occurrences mean the same function (possibly with different range restrictions).

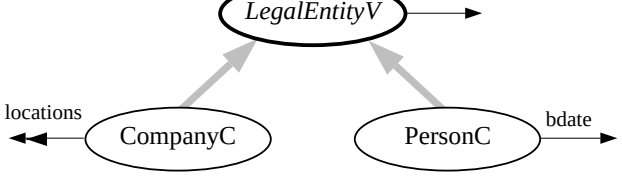

Figure 2. Unions create superclasses.

The function *name* is the only one applicable to members of the *LegalEntityV* view. All other functions of companies or persons are only available in the context of either *CompanyT* or *PersonT*.

As for updates there is one inherent problem in union views: If we want to **insert** or **add** objects to the view, we have to propagate these to at least one base class. But, which one to choose? In general, the choice is ambiguous, so we either have to disallow such updates, or to include the object into both base classes. Class predicates associated with the base classes, if any, could eventually determine which one will contain the new object. Since class predicates are optional, however, we can not rely on this solution. Our choice is therefore, to propagate **insert** and **add** to both base classes of union views. Nevertheless, users can define **insert** and **add** methods specific to the view's type.

All other generic update operations are applicable to union views (**delete**, **set**, **remove**) in the obvious way: They propagate to both base classes, *CompanyC* and *PersonC*. Particularly, **remove** puts objects out of *both* extents (if they were members). Further, as with all views, any methods included in the result type can be applied.

3.4 Intersection Views

Views defined by the intersection of two classes are subclasses of them containing objects that are members of both base classes. Therefore the result type includes the functions of both classes' types (it is the greatest lower bound of the input types). As an example, we can define working students via the following intersection:

define view *WorkStudV* **as** *StudentC* **intersect** *EmployeeC*;

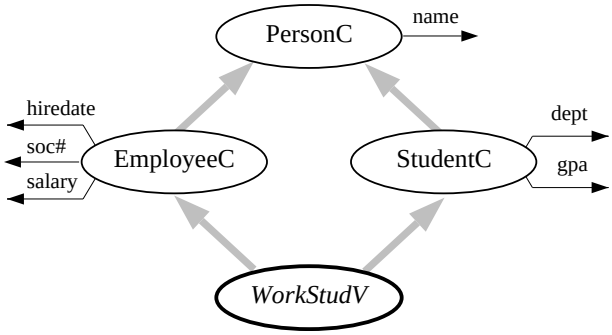

Figure 3. Intersections create subclasses.

Users of this view are permitted to apply all functions (and methods; none is shown here) of students (name, dept, and gpa) and of employees (name, hiredate, soc#, and salary). These functions applied to the members of the view yield the same results as applied to the base classes, because the objects are the same.

Analogously to the union operation, there is a problem in removal from intersection views: due to the symmetry of intersection, it is ambiguous to which underlying class the **remove** operation should be propagated. Our choice is, again, to propagate to both base classes.[5] Inserting or adding objects to the view propagates and adds them into both base classes. Propagation of deletion or modification (**set**) is straightforward, because of object preservation.

[5] Again, users can define type-specific **remove** methods as appropriate.

3.5 Projection Views

Projection views are superclasses of their underlying base class with changed types. The list of functions that can be used in the view determines the result type. Projection "has no effect" on the instance level, it just affects object types by "hiding" some functions (like a "type cast"), that is, the extent is the same as the base class. For example, we may wish to "hide" the salary from the users of a view *PublicEmpV* on *EmployeeC* by projecting away the *salary* function:

define view *PublicEmplV* **as project** [*name,soc#,hiredate*] (*EmployeeC*);

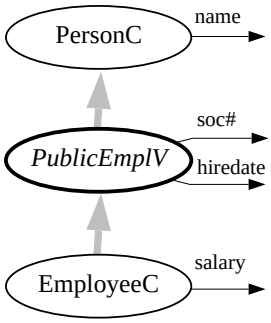

Figure 4. Projections create superclasses.

Users of this view operate on exactly the same objects (the same extent) as those in *EmployeeC*, they can use all the employee functions (including methods), except *salary* (see Fig. 4)[6]. Therefore the type of *PublicEmplV* is a supertype of *EmployeeC* but a subtype of *PersonC*.

Updates performed on the view will also affect the members of class *EmployeeC*, since they *are the same* as those of *PublicEmplV*. Inserting, adding, deleting and removing objects in a projection view is feasible. Suppose we create a new employee in the view, using the following operation:

NewEmp := **insert** [*name* := ..., *addr* := ..., *soc#* := ...] (*PublicEmplV*);

The **insert** works as if it were applied to the base class, *EmployeeC*. The only difference is that we can not assign a salary using the generic assignment (**set**) operator, since this would result in a type checking error. However, as the types of objects also contain methods, projections can also specify the subset of methods retained in the view. In particular, our example projection may hide a retrieval function (such as *salary*) but contain the corresponding update method (such as *give_raise* or *assign_salary*). In this case, a value associated with the object can be updated but can not be retrieved by the users of this view. As a consequence, projection views can be used to restrict access privileges in a rather elaborate way: modification can be permitted where retrieval is forbidden or vice versa. In case a class predicate on the base class states that certain functions (such as salary) have to have a value (i.e., are total), insertions into projection views that eliminate these functions are automatically refused by checking the class predicate upon insertion.

[6] Here we see that the objects are instances of two types at the same time. In order to decide type safeness of function applications, type checking works on the *class* level: using the *salary* function on class *PublicEmplV* would result in a (compile-time) type error, even though each employee *object* "has" both types, *EmployeeT* and *PublicEmplT*.

3.6 Extend Views

Views defined by **extend** are subclasses, the type of which contains some additional, derived functions. The view's extent is identical to that of its superclass. We can, for instance, define a derived function *resp_for* ("responsible for") for employee objects by the following query (see Fig. 5):

define view *PresEmplV* **as**
extend [*resp_for* := **select** [*president=e*] (*CompanyC*)] (*e* : *EmployeeC*);

Inserting, adding, deleting, removing objects to/from the view propagates to the base class and vice versa. Further, all methods can be applied.

In this respect, projection views and extend views behave similar. Both create new classes, whose extent has to be identical to that of the underlying classes. The new classes could have been introduced in the DB schema equivalently by:

define class *V*: *VT* **all** *C*;

That is, *extent*(*V*)=*extent*(*C*) and $VT \preceq$ *member_type*(*C*) for **extend** and vice versa for **project**.

The values of the extended function, *resp_for* in the example, can not be updated directly by assignments (**set**). This is not particular to views, but occurs with all derived functions, because their values are calculated. They are updated indirectly instead, by changing other functions' values. In the example above, changing the *president* value of a company object may update the value of *resp_for* for some employee objects.

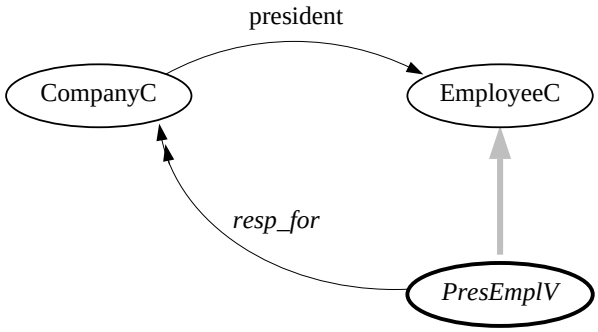

Figure 5. Extend defines subtypes with new functions.

Disallowing the manipulation of extended functions' values seems quite a strong restriction. However, all this really means is that we can not assign a new value "out of the blue" to such a function. But if the function value is a (set of) other object(s), we can update this (or these) object(s). In the example, since *resp_for* returns companies, we can apply all (generic and type-specific) update operations to these companies. Some of these updates may have an effect on the derived function: if we change the value of the *president* function for the objects in *resp_for*, this may lead to *resp_for* return another value next time for the current employee!

3.7 Join Views

Joins are usually used to combine objects that are related by some 'relationship' expressed as a logical predicate over their values in some functions (attributes). The relationships that are of most interest are realized via functions in our model. So, only a few other relationships may eventually need to be established by some kind of join.

We have already seen a way of establishing new 'relationships' among objects, namely, defining new functions between them: the **extend** operator does this. So, one way of expressing *join views* in our model is to define the required relationship as a new function connecting the *matching pairs* of objects.

For example, to support queries like "employees living in a town in which some (not necessarily *their*) company is located" — which would require an (equi-) join of the two corresponding relations

on the predicate "*COMP.LOC=EMP.ADDR*" in a relational DB —, we can define a view *OuterJoinV* over the employee class by introducing a new function *LocalComp* returning companies located at the employee's home town by the following **extend** operation:

define view *OuterJoinV* **as**
extend [*LocalComp* := **select** [*address* ∈ *location*] (*CompanyC*)] (*EmployeeC*);

The view *OuterJoinV* will be a subclass of *EmployeeC* containing all *EmployeeC* objects. The new function returns the set of "join partners" (which may also be empty). To implement an inner equi join, an additional selection with the predicate "*LocalComp* $\neq \emptyset$" must be applied, for example:

define view *InnerJoinV* **as select** [*LocalComp* $\neq \emptyset$] (*OuterJoinV*);

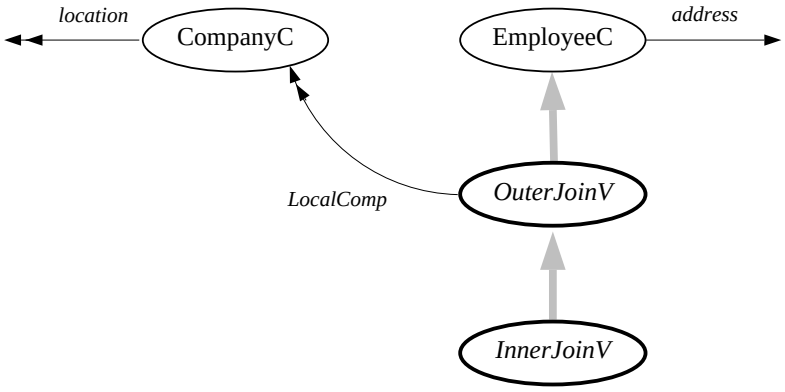

Figure 6. Outer and inner join defined by extend views.

"Joining" objects by means of the **extend** operator is *object preserving*, that is, the result of this join is a new "relationship" among *existing* objects. Update operations on "join" views behave exactly as discussed for extend views. That is, all updates are allowed, except simply assigning a new value to *LocalComp*. For example, values of employee and/or company functions may be changed, deletion and removal of employees is also possible, and all type-specific methods may be applied. Particularly, an object may be added or removed from the view indirectly by changing a company's location, or the address of an employee (cf. selection views).

The reader may have noticed that 'joining' by means of the **extend** operator is not a symmetrical operator. We have discusses this problem in depth in [SS90a]. In short, we can provide a symmetric solution by extending the other class with the inverse function. Alternatives for join-like operations include object generating ones and tuple (pairs of objects) generating ones. Comparisons with other query languages' solutions to this problem are also contained in that other paper.

3.8 Typing, Classification, and Composite Queries

In the sections above we have defined the query operators in terms of result type and extent. We have also shown how updates on views can be propagated to the underlying base class(es). The types and extents of query results have been positioned *relative to their operand class(es)*. To complete the discussion, let us now discuss three remaining problems (cf. [Kim89]):

1. What is the *final* position of the result type in the type lattice?
2. What is the *final* position of the result class (i.e., the view) in the class hierarchy?
3. How are types and extents derived for views defined by composite queries (or, equivalently, views defined over views) and can they be updated?

Typing. For the type changing operators project and extend (as well as union, intersection) we have defined the result type and whether it is a sub- or supertype of the input type. However, this result type need not be an *immediate* sub- or supertype. For example, consider the view defined by the following expression (see the left part of Fig. 7):

define view *EmpV* **as project** [*name, hiredate*] (*EmployeeC*);

Result types have to be positioned in the type lattice consistent with the other existing types. This may lead to the new type being placed further up or down the hierarchy. In the example, *EmpV* moves up the lattice beyond *PublicEmpV*. Also, it may be necessary to introduce several other new types as intermediate nodes in the type hierarchy. For example, consider the view *V* defined by projecting the *name*, *soc#* and *salary* of *EmployeeC* (cf. [GTC$^+$90, MS89]). Here, in order to get function inheritance right, an additional (anonymous) type is 'introduced' (see Fig. 7).[7]

The typing problem and the reorganization of the type lattice is nevertheless rather straightforward. The type lattice is actually (a part of) the powerset lattice over the set of all functions in the global schema. Complete algorithms for insertion into a type lattice, including complexity analyses, are given in [MS89], for instance.

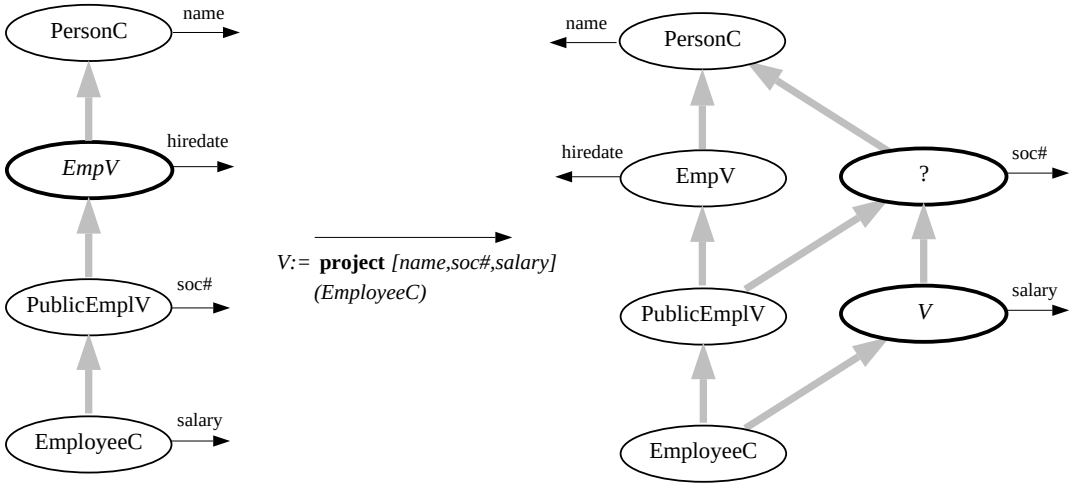

Figure 7. Absolute typing of projection.

Classification. Operations affecting the extent, especially selections, pose another problem. In order to position the result extent correctly in the class (i.e., subset) hierarchy, we have to decide class predicate implication: class *S* is subclass of *C* if *C*'s predicate implies that of *S*.

It is well-known that predicate subsumption is undecidable in general. So, how can we succeed in positioning a view in the class lattice ?

There are two ways out: (i) Permit only decidable predicates. Research on knowledge representation languages in the KL-ONE family [BS85] shows that, even with quite strong restrictions on the form of allowed predicates, the problem remains undecidable [SS89, Neb90]. In our case, in the presence of computed functions and subqueries, the situation is even worse. (ii) Implement an incomplete decision procedure. This means to consider only those parts of the class predicates during the classification that are known to be decidable. The position determined by the incomplete classification procedure is guaranteed to be correct, but there may be cases where the view could have been placed further down the class lattice, if the complete class predicate had been used. We have chosen that second solution for our implementation. The simplest and straightforward solution would be not to classify at all, but then selections will always result in *direct* subclasses. Besides the less informative class hierarchy that would be presented to the user in this case, this has other (performance) penalties. In case of queries,

[7] Formally, since the type hierarchy, as defined by the user, is internally completed into a lattice, this 'new' type has already been present before.

a known subclass relationship can help to optimize: for example, an intersection between two classes can be reduced to a simpler expression with only one class, if one is known to be a subset of the other. Furthermore, consistent processing of update operations, for example, insertions into a class, is also faster the more accurate the class hierarchy is: insertion into a subclass automatically inserts into all superclasses. If the class hierarchy were a complete lattice, nothing more had to be checked. otherwise, we have to check the (instantiated) class predicates of all other classes too.

Composite queries. Views defined by composite queries are well-defined, since we can stepwise compute result types and extents (since the model is closed and the operators combine orthogonally). However, the final view extent and type may be rather "far" away from the input in the class and type lattice, respectively. Each operator introduces an intermediate class and type, so if we want to keep the final result "connected" to the input in the lattices, we have to insert these intermediate nodes too. One might ask, how are names to be given to these intermediate types and classes. First of all, they need not be named, since we would not want to use them in declarations (for types) or queries (for classes) by explicit names. Alternatively, we can easily add syntactic sugar to give explicit names to intermediate results in a query.

We illustrate the classification of composite queries by the following example that uses a well-known algebraic equivalence of the relational algebra:

define view *V1* **as select** [*age > 20*] (**project** [*age, name*] (*PersonC*));
define view *V2* **as project** [*age, name*] (**select** [*age > 20*] (*PersonC*));

V1 first generates an intermediate class with all person objects but an associated member type that is a supertype of *PersonT*. Then it results in the final class *V1* being a subclass of that intermediate with the same type, but only a subset of the extent. *V2* proceeds the other way round: first a subset of *extent(PersonC)* is created with an unchanged type. Subsequently we keep this intermediate extent for the result class *V2* but it has a supertype. As suggested by Figure 8, the two expressions are in fact equivalent.

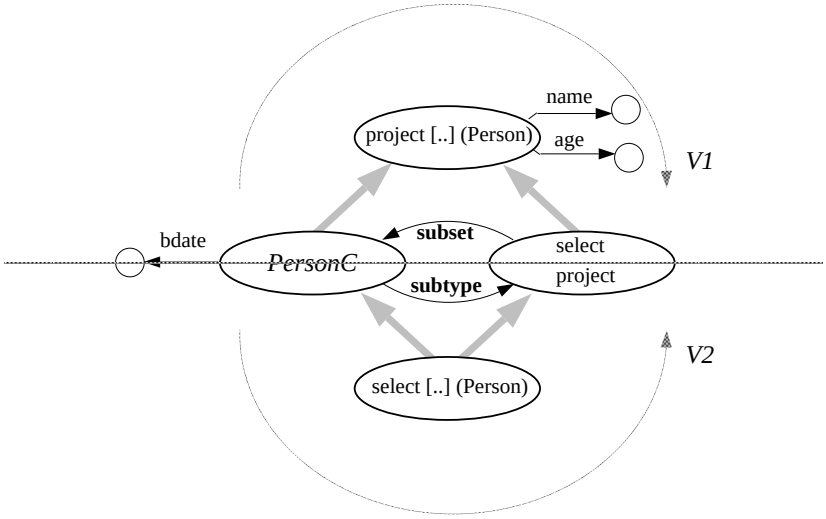

Figure 8. Classifying composite queries

This example also serves as a good argument for strictly separating subtype and subset relationships among classes (also see [Bee89]): If we use the subclass relationship that combines subtype and subset, then neither is *V1* a subclass of *PersonC* nor vice-versa. If, however, we separate the two concepts, two relationships hold between *V1* (or *V2*) and *PersonC* — but in opposite directions! The view class *V1* has a member_type, say *VT*, that is a supertype of *PersonT*, but the extent of *V1* is a subset of *extent(PersonC)*! Graphically, with two kinds of arrows, one for 'subset_of' and one for 'subtype_of', we have the "cyclic" situation shown in Figure 8.

Updating composite views can be regarded as a cascade of update propagations via the intermediate classes that are in fact views. That is, update operators applied to a composite view have the same effect as if they were applied to the intermediate view, created as predecessor while the stepwise computation of the composite view. Updating this intermediate view, however, also propagates to its predecessor, and so on until the propagation affects a base class. Therefore, updating composite views is restricted to those update operators that can be stepwise propagated.

4 Related Work

Recently, there have been other proposals for view support in ooDBMSs [AB91, HZ90, SJGP90]. They are fundamentally different in that our approach is the only one that uses the standard ways of defining views by nothing else than query language expressions. They either introduce special view definition features [AB91] or use other facilities of their systems (FUGUE [HZ90] uses type hierarchies for information hiding, POSTGRES [SJGP90] uses the rule system to simulate views).

The O_2 approach of [AB91] introduces special constructs for defining "virtual classes". They try to use the query language "as much as possible" in their view capability, however, some functionality is duplicated (for instance, union and intersection as "generalization and specialization abstractions"). They use a slightly modified version of the O_2 data model, dropping the distinction between (stored) attributes and (derived) methods. This is a first way of introducing derived information in the form of computed attribute values; this feature is included in our model from the beginning. In some respects, their views more general than ours, for instance they can hide functions (i.e., project away) not only in one class, but automatically in all subclasses. Our projection views, in contrast, hide the attribute only from the one class that is the view's base class, so we would have to derive views from all subclasses explicitly. What they called "behavioral abstraction", i.e., deriving a virtual class as the union of all classes whose type contains some given functions, is possible in our approach using queries that mix the meta-level and the object level. We allow such queries, however, they have not been discussed here [Tre91]. On the other hand, our approach is more general in that it allows all legal queries to serve as view definitions, without introducing any "imaginary" new objects. This includes joins, for which they have to maintain internal data to make sure that the new view objects get the same ID every time the view is queried. The most important difference, however, lies in the simplicity of our approach, since just use the query language for view definitions.

FUGUE [HZ88] allows multiple instantiation and each type of an object serves as one view onto it. More general view mechanisms are discussed in [HZ90]. There, "view" means what we called a subschema above, that is, a collection of base and derived classes, subject to some closure constraints. What corresponds to our views are additional types (and collections of objects of these new types) defined in their "views". The mechanism is more general, since each new type may be implemented differently over the existing types. this includes issues such as object preservation versus object generation, and whether one view object correspond to one or many base objects. While the underlying algebra of [SZ89b, SZ89a] does not provide object-preserving operator semantics, they can simulate object-preserving views by a type that implements object identity by using the OIds of some other (i.e., the base) objects. Updates on their views are possible, if the implementor of the derived types provides the corresponding methods (which may be implemented using methods of the base types). Their view mechanism basically builds on abstraction mechanisms of object-oriented programming rather than on a generic query language.

A similar approach can be found in [TYI88], where 'virtual' classes can be defined over Smalltalk classes. Virtual classes are defined by a (class) predicate that determines membership. The do not permit multiple inheritance, nor do they operators changing the type of objects. However, new methods can be defined for virtual classes.

The new POSTGRES rule system (PRS2, see [SJGP90]) can also be used to establish the functionality of views: Rules can define derived relations, and other rules can specify update semantics for these view

relations. As the rule language is more powerful than our query language, their views are more general than ours. On the other hand, we provide a standard way of view definitions together with a standard update semantics. Similar to their approach, our update semantics can be changed as appropriate. In PRS2 this is achieved by defining rules, in our approach by overriding some of the generic update operators and/or defining type-specific methods.

5 Conclusion

In this paper we presented a simple approach to views in object-oriented databases. Like in relational systems, queries can be used to define the extents of virtual classes, that is, collections of objects that are derived from some base classes. The presentation was informal and intended to carry over the ideas, not the exact details. A more formal treatment is contained in [LS91]. The central idea is that once the query language fulfills some basic requirements, its use for defining *updatable* views follows quite naturally. Nevertheless, to our knowledge, this is the only object-oriented query language with such a view capability. In contrast to other approaches, we can use arbitrary queries to define views, in exactly the same way as in relational query languages. In the sequel we summarize the properties of the query language that have been essential for the view definition capability and the view update semantics:

– object preservation,
– type/class separation,
– multiple instantiation and multiple class membership.

If some of these properties are not met by a language, we will see that our solutions will fail, partly or completely. Thus, the results we obtained are not bound to the COOL language, but to these properties.

Object preservation is the central concept. It is crucial for a straightforward view definition facility. Object preserving operator semantics means that the results of queries are (some of) the existing objects from the database. The other choices are: object-generating operators (results are objects, but newly generated ones) or tuple-generating operators (results are data, not objects). Examples of query languages with tuple-generating and/or object-generating semantics are [AK89, ASL89, SZ89b]. If queries just return data about objects, e.g. relations of tuple values: how could one try to apply updates to query results? Updating the result relation will not affect the objects. Such query semantics are useful for output purposes, for interfacing with value-oriented systems, or for restructuring operations [HS91], but not for updatable views. If a query language generates objects that are *new*, similar problems arise: we might apply methods, but how can they affect the original objects? One can, of course, play some implementation tricks, such as keeping the old OId as a "hidden" field, but this is a hack rather than a clear concept. To our knowledge, the object algebra of [SÖ90] and that of [HFW90] are the only other (algebraic) languages with object preserving operators. They also mention view support as one of the reasons for that semantics, but view updates have not yet been investigated. Some rule-based languages, such as F-logic [KL89], can specify object-preserving as well as object-generating operations, since there, OIds are available in the language.

The *type/class separation*, that can also be found in [Bee89, HFW90], is a consequence of object preservation: if both projections and selections are to preserve objects, and if composite select/project queries are permitted, we need this separation in order to connect the view class properly with the base class. The position of query results in the type and class hierarchies have to be less precise without this distinction (see [Kim89], where all query results are direct subclasses of "OBJECT"). Particularly, this separation of the difficult classification problem from typing allows for a strongly typed language with precise type inferencing. Furthermore, no algebra operation changes both, type and extent, except for union and intersection of two classes with differing types. So, the separation is a clarification of distinct concepts.

Multiple instantiation and multiple class membership are an immediate consequence of object preservation: since projection, for example, changes the type, all objects in the result "acquired" a new type.

If we consider objects in query results as being members of the result class as well as the input class(es), we can treat updates to query results in the same way as updates to stored classes and the updates propagate automatically. [Kim89] does not permit multiple class membership, another reason why view support is non-trivial in this model (views are mentioned in the paper as an important concept, but no solutions are presented).

Classification of query results has been mentioned in [Kim89] as a problem to be solved for closed query languages. But the positioning of result classes there (directly below "OBJECT") is not very helpful. With the separation into two hierarchies and with multiple instantiation and class membership, however, we can keep the position of the query result very close to the input class(es). None of the previous work we have seen offers similar solutions. The model described in [Day89] is quite close to our approach. View definition is also recognized as important. The presentation of the query language, however, does not elaborate on view definitions, updates via views are not investigated, neither is the placement of results in the lattices.

Dynamic reclassification during updates: More automatic classification functionality known from AI systems becomes necessary when we take into account, that objects can dynamically gain and loose types during their life time. In particular, update operations may affect the type and classification of objects; a change of an existing object can make it a member of a more specific class (because now it satisfies it's class predicate) or a more general one (if the class predicate of it's current class is violated by the update). Examples for this have been discussed above for selection views.

Acknowledgement The COCOON model has been developed jointly with Hans-Jörg Schek, to whom the authors are indebted for numerous discussions of the subject. He, Bin Jiang, and Christian Rich helped in improving an earlier version of this paper [SLT90]. An extended abstract of this paper appeared in [SS91].

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.

[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.

[AK89] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 159–173, Portland, June 1989. ACM, New York.

[ASL89] A.M. Alashqur, S.Y.W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 433–442, Amsterdam, August 1989.

[Bee89] C. Beeri. Formal models for object-oriented databases. In Kim et al. [KNN89], pages 370–395. 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.

[BTBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Static type-checking in object-oriented databases. *IEEE Data Engineering Bulletin*, 12(3):5–12, September 1989. Special Issue on Database Programming Languages.

[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.

[DMB+87] U. Dayal, F. Manola, A. Buchmann, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal. Simplifying complex objects: The PROBE approach to modelling and querying them. In

H.-J. Schek and G. Schlageter, editors, *Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications*, pages 17–37, Darmstadt, April 1987. IFB 136, Springer Verlag, Heidelberg.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derret, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, and M.A. Neimat. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–521, December 1988.

[GTC+90] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz, and X. Pintado. Class management for software communities. *Comm. ACM*, 33(9):90–103, September 1990.

[Heg90] S. J. Hegner. Foundations of canonical update support for closed database views. In S. Abiteboul and P.C. Kanellakis, editors, *ICDT '90 – Proc. Int'l. Conf. on Database Theory*, pages 422–436, Paris, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[HFW90] A. Heuer, J. Fuchs, and U. Wiebking. OSCAR: An object-oriented database system with a nested relational kernel. In *Proc. Int'l Conf. on Entity-Relationship Approach*, Lausanne, Switzerland, October 1990. North-Holland. to appear.

[HS91] A. Heuer and M.H. Scholl. Principles of object-oriented query languages. In H.-J. Appelrath, editor, *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications*, pages 178–197, Kaiserslautern, March 1991. IFB 270, Springer Verlag, Heidelberg.

[HZ88] S. Heiler and S.B. Zdonik. Views, data abstractions, and inheritance in the FUGUE data model. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*. LNCS 334, Springer Verlag, Heidelberg, September 1988.

[HZ90] S. Heiler and S.B. Zdonik. Object views: Extending the vision. In *Proc. IEEE Data Engineering Conf.*, pages 86–93, Los Angeles, February 1990.

[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.

[KL89] M. Kifer and G. Lausen. F-Logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 134–146, Portland, OR, May 1989. ACM.

[KNN89] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, Kyoto, December 1989. North-Holland.

[LS91] C. Laasch and M.H. Scholl. Generic update operations keeping object-oriented databases consistent. Submitted for publication, July 1991.

[MS89] M. Missikoff and M. Scholl. An algorithm for insertion into a lattice: Application to type classification. In W. Litwin and H.-J. Schek, editors, *Proc. 3rd Int'l Conf. on Foundations of Data Organisation and Algorithms (FODO)*, pages 64–82, Paris, June 1989. Springer Verlag.

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

[SJGP90] M.R. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In H. Garcia-Molina and H.V. Jagadish, editors, *Proc. ACM SIGMOD Conf. on Management of Data*, pages 281–290, Atlantic City, May 1990. ACM, New York.

[SLR+91] 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, 1991. In preparation.

[SLT90] M.H. Scholl, C. Laasch, and M. Tresch. Views in object-oriented databases. In *Proc. 2nd Int'l GI Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1990. Techn. Report 90/3, TU Clausthal, Germany.

[SLW88] A.P. Sheth, J.A. Larson, and E. Watkins. TAILOR, a tool for updating views. In J.W. Schmidt, S. Ceri, and M. Missikoff, editors, *Proc. Int'l Conf. on Advances in Database Technology (EDBT)*, pages 190–213, Venice, Italy, March 1988. LNCS 303, Springer Verlag.

[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.

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, jun 1986.

[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.

[SS91] M.H. Scholl and H.-J. Schek. Supporting views in object-oriented databases. *IEEE Database Engineering Bulletin*, 14(2):43–47, June 1991. Special Issue on Foundations of Object-Oriented Database Systems.

[SZ89a] G.M. Shaw and S.B. Zdonik. Object-oriented queries: Equivalence and optimization. In Kim et al. [KNN89], pages 264–278.

[SZ89b] G.M. Shaw and S.B. Zdonik. An object-oriented query algebra. *IEEE Data Engineering Bulletin*, 12(3):29–36, September 1989. Special Issue on Database Programming Languages.

[Tre91] M. Tresch. A framework for schema evolution by meta object manipulation. In *Proc. 3rd Int'l Workshop on Foundations of Models and Languages for Data and Objects*, Techn. Report, Institut für Informatik, TU Clausthal, Aigen, Austria, September 1991.

[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.