Derived Types and Subschemas: Towards Better Support for Logical Data Independence in Object-Oriented Data Models

Andreas Geppert
Stefan Scherrer
Klaus R. Dittrich

Technical Report 93.27
Institut für Informatik, Universität Zürich[1]

Abstract

Two concepts for the support of logical data independence in the context of the structurally object-oriented data model NO^2 (**N**ew **O**bject-**O**riented data model) are presented: *derived types* and *subschemas*. In general, object types can be derived in two ways: derived types can restrict (the value set of) an existing base type, or new atrtibutes can be added to the tuple value set of an existing object type. One major design issue of the concept is to preserve object identity: an instance of a derived type has the same object identity as the corresponding instance of the base type. This enables us to permit updates to derived types even in cases where (say) the relational view mechanism could not permit them. The second concept, subschemas, supports application-specific views of schemas and the straightforward identification of those parts affected by schema modifications.

1 Introduction and Motivation

NO^2 is the data model of the structurally object-oriented database system CoOMS (**Co**mbined **O**bject **M**anagement **S**ystem; [Dumm 91]) which is currently under implementation at SNI[2]. NO^2 offers a powerful data definition language (DDL) for the modelling of objects and relationships between them.

Nevertheless, an application may desire to view objects according to a structure different from the one defined in the underlying schema. More precisely,

- a user (in the sequel, let "user" include application programs as well) may not be interested in (parts of) the values of objects. In the university example (fig. 1; explained in more detail in section 2), for instance, a user may desire to have an overview of departments, but not of the concrete research groups and their assistants.

---

1. Authors' address: Institut fuer Informatik, Universitaet Zuerich, Winterthurerstr. 190, CH-8057 Zuerich, Switzerland. Email: {geppert,scherrer,dittrich}@ifi.unizh.ch

2. Siemens-Nixdorf Informationssysteme

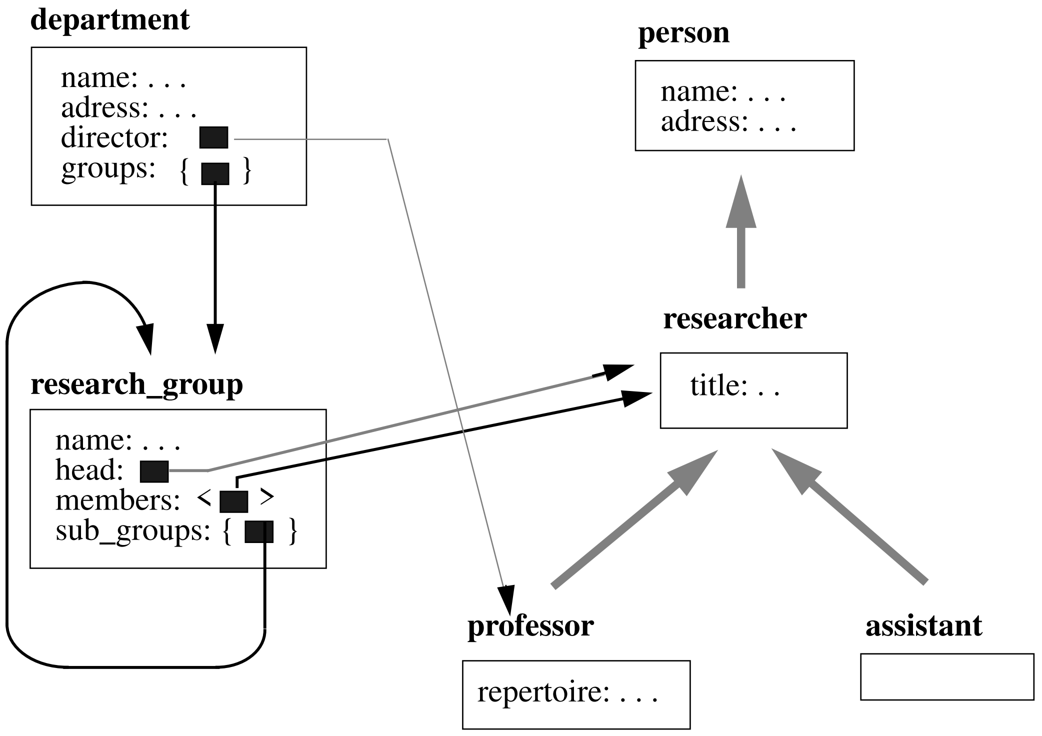

Figure 1 Sample NO^2 Schema

- a user may be forbidden to see specific information on objects. As a typical example, assume that we attach a *salary* attribute to *researchers*. Usually, only a few users will be allowed to modify or even read the *salaries* of *researchers*.
- a user may desire to see objects (and their values) conforming to a structure different from the one defined in the schema. As an example, assume a user who is responsible for the administration of rooms. Hence, she will be interested in the number of people that are working for a department, but not in details like names or titles.
- a user may desire to reuse queries. In this case, the result of the query has to be represented with the concepts of the data model, and can in turn be subject to further queries.

Obviously, it is not desired to define *new* object types and to force users to store such information in a redundant manner. Since the objects one wants to refer to are already present in a database (and their types are defined in a schema), a mechanism to *derive* types from existing ones is required. In other data models (e.g., the relational one), these derived types are called *views*[3]. In addition to derived types, a mechanism is required to restrict schemas to those object type definitions the user is essentially interested in (or allowed to use). In NO^2, all the

3. In the literature, the notion of “view” is used in two different ways: first, it describes the derivation of new types from old ones, and second, it refers to the restriction of schemas to user-specific subschemas. In order to not confuse terms, we do not use the term “view”. “Derived type” refers to the first meaning, and “subschema” to the second.

types (whether they represent “normal” or derived types) required by an application are collected into a *subschema*. In other words, an application has to be attached to a subschema and can only refer to those types that are included there.

In addition to the requirements for the functionality of the type derivation mechanism, the definition and the usage of derived types should allow for the following typical properties:

- Derivation of types should be possible at any time; i.e., even after the schema that contains the base type has been defined.
- For the specification of derivations, the query language of the system should be used. In other words, the structure of a derived type is defined by the result type of a query, and its extension is determined by the execution of the query. This requirement is sound, since typically query languages are closed, and it is reasonable, since users do not have to deal with a completely new language for the definition of derived types.
- It should be possible to use a derived type wherever a “normal” base type is permitted. In consequence, it should be possible to derive types from derived types, and it should be possible to modify, create, and delete instances of derived types.

In this paper, we present a mechanism for defining derived types in the context of NO^2. Actually, derivation of new types is possible in the following two ways:

1. *Restriction* refers to the omission of information (entire objects, or parts of objects),
2. *Extension* supports the derivation of types in that information is added to the old type (the so-called *base type*).

The remainder of this paper is organized as follows: the next section shortly introduces the NO^2 data model. Section 3 describes how NO^2 supports derived types, while section 4 addresses logical data independence in NO^2 and rules to be preserved by subschemas. In section 5, a comparison with related work is given, followed by a conclusion.

2 The NO^2 Data Model

In this section, we introduce the core concepts of NO^2, as far as necessary for understanding the derived type mechanism.

Data Definition Facilities

The most important concepts in NO^2 are *objects* and *object types*. An object is used to model a real-world entity and has an *object identifier* (OID) and a *value*. Object types define the structure of (the values of) their instances (the objects) in that a *value set* is attached to the object type. Additionally, each object type has a set of currently existing instances assigned (its *extension*). An OID uniquely identifies an object and distinguishes it from all other objects [Khoshafian 86]. OIDs are immutable over the lifetime of the corresponding object and cannot be reused after the object is deleted.

Furthermore, NO^2 supports a collection of value sets and a completely orthogonal set of *value set* constructors for the modeling of object values. Essentially, the following kinds of value sets are supported:

- basic value sets, including strings, integers, floating point numbers, characters, enumeration types, subrange types, and so on,
- complex value sets:
  - tuples,
  - sets,
  - lists, and
  - arrays.
- associations between objects:
  - general references,
  - subobject references.

Basic and complex value sets have the usual intuitive meaning. Complex value set constructors are completely orthogonal, i.e., they can be applied to any element or component type (be it a basic or another complex value set, or a reference type). General references represent unidirectional "pointers" to other objects. They are called "general" since they do not imply any specific semantics. On the other hand, subobject references are provided to represent *is-part-of* relationships between objects. In this case, referenced objects are termed *subobjects* and the source of the reference is called a *complex object*. The meaning of "complex object" is that objects are considered to be composed out of other objects, and possibly further values. The *is-part-of* relationship is also reflected by the semantics of DML-operations: for instance, complex objects are retrieved together with their subobjects, and deletion of complex objects can imply removal of subobjects (depending on the kind of subobject, see below). Two different kinds of integrity constraints can be assigned to subobjects: they can be either *sharable* or *exclusive,* and they can be defined as *independent* or *dependent* [Kim 89]. Sharable subobjects can be contained in more than one complex object, while exclusive ones can be a subobject of at most one complex object at a given point in time. Dependent objects can only exist as long as they are subobjects of another object, while independent subobjects can also exist of their own. One further type of consistency constraint can be used to define the uniqueness of attribute values within the objects of an extension. Finally, it can be specified for an object type that parts of the values of its instances have to be specified at insertion time (i.e., these values are REQUIRED and might not be NULL).

Finally, NO^2 supports *is_a* relationships and *inheritance* for object types. For an object type, one or more supertypes can be specified. In this case, each instance of the object type is regarded to be an instance of the supertype as well, and in each context where an instance of the supertype is required, an instance of the (sub)type is permitted as well (*substitutability*). Furthermore, the object type inherits the value set definition of its supertype(s). The subtype then can refine the value set definition in two ways: attributes can be added to tuple value sets (*extension*), and some basic value sets can be restricted (*restriction*, e.g., integers can be restricted to a subrange of integer).

Query Language and Data Manipulation

NO^2 provides for the declarative, set- (and list-) oriented query and data manipulation language *Quod*. This query language is similar to SQL, but obviously goes beyond it due to the more expressive structures of objects. *Quod* supports retrieval of objects and values (selection) including numerous operators for the various kinds of value sets. Operators for navigating within complex object structures are also allowed for. Furthermore, *Quod* provides built-in operators like *sort* or *group* and aggregation operators like *average*, *min*, *max*, and so on.

The semantics of queries is to retrieve objects, or (parts of) their values. Substitutability is obeyed, i.e., instances of subtypes are also retrieved whenever the extension of a supertypes is queried. The result of a query is always a set (or list) of objects or values, and no new objects are created as result of a query.

Finally, generic operations for the insertion, deletion, and modification of objects exist.

Schemas and Databases

Collections of semantically related object type definitions are grouped into *logical database schemas*. Each logical schema describes the structure of a part of the mini-world of interest for a set of applications, and has one or more logical databases attached to it. Hence, instances of an object type are stored in one of the corresponding logical databases. The *global schema* is the union of *logical database schemas*, and corresponding instances can be conceived as being stored in the *global database*.

However, given a set of such applications (or users) and a corresponding logical schema, it is likely that different applications or users desire to view the logical schema and a related database from different angles. Hence, each application has a *subschema* attached to it that describes how it desires to view the objects it is working with. A subschema is a collection of object type names and/or derived type names (see below) which are defined in the corresponding logical schema.

Fig. 1 shows a sample NO^2 schema in graph notation. In this figure, boxes represent object types. Solid arcs represent subobject references, dashed arrows general references, and bold dashed arrows *is-a* relationships. Set brackets refer to set value sets, while angle brackets are used for lists. Furthermore, fig. 2 shows the structure of and relationships between schemas and databases.

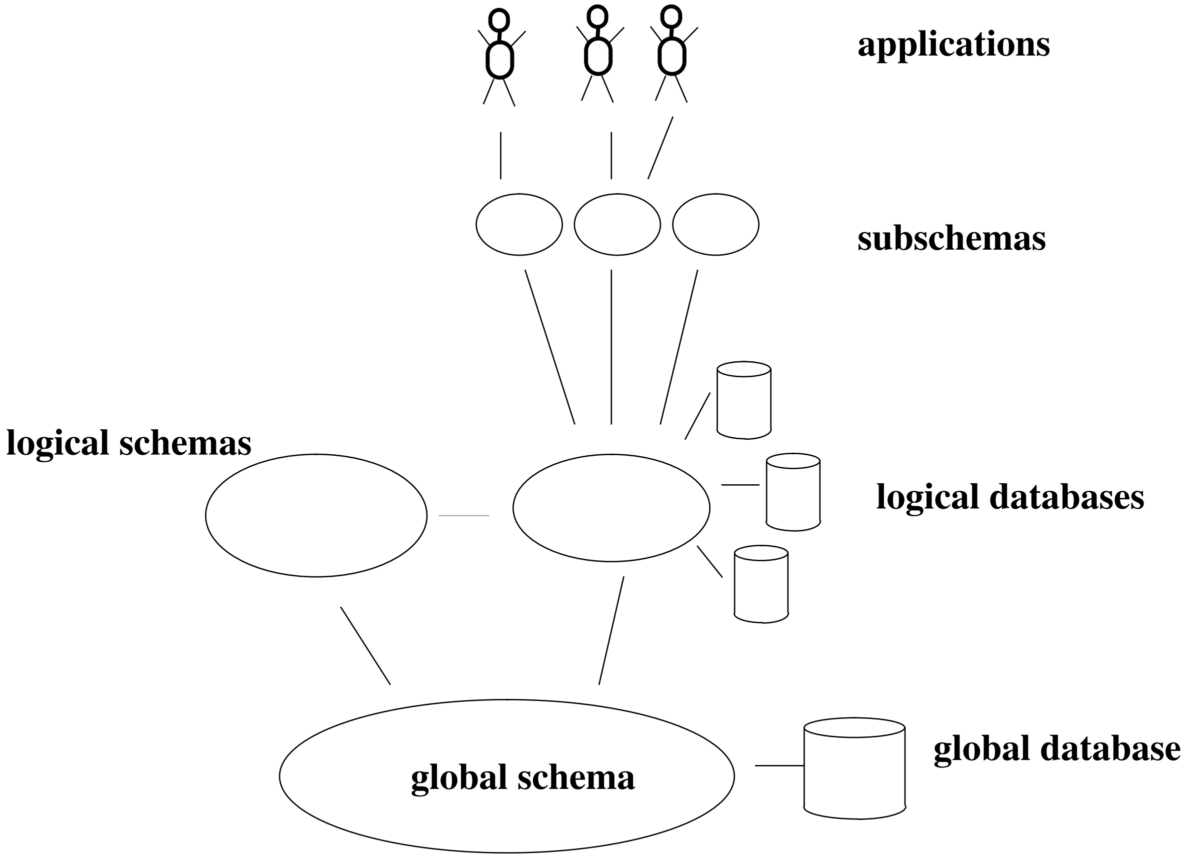

Figure 2 Structure of Schemas and Databases in NO^2

3 The NO^2 Approach for Derived Types

As stated above, we distinguish between two kinds of deriving types: restriction and extension. We motivate this distinction first and then elaborate on each of these cases separately.

3.1 Motivation

Due to the simple structures provided by the relational model and normalization during database design, real-world entities are often represented by multiple tuples in distinct relations. Hence, one purpose of the relational view mechanism is to collect all the information that describes one real-world entity. This has its price in that modification of views is often no longer possible since the affected tuples in the base relations cannot be uniquely identified. Another reason why (relational) views can sometimes not be modified is that attributes can be defined by the use of aggregation operations like *card*, *sum*, or *avg*. If such an attribute has to be modified, no unique translation into modification operations on the base relations can be found.

We claim that in a data model like NO^2 there is a lesser need to collect fragmented information by means of derived types. Actually, our approach (legally) assumes that relationships between objects are already defined in the schema by means of general or subobject references. One might argue that (say) SQL supports complete freedom in combining base types (relations) by means of joins. However, many of those will typically make no sense, and therefore need not to be supported. For the meaningful cases, however, our approach allows explicit sharing by means of general or subobject references.

In consequence, to a large extent types will be derived from *one* base type. To that end, the "restriction" form supports the definition of *single* derived types and extensions, but not of combinations of those. The advantage of this restricted kind of derived types is that they can be updated under simple conditions. Next, new types can also be derived in that base types are extended. Even in this case updates are possible, though the rules to be obeyed for them are more subtle. Finally, albeit it will be required less frequently, combination of types to a derived type is possible. In this case, however, we still require one base type, which then can be enriched with information from others. In other words, join-like combinations of types/extensions is a special case of extension.

3.2 Restriction

The first mechanism to derive a type from a base type is *restriction*. More precisely, the value set of an object type can be restricted by means of projection and/or selection. Essentially, projection means to restrict the value set of an object type. In principle, we could allow any projection operation supported by *Quod*. However, we feel that projection mainly makes sense for tuples only, but not for sets, lists, or arrays. Hence, derivation of types then means to specify the attributes that shall be contained in the value set of the derived type. As an example, assume the object type *employee* as follows:

```
DEFINE OBJECT TYPE employee =
        TUPLE (  name: REQUIRED string,
                 address: address_vs,
                 salary: integer).
```

Then, a derived type without the *salary* attribute would be defined as follows:

```
DEFINE DERIVED TYPE employee-2
        FROM emp IN employee
        AS
        SELECT [ name: emp.name, address: emp.address ]
```

Note that this derived type can be updated without any problems. The semantics of the derivation mechanism is that any instance of the (derived) type *employee-2* is derived from (exactly) one instance of *employee*, and that both have the same object identity. Hence, if an instance of employee-2 is to be updated, it is uniquely defined to which instance of *employee* this modification has to be applied. Additionally, we could create instances of *employee-2* and propagate the insertion to *employee*. An insertion would not be allowed, however, if the derived type would not possess the attribute *name*, since this attribute is required. Likewise, insertion is not permitted if at least one *unique* attribute of the base type has been omitted in the value set of the derived type.

Apart from projection, selection is another way to derive a type from a base type. Analogously to projection, *Quod* selection predicates can be used to specify subsets or sublists that should be visible in the derived type. In this case, selection does not make sense for tuples, but does make sense for sets and lists.

First of all, selection can be used to constrain an extension. As an example, assume that only those employees shall be visible that do not earn more than $100.000. This definition would look like as follows:

```
DEFINE DERIVED TYPE employee-3
        FROM emp IN employee
        AS
        SELECT *
        WHERE emp.salary <= 100.000
```

The result consists of a (derived) type with the same value set as *employee*, while the extension of *employee-3* is a subset of the extension of *employee*. Note that updates to instances of this derived type are possible, given that the *salary* attribute satisfies the specified predicate.

Likewise, it is possible to restrict a set or list *value* for each instance of a derived type. For instance, we want to derive a type from *research_group* such that the group contains only those *members* that hold a *PhD*:

```
DEFINE DERIVED TYPE brain-trust
        FROM rg IN research_group
        AS
        SELECT [  name: rg.name,
                  head: rg.head,
                  docs: (  SELECT m
                           FROM m IN rg.members
                           WHERE m.title = PhD)
                  ]
```

The value set of *brain-trust* consists of three attributes, one of which is renamed and redefined. The extension of *brain-trust* is the same as that for *research_group*, while instances of both extensions can differ in the *docs* and *members* attributes in that the WHERE-clause restricts the set of *members*. Instances of this derived type can be updated as well, as long as the selection predicate holds for subobjects.

3.3 Extension

In addition to the simple derived types that are based on restriction, derivation in the sense of *extension* is also possible. Similar to extension in NO^2 (base) object type definitions, derived types can only be extended in that new attributes are added to existing tuple value sets, while extension for other kinds of value sets does not make sense. As another constraint, the value of these attributes has to be determined by a query or a partial query. In the first case, a new attribute can refer to other extensions than the extension of the base type, while in the second case it is possible to restructure or recombine values of other attributes of the same object type. Hence, using extension makes it possible to restructure or compress information (e.g., by means of aggregation operations like *card*) and to assign it to new attributes. Addition of attributes is possible on any level of a nested tuple value set.

As an example, assume that we want to attach the number of members to each *research_-group*:

```
DEFINE DERIVED TYPE research_group2
        FROM r IN research_group
        AS
        SELECT *
        ADD member_cnt: card(r    .members)
```

Note that in the last line of the definition of *research_group2,* we refer to another attribute of the same object type (*members*). Hence, it is not always necessary to specify a complete query.

Like in the case of restriction, identity of objects is preserved when extension is applied. Hence, in principle types derived by means of extension can be updated. Obviously, the value of *added* attributes cannot be specified at insertion time or modified at update time, since they represent computed values.

3.4 Derived Types and Object Structures

Due to the support of object structures in NO^2, it is a problem how these structures have to change if either the referencing type or the referenced type is derived. The first case is rather simple: if a type whose instances reference other objects is derived, the “old” references can be included in the value set of the derived type or not. For instance, in the definition of *brain_-trust* the *head* of the group is an attribute of the instances of the derived type as well as of the base type.

The second case is slightly more subtle. Assume that a type is derived and that instances of other types also have to reference the instances of the derived type. For instance, let a derived type be defined for protection purposes (e.g., *employee-2*). However, other object types are likely to reference the base type (*employee*). Hence, how can references to the base type be substituted by references to the derived type? To that end, the definition of derived types has to propagate to the referencing types. The effect of that propagation is that attributes containing references to instances of the base type(s) are omitted and new attributes are defined that represent the references to instances of the derived type.

In the example, let *research_group2* be a derived type as defined above. Additionally, assume that (in a corresponding subschema) only the instances of this derived type have to be referenced by instances of the type *department.* Hence, a derived type *department2* has to be defined as follows:

```
DEFINE   DERIVED TYPE department2
         FROM d IN department
         AS
         SELECT [ name: d.name,
                  address: d.address,
                  director: d.director]
         ADD groups: (SELECT g
                      FROM g IN research_group2
                      WHERE g IN d.groups)
```

Note the peculiarity in this example: the variable *g* is required to be of type *research_group2* (as declared in the *FROM* clause), but is compared to instances of type *research_group*. Hence, definitions like this assume that objects are compared with respect to their identity (but not with respect to their immediate type).

Two further questions in the context of object structures and derived types are whether the latter can inherit from other types, and whether any type can inherit from a derived type. In the first case, derived types automatically inherit according to the definition of their base type. There is no possibility, however, to specify that a derived type should additionally inherit from other types. Finally, referring to the second question, types cannot inherit from derived types.

3.5 Operations on Instances of Derived Types

In this section, we summarize the rules concerning permitted updates of instances of derived types. We investigate the rules for the three generic update operations *insertion*, *deletion*, and *modification*, respectively. In the sequel, we refer to *excluded attributes* as those that are not contained in the value set of the derived type. For the sake of completeness, we also include retrievals.

3.5.1 Retrieval

Retrieval of instances of derived types or of entire derived extensions is possible by means of *Quod* queries as if the derived type were a "normal" NO^2 type. Hence, no specific rules have to be obeyed in this case. As an example, assume that we want to query the *names* and the number of *members* of all instances of the *research_group2* derived type:

```
SELECT   [ name: r.name, cnt: r.member_cnt]
FROM     r in research_group2
```

Hence, queries against derived types are not different from those against base types.

3.5.2 Insertion

In principle, object creation is no problem. Unlike in the relational model, we have exactly one base type and hence one base extension the object has to be inserted into. However, the following rules have to hold in order to permit insertion:

- no *unique* attribute may be excluded from the derived type,
- no *required* attribute may be excluded from the derived type.

Obviously, these rules have to be preserved transitively, i.e., in the case where the base type is also a derived type, the latter has to fulfill the rules, too.

Additionally, further insertions may be prohibited depending on the semantics of the *dependent* clause: if this clause refers not only to cascading deletions, but also requires that dependent objects always have to be subobjects, then the following rule applies as well:

- no attribute containing dependent subobjects as values may be excluded.

Finally, the following rules are noteworthy with respect to the values to be specified at insertion:

- at insertion time, computed attribute values (e.g., as results of queries or partial queries) cannot be specified for a new object.

The last rule addresses the problem that a created object of a derived type does not fulfill some restriction (i.e., selection predicate) specified for the derived type. In this case, a created object would not be an instance of the extension of the derived type, which contradicts the insertion request.

- an insertion is illegal if the specified object value does not preserve the restrictions specified by the derived type.

As an example, assume a derived type *employee2* for such employees earning less than $100.000 per year. In this case, insertion of an object with a *salary* attribute value with more than this amount would not be allowed.

3.5.3 Deletion

For deletions, the overall condition is again that no unambiguity or integrity violations may occur. Especially, the following rule has to be obeyed:

- dependent subobjects may not be excluded from the definition of the derived type.

In this case, the application would not be aware of the cascading deletion (to dependent subobjects), and hence might not realize that more objects as desired are essentially deleted.

3.5.4 Modification

Modification refers to the change of (parts of) the value of an instance of a derived type. This operation is prohibited if one of the following rules is violated:

- the new value of *unique* attributes has to be unique amongst the instances of both, the derived type as well as the base type.
- derived/computed attributes or values cannot be modified,
- new values of modified instances have to satisfy selection predicates defined for the type.

4 Derived Types, Subschemas, and Logical Data Independence

As already mentioned above, some approaches use the concept of "view" to represent application-specific collections of types and extensions. This is supported in NO^2 by the notion of *subschema.* A subschema is a subset of the type definitions of a logical schema. In other words, both, base types as well as derived types are defined in the logical schema, and types of interest for a specific application have simply to be declared (i.e., mentioned) in the corresponding subschema.

In consequence, NO^2 supports a high degree of logical data independence in two ways:

1. A subschema acts as a "window" on the logical schema. Hence, if a logical schema is modified, we can easily determine those applications that are potentially affected by the modification. Each application having a subschema that does not use the modified logical schema parts cannot be affected. Without a subschema concept, however, knowledge on schema parts used by an application would not be available, and possibly affected applications could not be determined.
2. Derived types increase the degree of data independence of applications. Given that an application might be affected by a schema modification, the definition of derived types has to change only if the definition refers to these parts of the base type that are changed by the schema modification. In other cases, the definition of the derived type remains unchanged. Otherwise, if the definition of a derived type has to change, the intention of the derived type can still remain the same (i.e., the structure of its instances). Hence, the application itself will not be affected.

In order to support logical data independence properly, two conditions are required to hold for any subschema:

- a subschema has to be well-defined,
- a subschema has to be closed.

A subschema is well-defined if all the object types it contains are so. A derived type is well-defined if it is defined (i.e., contained) in the corresponding logical schema, if it is derived (directly or indirectly) from an object type in the logical schema, and if the definition is syntactically and semantically correct.

A subschema is closed if any type that is referenced by a type (of the subschema) is also contained in the subschema. Obviously, this rule applies transitively. In other words, if the value set of an object type (directly or indirectly) contains general or subobject references to another type, this type has to be in the subschema as well. However, if those references are omitted in the definition of a derived type, the target object type need not be in the subschema (as long as it is not referenced by other types of the subschema).

5 Related Work

Since there is a consensus that object-oriented DBMS should provide some kind of derived types, but unfortunately no standard object model exists yet, it is not surprising at all that numerous derived type/subschema mechanisms have emerged in the recent past. The approaches surveyed in the following differ in various aspects which are shortly described and presented in a comparison table (Table 1).

The aspect to start with is *schema modularization*. Offering derived types without concepts to modularize possibly very large schemas tends to be of limited use. A related aspect is how derived types are integrated into the schema; that is whether they are located in the inheritance graph as other object types or whether they represent a third schema dimension (besides composition and inheritance).

With respect to derived types, the probably most important aspect is whether the facilities for the *population* of derived types *preserves object identity*, or whether new object identifiers are assigned whenever the derived type is accessed. The latter significantly complicates *updatability* of derived types since whenever it is accessed, the users expect the same identifiers for the same values as in the previous accesses. In the former case, both, the propagation of changes from object types to derived types and vice versa is easily possible. Hence, in order to support updatability of derived types, an identity preserving (subset of the) query language is almost mandatory for the underlying data model in order to properly *populate* derived types.

The *structural concepts of the data model* represent the next important aspect. The more powerful the concepts of the data model are, the less likely the user has to perform expensive join operations in order to assemble complex real-world entities from simpler stored objects. We claim that if associations between objects are missing *and* the data model provides appropriate association concepts (as this is the case in NO^2), it is more reasonable to evolve the schema using the *schema modification* facilities (requiring adequate data model concepts) rather than defining derived types whose underlying queries consist of some kinds of joins. This claim bases on the assumption that the computation of projections (hiding superfluous associations) is much cheaper than computing joins.

A *set of orthogonal value-set constructors* provides even more chances that joins are required in fewer cases. Moreover, such constructors (like *set*, *list*, *tuple*) etc. give chance to express more semantics by derived type definitions in that existing value set definitions can be orthogonally refined by means of *intensional restrictions* (*projections* and *selections*). Note that in contrast to schema evolution, the old schema state does not disappear. Besides intensional *re-*

striction of existing object types, their *intensional extension* is an important issue as well. Depending on the data model, behavioral (adding new methods) and/or structural (adding computed attributes) extension is provided.

Derived type compositionality refers to the ability to base one derived type on another one (or others).

Other approaches not easily fitting into our comparison table include:

- [Stonebraker 90], where a production rule mechanism is used to simulate views,
- [Dayal 89], where issues for views in the OODAPLEX model are sketched,
- [Wiederhold 86], where object-oriented views are investigated in order to interface object-oriented programming systems to relational database systems.

6 Conclusion

In this paper, we have presented the concepts of *derived type* and *subschema* in the context of the NO^2 structurally object-oriented data model. Both concepts support a high degree of logical data independence (apart from other purposes —like protection— derived types may be used for).

Derivation of types from existing ones is possible through restriction and/or extension. It turns out that the restriction case is more powerful than corresponding concepts in other approaches. For instance, it is not only supported to restrict extensions (of base types), but it is also easily possible to restrict complex values for each instance of the derived type (in relational terminology, this refers to constructing “views in views”).

Due to preservation of object identity, instances of derived types can be updated under rather simple conditions in the restriction case. Even in the extension case, it is still possible to identify corresponding instances in the extension of the base type, and hence updates can be performed in this case as well. Our concept supports combination of types/extensions not very well (i.e., deriving types through joins). Nevertheless, we have argued that this is not an important feature in object-oriented databases anyway, and therefore this lack is no true disadvantage.

Subschemas are the other important concept to support data independence, since in this way we are able to easily identify applications that are affected by schema modifications.

As for future work, the inclusion of behavior is an interesting open issue.

	NO2	[Abiteboul 91]	[Bertino 92]	[Heiler 90]	[Runden-steiner 92]	[Scholl 91]	[Shilling 89]	[Tanaka 88]
schema modularization; derived types	global, logical, and subschemas; derived types	global schema, views; virtual classes	global schema; views	global schema, views; abstract data types	global schemas, view schemas; virtual classes	global schema, subschemas; views	no schemas since only an extension of C++; view classes	global schema, virtual schemas; virtual classes
integration into inheritance graph	no	yes	no	no inheritance	yes	yes	no	no
identity preservation	yes	yes	DBA defined	yes, if the view definition obeys some criteria	yes	yes	yes	unclear
population	query returning set of existing objects	query returning set of values	query returning set of existing oids or values	query returning a set of existing oids or a set of tuples with existing oids	query returning existing set of oid	query returning set of existing oids	simple object creation, no queries	Smalltalk-80 predicates
updatability	yes	yes	future work	yes, but DBA-defined operations required for views on multiple base classes)	unclear	yes	yes	not an explicit issue
schema evolution	yes [Scherrer 93]	yes	no	no	no	yes	no	no
intensional restriction/ extension	orthogonal set/ list selection, tuple projection, computed attributes	specialization, generalization, behavioural generalization, parameterized virtual classes, computed attributes	tuple project, adding attributes, adding behavior	hiding of attributes and behavior, adding of attributes and behavior	similar to [Scholl 91]	union (by generalization), intersection (by multiple inheritance), difference (superfluous), extend (subclassing), project (superclassing)	projection of attributes, marking attributes as read-only, multiple copies of attribute values	joins
composition	yes	yes	yes	yes	yes	yes	yes	yes

Table 1 Related Work

7 References

Abiteboul 91 S. Abiteboul, A. Bonner: *Objects and Views*. Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Denver, Colorado, May 1991.

Bertino 92 E. Bertino: *A View Mechanism for Object-Oriented Databases*. Proc. Intl. Conf. on Extending Database Technology, Vienna, Austria, March 1992.

Dittrich 91 K.R. Dittrich: *Object-Oriented Database Systems: The Notions an the Issues*. In K.R. Dittrich, U. Dayal, A.P. Buchmann (eds.): On Object-Oriented Database Systems, Springer 1991.

Dayal 89 U. Dayal: *Queries and Views in an Object-Oriented Data Model*. Proc. of the 2nd Intl. Workshop on Database Programming Languages, 1989.

Dumm 91 T. Dumm, T. Gorchs, M. Watzek: *The CoOMS Persistent Object Server. Requirement Specification, Functionality, and Architecture*. Project Report ITHACA.SNI.91.X.5#2, Siemens Nixdorf Informationssysteme, Berlin 1991.

Geppert 93 A. Geppert, K.R. Dittrich, V. Goebel, S. Scherrer: *The NO2 Data Model*. Technical Report 93.09, Institut für Informatik, Universität Zürich, 1993.

Heiler 90 S. Heiler, S. B. Zdonik: *Object Views: Extending the Vision*. Proc. 6th Intl. IEEE Conf. on Data Engineering, Los Angeles, California, February 1990.

Khoshafian 86 S. Khoshafian, G. Copeland: *Object Identity*. Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, September/October 1986. .

Kim 89 W. Kim, E. Bertino, J.F. Garza: *Composite Objects Revisited*. Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Portland, Oregon, May/June 1989.

Rundensteiner 92 E.A. Rundensteiner: *MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases*. Proc. of the 18th VLDB Conference, Vancouver, British Columbia, Canada.

Scherrer 93 S. Scherrer, A. Geppert, K.R. Dittrich: *Schema Evolution in NO^2*. Technical Report 93.12, Institut für Informatik, Universität Zürich, April 1993.

Scholl 91 M. H. Scholl, Ch. Laasch, M. Tresch: *Updatable Views in Object-Oriented Databases*. Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases, Munich, FRG, December1991.

Shilling 89 J. Shilling, P. Sweeney: *Three Steps to Views: Extending the Object-Oriented Paradigm*. Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications, New Orleans, Louisiana, October 1989.

Stonebraker 90 M. Stonebraker: *On Rules, Procedures, Caching and Views in Data Base Systems*. Proc. ACM-SIGMOD Intl. Conf. on Management of Data, Atlantic City, New Jersey, USA, 1990.

Tanaka 88 K. Tanaka K., M. Yoshikawa, K. Ishihara: *Schema Virtualization in Object-Oriented Databases*. Proc. of the 4rd Intl. IEEE Conf. on Data Engineering, Los Angeles CA, USA, February 1988.

Wiederhold 86 G. Wiederhold: *Views, Objects and Databases*. IEEE Computer 19:3, 1986.