Institut für Informatik der Universität Zürich

Schema Evolution in NO^2

Stefan Scherrer
Andreas Geppert
Klaus R. Dittrich

Technical Report Nr. 93.12 April 93

Schema Evolution in NO^2

ITHACA.Unizh.93.X.5.#3

Stefan Scherrer
Andreas Geppert
Klaus R. Dittrich

Institut für Informatik
Universität Zürich

Abstract

This paper gives an overview of the schema evolution capabilities of the structurally object-oriented data model NO^2. The overview starts with a survey of the inherent integrity constraints of NO^2 (schema invariants) which have to be preserved by any schema change. A taxonomy of possible schema changes is given. For each possible change, a short description is given which illustrates the effects to parts of the schema, and, equally important, to possible databases corresponding to the respective schema. Moreover, different strategies for the realization of schema changes are presented.

1 Introduction

The structure of a database is defined by the schema which itself has been constructed using the concepts offered by the data model (in the context of this paper, the NO^2 structurally object-oriented data model [Dittrich 92] for the database management system CoOMS [Dumm 91]). Schemas are likely to undergo changes during the life-time of the database due to evolving application requirements; for example, value set definitions may be changed, subobject relationships may be modified, and the inheritance graph may have to be restructured. However, today only few object-oriented systems offer general means to change the schema dynamically ([Zicari 91], [Banerjee 87]). This is mainly due to the complex implications which a schema change can cause.

Generally, three kinds of implications of schema changes exist:

- *Modification of programs that use the old schema*: Changes to object type definitions can influence (invalidate) existing applications using the old schema in various ways. The effects on applications in turn strongly depend on how the data model supports logical data independence. However, the investigation of schema change effects to existing applications is considered a non-issue in this paper mainly due to two reasons. First, NO^2 supports application subschemas whose use should significantly damp the effects of schema changes. Second, we feel that schema changes not only imply application changes, but schema modification in turn is required by evolving applications. In other words, schema evolution is a requirement *to* CoOMS from the programming environment (e.g., persistent CooL) rather than schema modification implies evolving applications.
- *Modification of existing instances which corresponded to the prior schema*: Of course, the life-time of a schema is at least as long as the life-time of the data in the database. Hence, the approach to unload the database, to define a new schema and to reload the old data using a hand-crafted conversion program becomes the more expensive the older and larger the data are. Schema evolution tackles this problem by providing for incremental schema changes, hence avoiding expensive unloads and reloads by means of modifying only those instances which are really impacted by the particular schema changes. Following an incremental approach, different realization strategies exist. First, changes may be propagated to the impacted instances immediately upon occurrence of a schema change. Second, changes may be executed upon the next access to affected instances. The latter strategy offers “incrementalism” of a higher order, but the object access time may become rather indeterministic. Another strategy is schema versioning. The pros and cons of each strategy are discussed in more detail in section 5.
- *Effects to the remainder of the schema*: The change of a particular part of the schema may have severe effects to the remainder of the schema. These effects have to be precisely defined; e.g., the DBA should be (made) aware very well of the effects of throwing away (say) an object type which is both, root of an inheritance graph and a subobject graph. To this end, schema modifications must not violate the inherent integrity constraints; i.e., constraints which define the correctness of a particular schema.

In previous work, different kinds of schema changes has been classified by means of taxonomies [Banerjee 87]. The following classes of schema changes typically occur[1]).

- *Changes of attributes*: Attributes can be subject to creation, deletion, and change of their names and change of their value set.
- *Changes of methods*: Methods can be created, deleted, and their implementation

1. Note, that they are explained using NO^2 terminology at least where adequate.

can be changed. Note that these kinds of changes have no correspondences in NO^2 since it does not address behavioral issues at all.

- *Changes of parts of the inheritance graph*: The inheritance graph represents inheritance relationships between object types. This relationship can change in that the list of super object types of a given object type grows or shrinks, in that the conflict resolution in the context of conflicting attributes/methods is changed.
- *Changes of object types themselves*: Object types can be created, deleted and their names can be subject to change.

This kind of taxonomy has proven particularly useful in discussing schema evolution in the context of behaviorally object-oriented data models. However, as the discussion of the different kinds of changes might already have shown, this kind of classification is of limited use for the structurally object-oriented data model NO^2 which in turn emphasizes the orthogonal value set construction system and does not support the modeling of behavior. In the following, we will therefore present a taxonomy which is particularly tailored to the specifics of the NO^2 data model, but nevertheless conforms to the typical taxonomy where possible. Furthermore, the NO^2 taxonomy goes one step beyond in that it addresses subschema definitions and logical database definitions as well[2].

The remainder of this paper is structured as follows. Section 2 repeats the aspects of the NO^2 data model and defines requirements for a schema evolution concept. Section 3 surveys the inherent integrity constraints of NO^2 (see also [Dittrich 92]); that is, the invariants that have to be obeyed by any schema modification. Section 4 presents the NO^2 schema evolution taxonomy using state diagram techniques and describes the potential implications of each kind of change. In section 5, different strategies for the different realization aspects are discussed. Section 6 evaluates the concept with respect to the requirements.

2 Preliminaries

In this section, we rush over the concepts of NO^2, as far as necessary to achieve a basis of the following discussion on schema evolution, and in a next step, we list requirements to be addressed by the schema evolution concept.

In a nutshell, NO^2 can be summarized as follows (see also Figure 1.). It contains a collection of basic value sets like string, real, integer, enumeration, and subrange. Composite value sets are defined by constructors for tuples, sets, lists, and arrays. Object types are defined on the basis of value sets. Complex object structures can be defined either by means of including objects as components in other objects, or by means of general references. Object types may be organized in a specialization/

2. Currently, the DDL supports the *creation* of value sets, object types and the different kinds of schemas only, but not their *deletion* and *modification.*

generalization hierarchy which allows for (multiple) inheritance of (structural) properties. Logical database schemas are collections of object type and value set definitions. Subschemas describe the interface of applications to logical databases by means of an enumeration of object type names and derived type definitions.

The following requirements shall be addressed by the schema evolution concept:

- A list of inherent integrity constraints shall be developed whose enforcement is required by the realization of the schema change operations (see also [Dittrich 92])
- The concept shall give a complete summary of the possible schema evolution operations in NO^2. An indication for completeness can be obtained by analyzing the existing grammar of the DDL [Dittrich 92]. Each production which contains the terminal symbol *name* is a candidate for a simple schema change operation. Additional operations can be found by searching for optional productions ("[*non-terminal*]"); i.e., denoting that *non-terminal* is allowed to be present or not. If it is actually not present, it could be introduced by means of an appropriate schema change operation. The last group of interest are repetition groups ("{*non-terminal*}"), denoting that an arbitrary number of items of *non-terminal* can be present. Usually, such constructs are indicators for potential schema evolution operation.
- For each of these schema change operations, the concept has to define the effects on possible instances as well as on the remainder of the schema. Wherever appropriate, the NO^2 algebra expressions [Geppert 92b] shall be used to express changes. Furthermore, possible effects to existing derived types have to be denoted as well.
- Different strategies shall be evaluated both for the realization of the schema operations themselves and the propagation of schema change effects to the objects populating the database whose schema is subject to change.

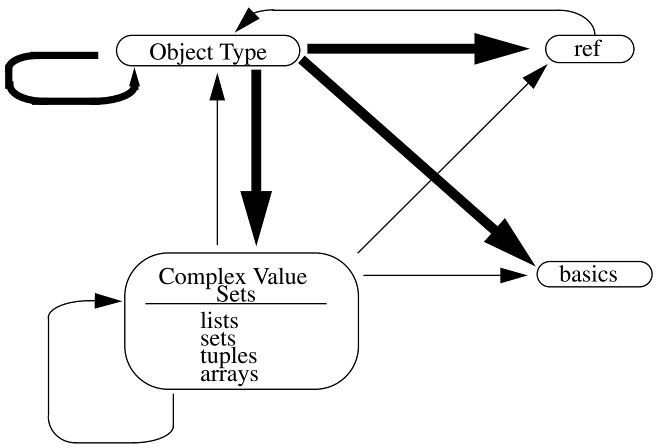

Figure 1. Value set and object type constructors

While the first requirement is addressed in section 3, section 4 is devoted to the second and third requirement, and the last one is dealt with in section 5.

By the way, note that the definition of appropriate DDL syntax to cover the schema changes is not yet a requirement since we feel it is reasonable to find a consensus on the operations to be supported before syntax becomes an issue.

3 Inherent Integrity Constraints

The following subsections enumerate the inherent integrity constraints of the NO^2 data model, which must not be violated by any schema. We assume that we start with an empty schema whose integrity holds trivially. Then, a schema change denotes a transition from one schema state to the next consistent one. Depending on the realization strategy (check section 5), transitions may be either groups of schema operations (schema transactions) or single schema operations. In both cases, the transition leads to a consistent state which indeed can be the old state as well (namely in the case where an integrity constraint violation has been detected at the end of the transition).

3.1 Inheritance Graph Acyclicity

Since NO^2 provides multiple inheritance, the set of object types of a schema may be represented as a set of directed acyclic graphs (DAG), where the object types are nodes and edges represent inheritance relationships. Hence, whenever the inheritance graph is modified, acyclicity has to be preserved.

3.2 Name Uniqueness

The overall name space of a schema is subdivided into the following subspaces:

- The names of logical schemas,
- the names of subschemas of a logical schema,
- the names of object types,
- the names of derived types,
- the names of value sets,
- the names of attributes in a tuple value set.

For each subspace, name uniqueness must not be violated by any schema state.

For example, in a new object type definition, if the tuple value set of the parent object type is extended with an attribute whose name already existed in the parent, name uniqueness would be violated (i.e., attribute overriding is not allowed).

3.3 Referential Integrity

A schema entity is usually composed of other schema entities; for example, a tuple value set definition may reference basic value sets or object type definitions in order to define the value set of its attributes. Inherent referential integrity of a schema state holds if all schema parts reference only schema parts which actually exist.

For example, when dropping an object type definition without special care, inherent referential integrity is violated if there exists a subschema definition containing the name of the dropped object type, or if at least one tuple value set exists which happens to have the name of the dropped object type as value set of one of its attributes (see also [Geppert 93], rules for subschemas and derived types).

3.4 Database-Schema Correspondence

Of course, the objects populating the extension of an object type have to correspond to the actual object type definition. Hence, whenever a new schema state occurs, the objects in the database have to reflect these changes; i.e. schema changes have to trigger proper database changes (change propagation).

For example, if an attribute of a tuple value set is dropped, the schema transits to a new schema state. If the change had only effects to the schema and not to all objects whose values are elements of the value set in question, then database schema correspondence would be violated. Thus, the system has to determine all those objects and to restructure the values of these objects.

4 Taxonomy

In this section, the possible schema changes are discussed and their effects to the possible instances and the remainder of the schema are analyzed. The set of operations is classified on top level by the different kinds of schema components, i.e. value sets, object/derived types, and (sub)schemas. The following subsections reflect this classification.

4.1 Value Sets

In this subsection, we present the modification applicable to existing value set definitions. The possible modifications to value set definitions are presented using the concept of transition tables. Given an old value set definition, the transition table indicates the possible schema changes. On the other hand, it also serves as a directory of the description of a particular conversion as well. The transition table has the following generic structure:

	basic-vs_j	$constructor_q(vs_j)$
basic-vs_i	simple conversion property modification	wrapping
$constructor_p(vs_i)$	unwrapping	wrapping unwrapping complex conversion property modification

Table 1. Generic Transition Tables for Value Sets

While the items in the title column (of Table 1.) denote the old kinds of schema entities, the title row represents the target of the schema change. *Simple conversion* ($i \neq j$) refers to the process of modifying a basic value set to another basic one (e.g., changing real to integer). *Wrapping* ($i = j$) refers to the process of applying another constructor to a given value set (e.g., changing integer to list(integer)). *Unwrapping* ($i = j$) denotes the process of removing a constructor from a value set (e.g., modify list(integer) to integer). *Complex conversion* ($p \neq q$, $i = j$) refers to the process of changing the top-level constructor of a value set (e.g., modifying set(integer) to list(integer)). A complex conversion, where $p = q$, refers to changing some parameters (e.g., changing the dimensions of an array value set, adding attributes to a tuple value set, and the like) while the element value set remains unchanged. *Property modification* ($i = j$, $p = q$) refers to the process of changing the name of a value set or changing the required implicit integrity constraint. While the former kinds of modifications change the structure of values, property modifications do not.

Cells of the value set transition table (Table 2.) in turn contain section numbers serving as an index to a more succinct description. The descriptions themselves contain the issues (impacts on remainder of the schema, impacts to existing instances, parameters needed for the conversion, etc.) related to the modification operation that is denoted by the position of the cell.

The aforementioned classification into simple conversion, wrapping, unwrapping, complex conversion is used to straddle smaller cells to larger ones which have similar properties. Moreover, the visual effect of cell straddling is amplified by the use of different shadings, where the most important shading pattern is the white one, which indicates that either no operation is available or that it does not make sense. The grouping of cells with different sizes and with different shades proves to reduce the description effort significantly.

	boolean	integer	string	real	longfield	enumeration(a, b,...)	subrange (a, b)	set(vs)	list(vs)	array (n, vs)	tuple(a: vs, b:vs...)	ref(ot_name)	dependency sharability ot_name
boolean	4.1.1	4.1.2						4.1.3	4.1.3	4.1.3	4.1.3		
integer	4.1.2	4.1.1	4.1.2										
string		4.1.2	4.1.1	4.1.2									
real			4.1.2	4.1.1	4.1.2								
longfield				4.1.2	4.1.1	4.1.2							
enum(a, b...)					4.1.2	4.1.1	4.1.2						
subrange(a, b)						4.1.2	4.1.1						
set(vs)	4.1.4							4.1.1 4.1.3 4.1.4 4.1.5.2	4.1.3 4.1.4 4.1.5.1	4.1.3 4.1.4 4.1.5.1	4.1.3 4.1.4 4.1.5.1	4.1.4	
array(n,vs)	4.1.4							4.1.3 4.1.4 4.1.5.1	4.1.1 4.1.3 4.1.4 4.1.5.2	4.1.3 4.1.4	4.1.3 4.1.4 4.1.5.1	4.1.4	
list(vs)	4.1.4							4.1.3 4.1.4 4.1.5.1		4.1.1 4.1.3 4.1.4 4.1.5.2	4.1.3 4.1.4 4.1.5.1	4.1.4	
tuple(a:vs, b:vs...)	4.1.4							4.1.3 4.1.4	4.1.3 4.1.4	4.1.3 4.1.4	4.1.1 4.1.3 4.1.4 4.1.5.2	4.1.4	
ref (ot_name)								4.1.3	4.1.3	4.1.3	4.1.3	4.1.1	4.1.5.1
dependency sharability ot_name												4.1.5.1	4.1.1 4.1.5.2

Table 2. Value set transition table

In the upper left, we have simple conversions (4.1.2) which are not subject to further investigation. Unwrappings (4.1.4) are shown in the lower left corner. The constraint here is that vs corresponds to the column name. For example, an allowed modification is to convert list(integer) into integer, but not list(integer) to real, since this would require a simple conversion from integer to real first and then applying an unwrapping. Therefore, the table proposes a stepwise modification of value set definitions. The right half of the transition table shall be interpreted as wrappings (4.1.3); i.e., the row names become vs in the new value set definitions; that is, to any value set, each of the constructors tuple, list, set, and array may be applied. The lower right area can also refer to complex conversions (4.1.5). Hence, either the top level constructor is changing (however, column vs remains the same as row vs in any case) or remains the same (diagonal cells). In the latter case, the value set constructor parameters are subject to change.

4.1.1 Property Modification

When defining a value set vs, the DBA has the option to define whether null values are allowed or not by using the keyword REQUIRED of the DDL. However, she might want to change this property upon new application requirements. While no impacts to the remainder of the schema exist, changing from non-required to required involves providing the missing values of the particular objects. This is considered to be cumbersome; i.e., we allow the change from required to non-required value sets only.

Changing the name of the value set requires checking both, referential integrity and name uniqueness of the value sets of the schema of which the value set under consideration is a part.

4.1.2 Simple Conversion

The cells with contents 4.1.2 denote the simple conversions from basic value sets to basic value sets. The effects to the remainder of the schema are marginal. However, all affected values have to be converted.

4.1.3 Wrapping

Typically, this kind of schema change occurs when the DBA has modelled a value set vs and now recognizes that more complex values (for example, multivalued) would be adequate. Hence, she alters the old value set to an array, list(vs), set(vs), or even a tuple(a: vs,...).

The remainder of the schema suffers no effects, but, however, all the values which have been of value set vs have to be converted; e.g., if we had a value set vs1 = set(tuple(a: integer, b: real) and now want to change it to set(tuple(a: list(integer), b: real), all the values of type vs1 suffer conversion.

The differences of the various wrappings are discussed in the following. The value set at the start of each item of the following enumeration shall denote the newly defined value set.

a) set(vs): The evolution operation takes no other further parameters than the old value set. By convention, values according to vs become the only element of the set. In other words, the new values are singleton sets.

b) list(vs): Similar to the previous case, the old value according to vs becomes the only element of the list.

c) array $n_1,..., n_n$ (vs): Besides the old value set, this operation will take the dimensions of the array as additional parameters. By convention, the existing value corresponding to vs will be the first element in the first dimension. The other values are undefined.

d) tuple(n: vs): Besides the old value set, this operation takes the name of the attribute as additional parameter[3]. By default, the existing value becomes attribute value in the newly created tuple value.

As a summary, wrapping operations are the only means to add more value complexity to the schema.

4.1.4 Unwrapping

Unwrapping refers to the change of multi-valued values to single-valued ones. We admit that this kind of change will not occur that frequently since it results into a loss of information. For example, if the DBA has stored a list of values ordered by their relative occurrence time (list(integer)), she might decide that it is too expensive to store historical values and therefore might opt for a schema modification with the objective to store one integer value (for example, the most recent one) only. The decision to be taken in this context is the selection of the most actual value. As one possibility, the DBA defines this decision by means of a *Quod* expression [Geppert 92a] which then passed as an actual parameter to the according unwrapping schema modification operation.

Similar to the wrapping of value sets, unwrapping has no effects to the remainder of the schema. However, for all the complex values, a query has to be executed in order to determine the element. The differences between the various kinds of unwrapping are defined in the following list. The items start with the complex value set to be unwrapped and result in a vs value set definition:

a) set(vs): The evolution operation takes as only arguments a query on a set value. The query must result in a value of value set vs.

b) list(vs): Again a query to a list value is taken as only argument of this schema modification operation. Typically the query consists of a list projection which results in a value of type vs.

c) array $n_1,...,n_n$ (vs): Very similar to b), but the query primarily consists of an array projection which delivers exactly one element of type vs.

d) tuple(a_1: vs_1,..., a_n: vs_n): In this case, the query is a tuple projection returning a value of type vs_i supposing that vs_i is the value set resulting of this schema change and vs_i is an attribute value set of the tuple value set to be unwrapped.

3. Additional attributes can be defined using the tuple value set modification operations listed in section 4.1.5.2

As a summary, unwrapping operations are the only means to collapse existing complex values to simpler ones. Of course, such operations incur a loss of information and should therefore (if at all) be used with precaution.

4.1.5 Complex Conversion

A complex conversion transforms a complex value set definition matching the pattern $\mathrm{constructor}_p(\mathrm{vs}_i)$ into a modified complex value set definition which matches $\mathrm{constructor}_q(\mathrm{vs}_j)$. As already pointed out in the introduction of this section, two kinds of complex conversions are distinguished: conversions that change the top level constructor while keeping the wrapped value set unchanged, and conversions that keep the top level constructor unchanged, but change some parameters[4]. These two kinds of conversions are elaborated in the following two subsections.

4.1.5.1 Changing the Top-Level Constructor

The following itemization contains not all possible n*(n-1) conversions, but only those which are considered to be meaningful. These operations are useful when the semantics of a collection of values do not change significantly. All these changes are considered to be local with respect to the remainder of the schema.

a) set(vs) → list(vs): In this case the DBA happens to decide that a homogeneous collection of arbitrary size is considered to be ordered from now on. The DBA therefore has to provide for the ordering criterion which is used to convert the set values to list values (e.g., by means of the *Quod* sort statement).

b) list(vs) → set(vs): This kind of change is rather trivial since it takes no parameter. The meaning is simply that the ordering criteria which has previously existed on the collection of values is not of future interest.

c) array(vs) → list(vs): The change of semantics in this operation is to abandon the fixed size restriction to array values of the particular value set. Again, when converting to a list, an ordering has to be provided. Either the DBA provides such an ordering or it defaults to the initial ordering of the elements in the array.

d) list(vs) → array(vs): Changing lists to arrays is more tricky since it is rather likely coupled to a loss of information; i.e., the flexibility on the number of elements gives room for a fixed number of elements for all values of this value sets. Thus, as an actual parameter of the operation, the number of elements has to be provided by the DBA. The elements of the array are filled using the order given by the list. Note that the element values of the list with an index higher than n are truncated; that is, they will get lost at commit of the conversion operation.

e) array(vs) → set(vs): The same holds as for the conversion of arrays to list except that no ordering criterion is allowed. The initial elements of the set value are the

4. Note that usually vs_i remains unchanged since we have adopted a one-step-at-a-time approach. Hence if the goal is to change vs_i, one would do it directly and not by using complex conversion.

prior elements of the array.

f) set(vs) → array n (vs): This is rather similar to changing lists to arrays except that the DBI has to provide explicitly for an ordering criteria. Again the first n elements are filled into the array while the other ones will be lost.

g) tuple(vs) → any: The purpose of the tuple value set constructor is to define a constant-sized collection of named values (attributes) with heterogeneous value sets. Hence, we refrain from proposing an operator which, for example, directly maps tuple value sets to lists. What we consider to happen more likely are sequences of unwrapping and wrapping schema changes; e.g., the DBA might decide to focus on one attribute of a tuple value set (unwrapping schema operation) and then to have a collection of such values (wrapping schema operation).

h) ref(ot) → dependency sharability ot: This kind of top level constructor change is applied whenever the DBA wants to specify additional structural constraints in the schema (implicit constraints); e.g., the DBA might want to specify the propagation of the deletion operation along subobject relationships (dependency:=dependent) and/or to express that the target of the reference shall not be referenced by more than one object (sharability:=exclusive). Of course, such conversions can prove expensive concerning the effects to existing objects. In order to facilitate the test for exclusiveness of a particular subobject reference between two objects, backward references have either to be stored or computed. Furthermore, acyclicity of the subobject relationship on object level has to be checked as well.

i) dependency sharability ot → ref(ot): This conversion represents a means to get rid of implicit integrity constraints which had been formulated in a subobject relationship. The effects to existing objects are marginal since the constraints on the subobject graph (if the general reference is interpreted as special subobject relationship) are rather weakened than strengthened, thus not presenting a subject to concern at runtime.

4.1.5.2 Changing Constructor Parameters

Depending on the top level constructor, different parameters may be subject to change:

a) set(vs) / list(vs): Since the set/list value set construction takes no parameters except the element value set, no modification operations are possible except that the element value set vs which can be changed according to the options listed in the vs row of the transition table.

b) array $n_1,..., n_n$ (vs): Fairly more interesting in this setting are the parameters of the array constructor. Existing dimensions can be made smaller (loss of values whose indexes are larger than the new size) or larger (values with index larger than the old size are unavailable). On the other hand, a *new dimension can be added* as a whole in that the DBA defines both, the range of the dimension and its index in the list of dimensions. Existing values remain intact even if their index is shifted. For example, consider an array of 20 b/w images of size 1024x1024 pixels; e.g., array 20, 1024, 1024 of boolean. The DBA might decide that not only images are to be stored, but 20 video sequences each of which lasts 5 seconds. Assuming that we need 24 frames per second, 120 frames have to be stored in essence. The DBI would therefore opt for a schema change which creates a new dimension of length 60 before the second one.

The second operation is to change the element value set vs of the array $n_1,..., n_n$ (vs) value set. Using the transition table one supposes vs to be the old value set and recursively checks the appropriate row for possible changes. In the example above, on might decide to store not only b/w images, but colored ones. Thus it might be appropriate to change boolean to integer having an index into a color table in mind.

c) tuple(a_1: vs_1,..., a_n: vs_n): Obviously, the tuple value set constructor has the most parameters and therefore provides more constructor parameter changes than the other constructors. Suppose a tuple value set representing a list of attributes.

Of course, it is possible to *change the name of an existing attribute* a_i. The potential effect of such a change to the remainder of the schema is the introduction of a name conflict with respect to multiple inheritance. However, the inherent integrity constraints, which prohibit conflicting attribute names, have to be preserved when attribute changes are changed. The effect to existing instances is marginal since this kind of change affects the catalogue only.

Similar to other schema evolution concepts, *dropping of an existing attribute* is possible as well. Assume that the tuple value set whose attribute a_i shall be deleted is part of the object type definition ot. The naming conflict clause of all object types ot_i which inherit a_i both from ot and from an arbitrary other object type, may become invalid. Hence, a dropping of attributes may propagate to changing the conflict clause of other object types. The effect to existing instances consists of removing the attribute value from all values which had the old value set definition as their base. Note that again that potential inheritance relationships offer potential to propagation of inherent integrity constraint violations.

The next basic operation is the *addition of a new attribute* to a given tuple value set definition. Potentially, such an operation can introduce a new naming conflict. However, this situation is rather easy to check. The changes to existing instances are rather small, and, depending on the realization strategy, might even be restricted to changes to the schema.

The last operation of interest is to *change the value set of an existing tuple attribute*. Note that this is again the point where we recursively consult the value set transition table (Table 2.); i.e., one could take the old vs_i and then look up the transition table again to both, check which changes are allowed at all and check what are the consequences of such a change.

d) sharability/dependence ot: As the last kind of parameter change, the modifications of the implicit integrity constraints related to the subobject relationship are discussed. Note that the two switches sharability/dependence allow four possible constraints which can be mapped into each other by the conversion concerned in this paragraph. Sharability is a structural constraint on the subobject relationship on object level, denoting that an object of the subobject type cannot be shared among objects of any parent object type. Hence, transforming sharable to exclusive involves checking for each subobject whether it is shared or not. Again, this requires either storing or computing backward references. The other way round (i.e., changing exclusive to sharable) is cheap.

The other constraint, dependence, rather determines the semantics of (generic) operations than the object structure. Hence, only the schema is affected for both kinds of changes, i.e., from dependent to independent and independent to dependent.

4.2 Object Types / Derived Types / Union Types

In this subsection, modifications to derived types, object types, and union types are discussed. Since union types and derived types have no extensions of their own, no impacts occur when modifying them. Hence, changing derived types and union types is rather uninteresting; that is, we refrain from offering detailed modification operations except the obvious creation and deletion. However, both creation and deletion have to preserve inherent integrity.

Principally (may be read as: syntactically), an object type consists of a name, a value set definition, a list of object types where the object type inherits from, and possibly an extension clause, refinement clause, restriction clause, and/or a unique clause. From this particular composition, different schema changes are directly derived and discussed in the next subsections.

4.2.1 Changing the List of Supertypes

Removing the object type ot' from the list of supertypes of an object type ot does not affect the set of nodes in the inheritance DAG. However, one element is removed from the set of edges; i.e., the element which connected ot and ot'. As a consequence, any conflict clauses of (subtypes of) ot referring to tuple attributes of ot' become invalid. The same holds for value set refinement clauses. Moreover, ot loses the tuple definitions of ot'. Of course, this leads very likely to a massive conversion of the instances of (subtypes of) ot, since its value set definition is completely changed. However, from a schema integrity point of view, removing an edge in the inheritance DAG is rather unproblematic since the DAG acyclicity invariant can be violated on insertion of a new edge only (supposing that the inheritance graph has been consistent before).

With respect to change propagation to objects, adding the object type ot' to the list of supertypes of an object type ot is rather similar to removing it, since the value set of ot is very likely to change massively. With respect to schema integrity, adding an edge to the inheritance graph implicates the risk of violating inheritance acyclicity.

4.2.2 Changing the Object Type Name

Changing an object type name affects the schema only; i.e., no objects have to be modified. If the new name fulfills the schema invariants, all the occurrences of the old object type name have to be replaced by the new one. Object type names typically occur in subtype lists, subobject definitions, general references definitions, derived type definitions, and in schema definitions.

4.2.3 Changing the Value Set Definition

The procedure of changing the value set definition vs of an object type ot simply consists of analyzing the definition of vs and modifying parts of it according to Table 2.

4.2.4 Changing the Refinement Clause

The purpose of the refinement clause is to modify the value set of an inherited tuple attribute. Hence, the corresponding schema change operation transforms (at least from a conceptual point of view) to a value set modification, i.e., to a conversion of subranges to more restrictive subranges or to a conversion of an enumeration to one with fewer elements. Thus, refer to section 4.1.2.

4.2.5 Changing the Conflict Clause

Given an object type definition, its conflict clause resolves possible name clashes occurring in the context of multiple inheritance by means of attribute selection. Changing the attribute selection involves changing the value set of an attribute, again. Note that the attribute selection is constrained by the possible value set changes (refer to Table 2.) with respect to the old value set.

4.2.6 Changing the Extension Clause

The purpose of an extension clause of an object type definition is to enhance the inherited tuple value set definitions with additional attributes. Hence, changing the extension clause refers to change operations which are possible for tuple value sets (see 4.1.5.2, paragraph c).

4.2.7 Changing the Unique Clause

The unique clause of an object type definition enables the DBA to prohibit duplicate attribute values of different objects. Since changing a non-unique attribute to a unique one involves duplicate elimination, we refrain from providing this kind of schema modification operation. However, the opposite (to redefine a unique attribute (combination) as non-unique) is allowed. No impact to the remainder of the schema exists except that this property is inherited by subtypes (according to DDL, p. 7). Furthermore, no impact to existing objects exists, either.

4.3 Schema and Database Definitions

In this subsection, we deal with the issues of the top-level DDL constructs, i.e., schema and database definitions. Indeed, the discussion of the latter construct can be kept brief, since a database definition assigns a logical schema with an (empty) database instance. The only kind of change that the DBA can take in this context is to unassign the database instance from the schema; that is, to drop the entire database which obeyed the schema of interest. In other words, no impacts to the schema exist, but the particular logical database gets lost.

On the other hand, logical schemas and subschemas offer a variety of possibilities for modifications which are presented in the following subsections.

4.3.1 Adding and Dropping an Object Type to/from a Logical Schema

The creation of a new object type is already covered by the current DDL definition, hence discussion can be kept brief. This operation is one of the few which has hardly noticeable effects to existing objects since it represents the only means to create new extensions; that is, it represents the operation which creates, as a desired side-effect, a container for new objects. However, whenever creating an object type, the name of the new object type must not violate the inherent name uniqueness constraint (for a discussion of the inheritance constraints refer to 4.2.1.).

In contrast, the dropping of an object type ot has effects to both the remainder of the schema and to the objects which populated the extension of ot. While the latter simply refers to the deletion of all objects contained in the extension of ot, the former requires a more detailed analysis. Thus, we analyze in which situations an object type ot can be component of another one and describe the integrity constraints which can be violated in such situations.

a) ot is in the supertype list of ot'. Refer to section 4.2.1.

b) ot is referenced by object type ot' in terms of a general reference. Consequently, the value set definition of ot' will violate schema referential integrity.

c) ot is referenced by object type ot' in terms of a subobject reference. Again, the value set definition of ot' will violate schema referential integrity.

Refer to section 5 for a discussion of strategies to avoid (or to repair, respectively) such inherent integrity constraint violations.

4.3.2 Adding and Dropping a Value Set to/from a Logical Schema

Adding a value set to an existing logical schema does not offer any problems, unless the inherent constraint which prescribes unique value set names is violated. No impacts to the remainder of the schema and database exist.

In contrast, dropping a value set from a logical schema involves checking for the referential integrity of the remainder of the schema. Since a particular value set may be referenced either as top-level structure description of different object types or as components of other value set definitions, referential integrity is violated very likely (unless the value set has been completely isolated beforehand). Again, refer to section 5 for a discussion of strategies to avoid (repair, respectively) of inherent integrity constraint violations.

4.3.3 Adding and Dropping of a Union Type to/from a Logical Schema

Modifications to the logical schemas with respect to union types have similar characteristics as modifications to object types except that they are simpler, since union types do not participate in inheritance graphs and have no extensions of their own. Consequently, it is principally less risky to run into an inherent constraint violation.

Adding a union type conceals the risk of name-uniqueness violation. Since a union type name may appear as subobject type or as target of a general reference, dropping a union type may endanger referential integrity.

4.3.4 Adding and Dropping of Derived Type to/from a Logical Schema

Derived types can be added to logical schemas at arbitrary points in time. Dropping of a derived type may, however, affect existing subschemas, and possibly other derived types which rely on the former one. Thus, again referential integrity has to be preserved.

4.3.5 Adding and Dropping of a Type Name to/from a Subschema

Subschemas represent collections of type names, i.e., union type names, derived type names, and object type names defining the visible portion of a logical database from an application point of view. Of course, the name collection can grow and shrink during the lifetime of an application. Hence, type names can be arbitrarily added and dropped to an existing schema. The only constraint that has to be asserted is closure and well-definedness of subschemas [Geppert 93].

5 Realization Strategies

In this section, various realization strategies are discussed with respect to value set changes, object type changes, and schema changes. Each of these strategies has to preserve the inherent integrity constraints of NO^2 schemas which have been introduced in section 3.

5.1 Propagation within the Remainder of a Schema

As the reader may have noticed, complex schema modifications may be mapped to simpler ones. In other words, in order to preserve inherent integrity, it might be necessary to apply several individual schema operations. For example, if an object type is dropped, all the references to this object type have to be dropped as well (unless the object type has already been completely isolated). In such a scenario, three strategies are distinguished where each of them preserves inherent integrity in a specific manner.

5.1.1 Single Modifications as Schema Transitions

In this strategy, each singular schema modification is regarded as a schema state transition, hence, is required to end in a consistent schema state. The detection of integrity violations is accomplished using the terminology discussed in section 3 and 4. With respect to the action to be taken in case of a constraint violation, two substrategies can be identified. First, all actions violating inherent integrity could be prohibited (reads: aborted). Second, if the execution of an action would leave some remaining parts of the schema in an inconsistent state, automatic repair actions could be devised depending on the constraint violated and the previous modification (change propagation).

The advantage of the former is its rigorousness since the effects of a schema modification are rather local in contrast to the modification propagation substrategy which can in turn lead to the deletion of the whole database in case of the DBA opted to delete a “root” object type, for example.

5.1.2 Groups of Modifications as Schema Transactions

This strategy presumes that a sequence of schema modification operations (schema transaction) transits from on old schema state into a new consistent one. Hence, at the end of such a transition, inherent integrity has to be checked. As a consequence, such a strategy requires a mechanism which accumulates the inconsistencies occurring during the execution of a schema transaction. If, at the end of the schema transaction, inherent integrity is violated, the transaction is either aborted (backward recovery) or the transaction takes necessary repair-actions (forward recovery). Of course, since such schema transactions typically have an interactive nature, it would be a great help for the DBA if she could easily track actual inconsistent schema definitions.

5.2 Propagation to Objects Corresponding to a Schema

A prerequisite to a discussion on the major strategies to realize the propagation of schema changes to affected objects is a procedure which computes all impacted objects.

We start with changing a value set and computing the values conforming to the old definition (vs_{old}), which then have to be converted to the new value set definition (vs_{new}).

1. vs_{old}:= old value set definition from the catalog.
2. Retrieve the set of value set definitions $vs_{impacted}$, of which vs_{old} is a component.
3. For each value set of $vs_{impacted}$, repeat step 2 until the overall set of impacted value sets does not grow anymore. If the actual value set directly defines the structure of an object type, the objects in its extension are subject to conversion.

This algorithm holds for any change to a value set, however it does not address the potential changes when the inheritance graph is modified. However, these changes can be expressed by value set changes, after all.

The different propagation strategies described below can be mainly classified by the point of time when the object conversions take place. Depending on the access pattern of a database, the selection of the strategy may have impacts on both, the performance of the schema change itself or the performance to future database accesses.

5.2.1 Eager Propagation

This strategy requires the affected objects to be converted as a part of the schema transition. In other words, object conversions are carried out immediately. Depending on the number and size of the affected objects, the execution of a schema change may lead to a blocking of the entire database, since large parts of the database can be affected by one operation. Hence, this option might only be realistic in settings where there exist intervals of time where the database system is idle (e.g., during the night). If there exists an object access pattern such that the ratio of creation and access is not that small (hence objects are not accessed too often after their creation), it is possible that eager propagation converts objects which are not used for a long time and conversion effort is spent although actually not (yet) needed.

5.2.2 Lazy Propagation

This strategy requires the impacted object to be converted as a part of the transaction which accesses the object the first time after schema modification took place. Hence, there is a penalty for the first transaction which accesses an object to be modified. This may lead to time indeterministic behavior since sometimes, the access comprises a conversion. Of course, the advantage of this strategy is that only objects that are accessed in the future are converted at all. Furthermore, the lazy strategy dissipates the conversion effort on a larger interval of time, and thus no idle time interval is needed in order to take the object conversions as in the eager case.

6 Discussion

In this report, we have presented an overview of the notions and issues in the realm of schemas which undergo modifications during their lifetime. We have given a summary of the inherent integrity constraints of NO^2, that is, the constraints to be asserted by any transition from one schema state to another. Furthermore, a taxonomy of the possible transitions has been given whereas the effects of each transition to the remainder of the schema and database have been enumerated; i.e., to a large extent, these operations require reorganization of the database. Therefore, different realization strategies have been discussed. The next step will include both the selection of the schema modification operations to be effectively supported and the according realization strategy, and finally, the extension of the *Quod* DDL syntax to capture these operations.

7 Bibliography

[Atkinson 89] Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D., Zdonik S.; The Object-Oriented Database System Manifesto; In: Proc. 1st Intl. Conf. on Deductive and Object-Oriented Databases, Kyoto, JPN, 89; pp. 40-58;

[Banerjee 87] Banerjee J., Kim W., Korth H.; Semantics and Implementation of Schema Evolution in Object-Oriented Databases; In: Proc. ACM SIGMOD 1987 Intl. Conf. on Management of Data, San Francisco CA, USA, May 87; ACM, New York NY, USA, 87; pp. 311-322;

[Cattell 91] Cattell R. G. G.; Object Data Management; Addision Wesley, Reading MA, USA, 91;

[Delcourt 91] Delcourt C., Zicari R.; The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System; In: Proc. European Conf. on Object-Oriented Programming, Geneva 91; LNCS 512, Springer, Berlin, FRG, 91; pp. 91-116;

[Dittrich 92] Dittrich K. R., Geppert A., Goebel V., Scherrer S.; The Data Definition Language of NO2; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; February 92;

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

[Geppert 92a] Geppert A., Dittrich K. R., Goebel V., Scherrer S.; Quod: A Query Language for NO2; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; January 92;

[Geppert 92b] Geppert A., Dittrich K. R., Goebel V., Scherrer S.; An Algebra for the NO2 Data Model; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; January 92;

[Geppert 93] Geppert A., Scherrer S.; A View Mechanism for NO2; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; January 93;

[Zicari 91] Zicari R.; A Framework for Schema Updates in an Object-Oriented Database System; In: Proc. 7th Intl. IEEE Conf. on Data Engineering, Kobe, JPN, February 91; IEEE Computer Society, Los Alamitos CA, USA, 91; pp. 2- 13;