Advanced Primitives for Changing Schemas of Object Databases

Philippe Brèche*

Projet VERSO – INRIA Rocquencourt
BP. 105 78 153. Le Chesnay Cedex - France
Philippe.Breche@inria.fr

Abstract. Currently, existing Object Database Systems (ODBSs) perform schema changes by means of primitives closely related to their respective data model. Software Engineering (SE) applications, Object Methodologies (OM) and designers building up object database schemata require a more abstract level. This paper addresses new facilities for updating a schema filling the gap between object design and object programming.
A set of advanced primitives, so–called "High Level Primitives", is presented to cope with these requirements. The semantics of these new primitives and how to maintain a schema consistent after such a schema update are the main contributions of this paper.
We discuss issues on implementation on top of the commercial ODBS O_2 and consider related work.

1 Introduction

In the last decade, the object paradigm benefited from a great success, successively covering the various steps of the software life cycle, from analysis to testing, and various software domains, in particular the database system field. Object technology, particularly (but not exclusively!), aims at producing software that is easy to understand and to re-use. Fully enabling these properties, requires more flexible and more powerful means of accessing, interpreting and evolving software modules. Regarding this latter point, achieving more flexibility on evolving software depends on two main parameters of software engineering:

- *change management*, which facilitates the changes occurring in the object software life cycle, and
- *reuse*, encouraging people to re–use and to integrate software modules already developed.

Change Management is the focus on this article. In the ODBS field, this topic referred to as Schema Change Management [4, 10, 25, 27], is widely addressed.

* On leave from DBIS department at the J.W. Goethe University, Frankfurt/Main, Germany. Partially supported by Esprit III project Goodstep.

Currently, most existing ODBS products and prototypes do not allow free and dynamic changes to the schema. They merely allow the performance of schema changes by means of primitives (from here on called *LLPs — Low Level Primitives*) close to the respective underlying data model. Taxonomies of such schema update primitives can be found in [4, 10, 15, 26, 16]. They deal with the creation, deletion or modification of the schema definitions of the data model, classes and properties (attribute and method) and some additional concepts specific to the ODBS. They may also differ regarding the level of integration between the data, the schema and the meta–schema, if any [26]. These primitives are considered as a first level of schema update primitives.
In contrast to this, Software Development Environments (SDE) should support more declarative complex/advanced operations occurring in the design process such as *generalization* or *merge* of definitions. Most of the object methodologies, such as OOSE [2] [9], OOSA[2][18], OOD[2][6], RDD[2], OMT[2][5]) highlight the need for designers to build information models and their corresponding database schemas at such an abstract level through all the incremental and spiral design process. The designer has to fill the gap between the functions currently provided by ODBS environments and the design requirements. For example, to generalize classes and to insert the class of generalization in the hierarchy, a designer usually spends time collecting the required information, and thus to find, group and order the appropriate basic primitives *LLPs* changing the schema. To do it properly, the designer has to be aware of the rules which maintain the consistency of the ODBS data model. To find the right *LLPs* accordingly, the designer has to foresee and to solve the intermediate conflicts which appear. Moreover, functions have to be written and run against the base to propagate the structural changes to the data and views. Application programs have to be checked and eventually changed to run against the new schema. All these maintenance operations are time expensive and error prone. New declarative complex schema change primitives (hereafter *HLPs – High Level Primitives*) are developed to free the designer from performing mundane tasks.
Schema changes are strongly related to the notion of correctness. Our approach of *HLPs* amounts to facing classical problems of schema change: specify the *semantics* of the new primitives and accordingly, specify *rules* to preserve the *consistency* of the resulting schema (the database consistency is not addressed hereafter). These two items are the main contributions of this paper.
This work was partially funded by the broader GOODSTEP project [30] under the ESPRIT III program. The baseline of the project is an existing commercially available ODBS, O_2.
The rest of the paper is organized as follows. Section 2 gives an overview of the *HLPs* semantics. Section 3 shows how to preserve a schema consistent after a schema update. Section 4 finally concludes on related work and points to future developments.

[2] OOSE: **O**bject–**O**riented **S**oftware **E**ngineeering, OOSA: **O**bject–**O**riented **S**ystems **A**nalysis, OOD: **O**bject–**O**riented **D**esign, RDD: **R**esponsability–**D**riven **D**esign and OMT: **O**bject **M**odeling **T**echnique.

2 High Level Primitives for Changing a Schema

The *HLPs* library we have implemented aims at providing the designer with new schema update facilities. It is based on the existing O_2 schema update primitives. The *HLPs* can be used in application program or interactively, through a GUI, by a schema designer.
We first detail the current O_2 *LLPs* for changing a schema. The general semantics of the newly developed primitives, *HLPs*, together with a revised taxonomy for schema modification, complete this Section.

2.1 Taxonomy of Current Schema Updates in O_2

In order to assess the level of schema change facilities offered by O_2, we have defined a taxonomy of its available basic update operations. The classification differentiates updates concerned with class structure, method or class hierarchy. Such a taxonomy gives an indication of the underlying data model. At this stage, the O_2 schema update facilities [19] (the *LLPs*) cover the taxonomy found in Figure 1.
So far, the O_2 *LLPs* are used by a designer either interactively by means of an O_2 shell interpreter, a class browser GUI, or embedded in C/C++ alike programs (calls to O_2API functions).

2.2 The New O_2 Schema Modification Primitives

In this section, we define the semantics of the set of *HLPs* that have been added to the O_2 database for changing the schema. The choice of such primitives is motived by a detailed analysis of schema update operations occurring in schema design. They are inspired from our practice of OMT, Booch and Schlaer & Mellor methodologies. Together with general definitions of these primitives, we present the important steps that occur when running them. These steps approximate the equivalent ordered set of *LLPs* that a designer should apply to perform an identical schema change. Intuitive definitions often give way to more formal definition for clarification. After such a schema change, the O_2 invariants hold. Rules of interest that preserve these invariants and allow a consistent target schema to be attained after the update are detailed in Section 3.
The mode which promotes interactions with the designer directed our attention, enabling us to seek solutions for the undecidable cases of certain schema changes. In such cases, further information provided by the designer avoid having to abort the update or to degrade the target schema.
These definitions extend the taxonomy of Section 2.1 to a new one discussed in Section 2.3.

Abstraction–Generalization of Classes As opposed to the specialization facility [3] which allows a designer to refine general concepts to reach a "solution", an

[3] This is usually provided in most of the ODBS.

1. Changes to the structure of a class (node)
   1.1. Changes to the type of a class
   1.2. Changes to attributes (in case of type `tuple`)
      1.2.1. Add a new attribute to a class
      1.2.2. Delete an existing attribute
      1.2.3. Change the name of an existing attribute
      1.2.4. Change the type of an existing attribute
      1.2.5. Change the visibility of an attribute (public/read/private)
      1.2.6. Change the inheritance (parent) of an attribute
      1.2.7. Rename an inherited attribute
   1.3. Changes to visibility of a class type (encapsulation – public/read/private)

2. Changes to the behavior of a class (node)
   2.1. Add a new method
   2.2. Delete an existing method
   2.3. Change the name of an existing method of a class
   2.4. Change the code of a method in a class
   2.5. Change the visibility of a method (public/private)
   2.6. Refine the code of an inherited method
   2.7. Change the inheritance (parent) of a method
   2.8. Rename an inherited method

3. Changes sub/superclass relationship (edge)
   3.1. Add a new superclass
   3.2. Delete an existing superclass

4. Changes to the class hierarchy (node and edge)
   4.1. Add a new class
   4.2. Drop an existing class
      4.2.1. a leaf class in the DAG
      4.2.2. a class and all its subclasses
   4.3. Change the name of a class
   4.4. Change a class share-ability (export/import)

Fig. 1. Current O_2 Taxonomy of Schema Updates

essential facility is being able to generalize information from existing concepts, that is, to factor information. This allows the improvement of classifications and the gaining in generality and modularity for further reuse purpose. This operation occurs in bottom up approaches in a design life cycle.

For example, a designer generalizes class `Employee` and class `Client` into class `Person`. This is performed factoring out the common properties of the source classes `Employee` and `Client`. The factorization is based on the (user–defined)

names of the properties. Renamings are emphasized and lead to interaction with the designer. A factorization of methods' implementations is performed following an interactive process only, due to the well–known non–safe use of polymorphism [11]. The source classes are kept and re–connected to the newly created class of generalization, C_g . They are refined accordingly.
Two main steps occur in this process:

– *How to construct the content of the class* C_g
  The specification of the features of the class is performed by means of two operators which generalize the source class types and the source classes methods. They look for *common names* and *intersect* the types (class types, attribute types and method signatures) following the covariance. The encapsulation (rights) is constructed, keeping in mind that a class should be only less "visible" than or equal to its subclass.
  Thus, further information has to be inferred to complete the class specification, namely:
  - the location of the class in the hierarchy, that is, the list of super-classes and subclasses of C_g . This location is such that C_g subsumes (directly) the two source classes whereas the lowest common super-classes of the source classes subsume it.
  - those properties that are inherited and locally defined in the class,
  - the list of inheritance links starting from the source classes; they have to be updated (deleted),
  - the list of properties to be deleted in the source classes,
  - the various conflicts to be solved in the source classes and their subclasses: name conflicts, eventual attributes/methods precedence, repeated inheritance conflicts and property identity conflicts.
– *How to realize the class* C_g *through a set of LLPs*
  Once the information dealing with the specification of the class is computed, the class is created and inserted in the hierarchy throughout an ordered set of basic updates. Hereafter, we simulate the order that a designer would follow to reach the target schema, applying the set of *LLPs* according to the information collected in the previous steps:
  1. create an empty class directly under the lowest common super-classes,
  2. fill the class with the locally defined properties,
  3. store and drop those properties in the source classes and their subclasses that create further conflicts in step 4, as well as the common properties already factored in the class of generalization,
  4. delete the inheritance links between the source classes (or eventually intermediate classes in between the source classes and the lowest common super-classes) and their lowest common super-classes,
  5. re–connect the two source classes (or the intermediate classes according to 4) to the generalization class,
  6. re–create eventual properties of the source classes and their subclasses concerned by step 3, filtering those of the properties of the source classes which are already present in the generalization class.

The corresponding synopsis of the **Generalize** primitive is expressed by:

generalize [classes] *Class_name* [,[*Class_name*]...]
into class *Superclass_name*

Abstraction–Specialization of Classes This primitive specializes one source class into one, two or several classes, which become subclasses of the source class. For example, designing figures, with class `Figure`, currently sub-classed with `Point`, `Line`, `Spline`, `Polygon` and `Circle`, a further classification is performed regarding the dimensions of the figures. Class `Dim_0`, `Dim_1` and `Dim_2` are inserted for this purpose.
This primitive completes the specialization usually found in the basic ODBS updates under class inheritance declarations (bottom–up approach). In the design process, this new facility is suitable while mixing bottom–up and top–down approaches.
A class existing in the class hierarchy is specialized into new classes inserted in the hierarchy. The location of these specialized classes might be constrained specifying super-classes and subclasses.
The consistency of the newly created inheritance links in the class hierarchy is the main point addressed in such a schema update. First, each specialized class is proven to be in a good sub-classing/super-classing relationship with its declared subclasses and super-classes. Then, the class of specialization is built up taking care of restructuring of properties in subclasses of the class of specialization, before and after the insertion of the classes in the hierarchy. Just as for generalization, these steps entail resolving renaming and repeated inheritance conflicts.
The specialization is addressed by the following declaration:

specialize [class] *Class_name* **into classes**
Subclass_name [**super**(*Class_name* [[, *Class_name*]...])]
[**sub**(*Class_name* [[, *Class_name*]...])]
[[, *SubClass_name* [**super**(*Class_name* [[, *Class_name*]...])]
[**sub**(*Class_name* [[, *Class_name*]...])]]...]

Accordingly, the following declaration maps the schema update proposed in our example with figures:

specialize class *Figure* **into classes**
`Dim_0` **sub**(`Point`), `Dim_1` **sub**(`Line , Spline`), `Dim_2` **sub**(`Polygon , Circle`)

Abstraction–Merge Classes Merging classes allows the refinement class hierarchy, changing generality, modularity and granularity of the model. This occurs both in a bottom up and top down approaches in a design process.
This primitive merges several source classes into one new class definition, deleting the source classes and consequently restructuring the class hierarchy. Options enable the specification of the kind of strategy carried out while merging the source

classes. *Union*, *intersection* or *difference* operators applied to the properties of the source classes are available. The primitive restructures the hierarchy, usually replacing the source classes to be merged by the class of merge.

Remark: regarding the chosen option and due to covariance constraints, it is not always possible to delete the source classes and to reconnect their subclasses to the class of *merge*. In this case, the class of *merge* is merely created and well–located in the class hierarchy whereas the source classes are kept unchanged.

To illustrate this notion, the classes `Employee` and `Client` are merged into the class `Person` with a criterion of properties generalization. The intention is to preserve the common information from the source classes, removing these classes in the target schema. Similarly to class generalization, this is performed by factoring the common properties (attributes and methods' signatures) of the source classes `Employee` and `Client`. The source classes are removed, their subclasses are re–connected to class `Person` and refined consequently. One might also choose to keep all the information from the source classes through a *union* option or to retain only information that distinguishes the source classes through a *difference* option.
The merge process carries on a new class creation and its insertion in the class hierarchy. This entails first building up the content of the virtual class of merge according to a criterion. Then, according to the properties of the class, one has to "classify" this class in the class hierarchy. The last step is to materialize the class and to restructure the source classes (and their subclasses) through a set of ordered *LLPs*.
The corresponding synopsis of the **Merge** primitive is expressed by:

merge [class] [– i | u | d] *Class_name* **[[,** *Class_name*] ...]
into *Class_name*

where *i, u, d* stand for intersection, union and difference (resp.).

Remove a Class The primitive *remove* deletes a class anywhere in the class hierarchy. The class hierarchy then has to be reorganized.
Let C_{del} be a class located in the middle of the class hierarchy (C_{del} has super-classes and subclasses). In addition, C_{del} has specific features specified as properties locally defined in the class.
Assume now that the class C_{del} is to be deleted. The policy of the designer is to retain in subclasses of C_{del} information specific to class C_{del}. This information was previously inherited. To be preserved, this information should be locally redefined in the direct subclasses of C_{del}. The designer has to *propagate* the properties locally defined in C_{del}.
At the same time, the designer decides that the subclasses of C_{del} should also preserve the information coming from the super-classes of C_{del}. Another designer argues that the classification of the class hierarchy should remain "similar" after the class deletion. Direct sub-classes of C_{del} should therefore be reconnected to the direct super-classes. This is called *reconnection*.

The previous examples attempt to illustrate the various options that might be required when deleting a class. This can be summarized with the willingness of preserving or losing information captured in a class C_{del}. It includes information about static (attributes) and dynamic (methods) properties and, at the same time, locally defined or inherited properties. Thus, deleting a class C_{del}, a designer may choose different options, combining *reconnection* to the super-classes of C_{del} and *propagation* of properties local to C_{del}, as follows:

1. propagation and reconnection preserves information locally defined and inherited through C_{del}. Properties locally defined in C_{del} and not locally redefined in the sub-classes are propagated. All sub-classes of C_{del} are reconnected to every super-class of C_{del}. This leads to information redundancy but retains the information.
2. no propagation, no reconnection loses information captured by C_{del}. The subclasses are not re–connected to the super-classes and the properties locally defined in C_{del} are not propagated to its subclasses.
3. propagation, no reconnection preserves the information locally defined in C_{del}. Information inherited from the super-classes through C_{del} is lost but the one locally defined in C_{del} is preserved. This is performed propagating (copying) the properties into the direct sub-classes of C_{del} unless the direct sub-classes redefine them. This may lead to a waste of information or to information redundancy.
4. no propagation, reconnection loses information locally defined in C_{del} but does preserve the one inherited from C_{del}. Properties locally defined in C_{del} are not propagated but each direct sub-class of C_{del} is reconnected to each super-class of C_{del}.

Performing such a class deletion amounts to find out an ordered set of *LLPs*. These *LLPs* delete inheritance links and create properties, taking into account name conflicts, repeated inheritance conflicts and property identity conflicts. This is further detailed in the example in Section 3.3.
The corresponding synopsis of the **Remove** primitive is expressed by:

remove [– p|–r|– pr] [class] *Class_name*

where *-p* propagates the properties locally defined in C_{del} whereas *-r* reconnects all the subclasses of C_{del} to its super-classes.
Readers interested on more details are referred to [8].

Abstraction–Aggregation This primitive creates an *aggregation* class which is built up out of several source classes. The designer can specify properties of aggregation links (e.g. cardinalities) and can establish *part_of* reverse–links (see the *composite* objects in [4]). For example, starting from class definitions for `Monitor`, `System_Box`, `Mouse`, `Keyboard`, etc, one can introduce the class `Micro-computer` with the attributes `Mouse`, `Keyboard` ...

Its representation is a one level−tree in which all nodes are classes. Figure 2 illustrates the example of the `Microcomputer`. The root of the tree represents the class produced by as the aggregation of classes at the leaves. Each arc in the tree, states that a leaf class is a part of the root class.

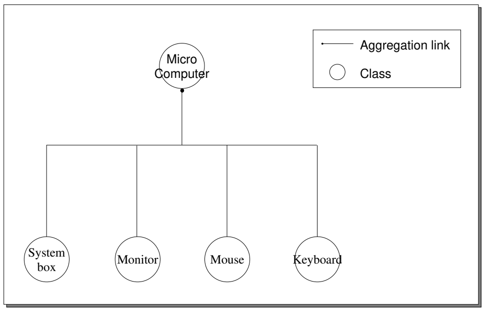

Fig. 2. Aggregation abstraction: the Microcomputer example.

The corresponding synopsis of the **Aggregation** primitive is expressed by:

> **aggregate [class]** [[−r][−b|l|s]] *Class_name* [[, [[−r][−b|l|s]] *Class_name*]...]
> **in [class]** *Class_name*

The option *-r* specifies a reverse link creation in the aggregated class. the options *-b|l|s* specify the cardinality of aggregation through constructors *b*ag, *l*ist or *s*et (resp.). By default, a one to one aggregation is constructed.
The following declaration updates the schema

aggregate −r−b `Monitor`, **−r** `System_Box`, `Mouse`, `Keyboard`
in class `Microcomputer`

creating class `Microcomputer` with the attributes of types bag of `Monitor`, `System_Box`, `Mouse` and `Keyboard`.
This primitive does not lead to unsual difficulties regarding either the schema consistency maintenance or regarding the set of *LLPs* and the order of application.

Abstraction–Decomposition This primitive acts in a way opposite from the way the aggregation primitive acts. It creates classes out of decomposition from a set of attributes addressed in a source class [4]. The reverse link *is_a_part_of* is automatically created in these classes. Only those attributes which are specified together with a class generate a class of decomposition. A *-s* option allows switching an attribute type specification to a newly created class type. The name of the new class is declared together with the name of the attribute addressed

[4] The source class is assumed to be of type *tuple*.

by this change. The "deepest" type specification of the attribute is taken into account while converting the type.
In addition, the source class may be preserved or dropped.
Accordingly, the corresponding synopsis of the **Decomposition** primitive is expressed by:

decompose [-d] [**class**] *Class_name*
[**into** [-s] *attribute_name Class_name*] [, [[-s] *attribute_name Class_name*]] ...]

The option *-d* deletes the source class whereas, by default, the class of decomposition is preserved. The *-s* option switches the attribute type.
For instance, class *Car* is decomposed into classes *Color*, *Company*, *Motor* and *Price* as in Figure 3 by means of a simple decompose statement.

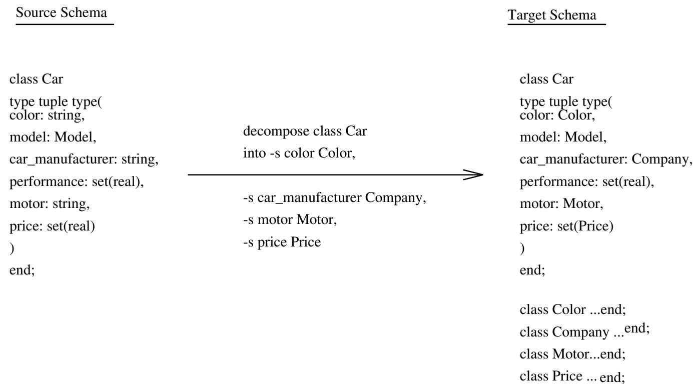

Fig. 3. Decomposition abstraction: the Car example.

2.3 A Revised Taxonomy for Schema Update Primitives

Consequently, we have revised the O_2 taxonomy for schema update primitives found in Section 2.1 as in Figure 4, including the new complex schema changes. In addition, these new facilities are available via an interactive GUI which helps the designer while performing schema change.

3 Schema Consistency and Schema Change

A schema update should result in a new schema with no inconsistencies. Moreover, the consistency of the other elements relying on a schema also have to be maintained. Potentially, a schema update may make the database(s) instance(s) of the schema inconsistent, as well as the view(s) based on the schema and some

4. Changes to the class hierarchy (node)
  4.2. Drop an existing class
    4.2.3. remove a class in the middle of the hierarchy
    4.2.4. remove a class in the middle of the hierarchy – propagate
    4.2.5. remove a class in the middle of the hierarchy – reconnect
    4.2.6. remove a class in the middle of the hierarchy – reconnect and propagate

  4.6. Change to a set of classes
    4.6.1. Generalization of classes
    4.6.2. Specialization of classes
    4.6.3. Merge two classes
      4.6.3.1. Merge–Unify classes
      4.6.3.2. Merge–Intersect classes
      4.6.3.3. Merge–Subtract classes

5. Changes to the class hierarchy and classes
  5.1. Aggregate classes into an attribute of a class
  5.2. Decomposition of an attribute into a set of classes

Fig. 4. Revised Taxonomy of Schema Updates

application programs running on top of the schema/base. Strictly relating to the schema, to preserve the consistency following an alteration of the schema relates to both the structural aspect and the behavioral aspect of the schema. This depends on the consistency definitions or *invariants* of a given ODBS and its peculiar rules to preserve this consistency.

Therefore performing any kind of *HLP* update should preserve as well the O_2 invariants. Hereafter we apply to the O_2 data model the methodology used in [4, 22] to specify the formal framework for schema evolution. Like ORION, the O_2 model incorporates all the basic object concepts. Data might be *values* or *objects*. The classes encapsulate the data defined by an underlying class type (the atomic types, the constructors list, set and bag and the class type are possible types) together with methods that operate on the data. A class might inherit from different super-classes (multiple inheritance). The inheritance mechanism is defined on a covariant sub-classing relationship and is carried out with *late binding*.

An O_2 class is roughly a quadruple c =(name, vis, type, methods), with a name, an access right *vis* (hereafter called visibility) that may be public, read or write, visibility defined on the global type *type*, and a set of methods. An attribute is quadruple a = (pid, name, vis, type) with an *pid* that identifies uniquely the property along an inheritance path, a name, an access right and a type (signature). Similarly, a method is a quintuple m = (pid, name, vis, si, body),

with a pid, a name, an access right, a signature (typed arguments and result) and an implementation.
Readers are encouraged to have a look at [3, 1, 20] for more details. The O_2 features lead to set of invariants a slightly different from that of ORION or Gemstone, but to different rules to preserve these invariants. In a first attempt, schema change focuses on preserving the structural consistency of the schema.

3.1 O_2 Invariants

The structural consistency is specified through the constraints listed bellow, so–called invariants of the database system, to be preserved after each schema update.
Hereafter, *property* denotes the notion of both *attribute* and *method* of a class. *Node* and *edge* in a graph representation of the class hierarchy stand respectively for a class (labeled with its name) and an inheritance link between two classes (labeled by the name of the classes).

1. **I1** Class Hierarchy Invariant
   The class hierarchy is a rooted (the O_2 root class is called *Object*) and connected (there is no isolated node) Acyclic Directed Graph (DAG) with named nodes and labeled edges.
2. **I2** Distinct Name Invariant
   All classes of a schema have distinct names (unique label for the nodes in the DAG).
   All properties of a class (locally defined or inherited) have distinct names.
3. **I3** Distinct Identity Invariant
   All the properties of a class have distinct identity (also referred to as *origin* or *pid*).
4. **I4** Full Inheritance Invariant
   A class inherits all properties (except the method *init*) from each of its super-classes unless there are conflicts related to invariants 2, 3 and 4. There are of two kinds of conflicts: name conflicts and values conflicts.
5. **I5** Domain Compatibility Invariant
   If a property p_2 of a class C is inherited from a property p_1 of a super-class of C, then the domain of p_2 is either the same as that of p_1 or a subtype of p_1 (covariance redefinitions).
   The number of parameters of the signature of p_2 and p_1 has to be the same.
   The default value of an attribute must be an instance of the type that is the domain of this attribute.
6. **I6** Access Compatibility Invariant
   If an entire type t_1 (tuple or non–tuple) of a class C is inherited from a super-class of C, then the visibility of t_2 is either the same as that of t_1 or more visible than that of t_1.
   An attribute p of a class C may have a specific visibility that is either the same as that of the entire type of the class C or more visible.

If a property p_2 of a class C is inherited from a property p_1 of a super-class of C, then the visibility of p_2 is either the same as that of p_1 or more visible as that of p_1.

7. **I7** Class Specification Invariant
An object is an instance of a class determined by its class specification. A class specification requires a class name, and optionally, a class type together with its visibility, a set of class methods together with their visibilities, a set of parent classes (inheritance) and a set of property renamings.

3.2 Rules to Preserve Schema Consistency

To preserve these invariants, O_2 rules are defined which cope with potential conflicts. We address hereafter only those rules that are relevant regarding our examples in Section 3.3, and which are different from the rules solving similar conflicts in *ORION* or *Gemstone*.

- **R1**: rules to preserve the Class Hierarchy Invariant
R1.1. Node deletion
Only those nodes that are leaves classes might be deleted.
A node together with all its subclasses can always be deleted.
R1.2. An edge BA might be defined (A direct super-class of B) unless BA already exists, B is imported or covariance, compatible visibilities, names and distinct identities conflicts appear because of *I4*. Thus A is added in the class inheritance of B, A becoming a direct super-class of B.
R1.3. Acyclicity
A partial ordered sub-typing relationship is defined on the O_2 types (set–inclusion semantics).
A strict partial ordered sub-classing relationship, based on the sub-typing relationship, is defined on the O_2 class types. Thus a class cannot be a super-class of itself.
- **R2**: rules to preserve Distinct Names Invariant
R2.1. If a class C is defined and its name is the same as that of an existing class in the current schema, the new class is selected over that existing unless the invariant *I5* is violated. This is, if the new specification is not a covariant re–definition of the existing C, or if the existing class C is imported, the new class creation is rejected.
R2.2. If an attribute is defined within a class C and its name is the same as that of an attribute of one of its super-classes, the locally defined attribute is selected over that of the super-class.
The same holds for a method.
R2.3 If an attribute p_2 in C is specified with the same name, a compatible signature and visibility as a method p_1 in a super-class of C, then p_2 overrides the definition of p_1 in C.
The same holds if the attribute p_2 is inherited and re-defined in class C (in that case p2 should be of the same origin as p_1 to avoid the automatic renaming).

The reverse case does not hold. A method cannot override an attribute.
This rule preserves as well *I4*.

R2.4. Multiple inheritance

If two or more super-classes of a class C have a property which appears with the same name but distinct super-class origins, the property is automatically renamed in class C, as many time as the property is present among all the conflicting direct super-classes. This is the *automatic renaming*.

This renaming is propagated going down in the class hierarchy from C.

This might occur while adding a property in a class or adding an edge between classes.

This rule also preserves *I3*.

R2.5. Repeated inheritance

If two or more super-classes of a class C have a property with the same name and the same super-class origin, and none of the classes on the different inheritance paths have redefined the property definition, the property is inherited only once from the class origin.

As a corollary, if on one path the property has been redefined, the property has to be locally re-defined in class C.

This rule preserves as well *I4*.

R2.6. Scope of a property

When a property in a class C is changed (visibility, name, type(s)), the changes are propagated to all of the subclasses of C that inherit them unless these properties have been re-defined within the subclass or name conflicts appear.

This does not hold from the method *init*.

This rule preserves as well *I4*.

- **R3**: rules to preserve Distinct Identity Invariant

  R3.1. If a property is created in class C not inherited from one of its super-classes, this property has a new unique distinct identity (pid).

  R3.2. If a property p_2 of a class C is inherited, renamed or not, locally re-defined or not, from a property p_1 of a super-class of C, its identity is the same as that of p_1.

  R3.3. The same holds if the property is inherited, locally re-defined or not, from two or more super-classes of the class C, with the same origin (the latter is a *repeated inheritance* case).

  R3.4. Deleting an edge BA (A direct super-class of B) is not allowed if there is at least one property p_2 locally re-defined in B or in one of its subclasses that is inherited only through this edge from a property p_1 of A or one of its super-classes. The property p_2 has to be deleted first. This avoids having properties p_1 and p_2 with a same pid in two non related (through inheritance link) classes after the schema change.

- **R4**: rules to preserve the Full Inheritance Invariant

  R4.1 A property inherited can be only re–defined through a creation in a subclass (including type and/or visibility and renaming).

Explicit property "renamings" updates work both on inherited and locally defined properties.
"Method body" updates work on inherited properties (it is well known to be unsafe).

- **R5**: rules to preserve Domain Compatibility Invariant
  R5.1. Covariance
  The domain of a property can be generalized or specialized. The domain of an inherited property cannot be generalized beyond the domain of the property origin and specialized beyond the domains re–defined in the subclasses.
- **R6** rules to preserve the Access Compatibility Invariant Since they are not used in the following sections, these rules are not detailed for simplification.
- **R7** rules to preserve the Class Specification Invariant
  These rules are not detailed herein.
  R7.1. Each type name provided in a type specification must be defined before use. If not, the forward reference becomes an *undefined* type.
  This is an interesting O_2 feature which enables incremental design.

Each *HLP* update, u_{hlp}, should preserve the invariants of the schema. Our general policy is to foresee whether the consistency is sound before applying u_{hlp} to a schema. For that purpose, an ordered set of *LLPs* u_i is generated, if any. In intention, we declare u_{hlp}, which is replaced by the equivalent set of basic updates $< u_1, \ldots, u_n >= u_{hlp}$ to attain the target schema s_n from the source schema s_o. Each basic update u_i such as $< u_i : s_{i-1} \rightarrow s_i >$ starts from a consistent schema s_{i-1} and proves the intermediate schema s_i to be consistent. That is, the u_i preserve the invariants.
Hereafter, following complex update declarations, two examples explain how to check and collect information and the way to perform the update through an ordered set of *LLPs*.

3.3 Examples of Methods Preserving the Consistency for HLP Updates

Hereafter, class generalization and class removal in particular, show how to keep a schema consistent, avoiding conflicts and preserving the invariants with the application of well–ordered set of *LLPs*.

Class Generalization: Domain Compatibility and Set of LLPs
When generalizing two classes, the content of the newly created class and its location in the class hierarchy have to be found.
In the first step, we built up the new class type from the source classes, focusing on preserving the Domain Compatibility Invariant, this is the *covariance*.
The type of the class of generalization, deduced from the types of the source classes, is constructed as the lowest super-type (lowest upper bound) of the source types, according to the O_2 sub-typing definition.

Informally, let T_1 and T_2 be the respective types of the source classes c_1 and c_2. The type T_g of the new class c_g is achieved "intersecting" the structure of T_1 and T_2. The operator $\sqcap_t$ is specified for that purpose and summarized in Figure 5, where T_1 , T_2 , t_1 and t_2 are types in the O_2 type set. The cases of the array highlight with different colors, together with certain type definitions, the type being attained through the $\sqcap_t$ operator.

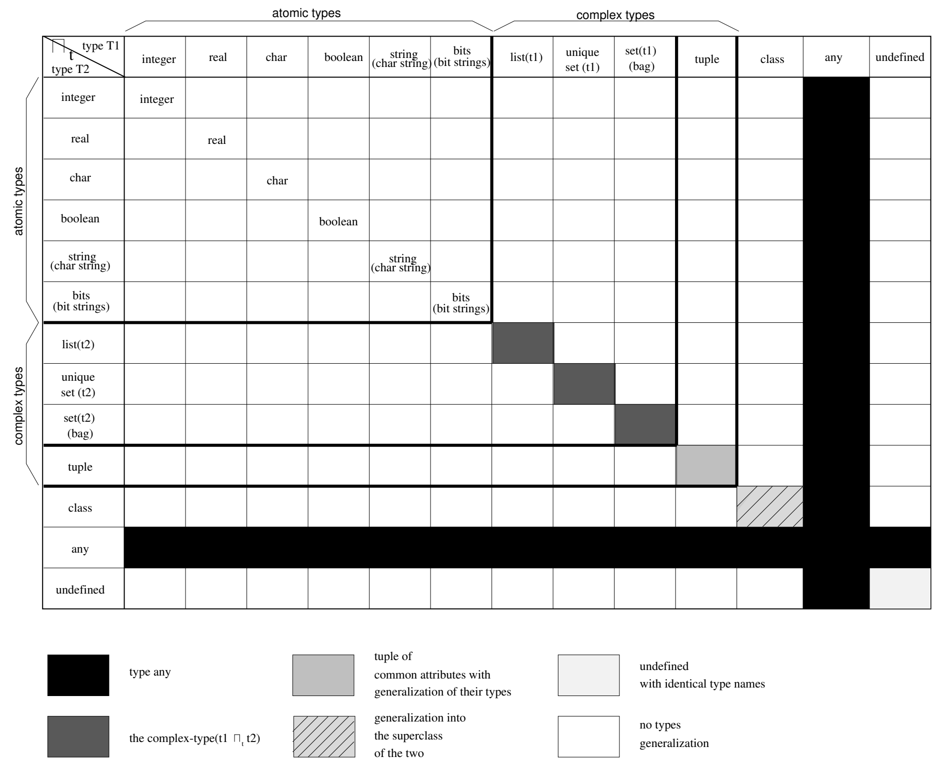

Fig. 5. Class types generalization: operator type intersection $\sqcap_t$.

Accordingly, $T_g = T_1 \sqcap_t T_2$ and the following relationships hold between the new type T_g and its source types T_1 and T_2:
$T_1 \leq_{ty} T_1 \sqcap_t T_2$ and $T_2 \leq_{ty} T_1 \sqcap_t T_2$
where $\leq_{ty}$ denotes the sub-typing relationship.
Moreover, $T_1 \sqcap_t T_2$ is the lowest common super-type of T_1 and T_2. That is, for all c_{cs}, common super-class of the source classes, of a structure T_{cs}, the following sub-typing relationships holds:
$T_1 \sqcap_t T_2 \leq_{ty} T_{cs}$
This definition also is not a sufficient condition to attain a singular generalization. Further arbitrary choices complete these conditions to make the general-

ization unique.
The following four cases then are distinguished regarding the $\sqcap_t$:
$T_1 \sqcap_t T_2 = any$, $T_1 \sqcap_t T_2 = T_1$, $T_1 \sqcap_t T_2 = T_2$ and $T_1 \sqcap_t T_2 \notin \{T_1, T_2, any\}$

This results in different placements of the resulting type $T_1 \sqcap_t T_2$ in the type hierarchy. The latter case only is of interest, when types are *type incompatible* but are *type comparable*. For example T_1 and T_2 have common type information (e.g. type `tuple` with common attributes names) and, in addition, have information that differentiates them. The set of the lowest common super-classes of the source classes is then computed in order to locate the class c_g in the class hierarchy.
The above solution is proved to follow the covariance required by the *Domain Compatibility Invariant* (this has to be restricted in specific cases of renamings in between the lowest common super-classes of the source classes and the source classes).
When all the informations is collected, we can create and insert the class c_g in the class hierarchy following invariant *I7*. A first *LLP* creates a class together with its properties defined locally. The class is located under the lowest common super-classes of the source classeswith creations inheritance links. In the following step, *LLPs* delete inheritance links between the source classes and the lowest common super-classes. According to rule *R3.4.*, this is preceded by the deletion of those properties that are locally redefined in the source classes and their subclasses, and properties which are already factored in the class of generalization.
Two *LLPs* recreate the inheritance links between the source classes and c_g (this is restricted in some cases of intermediate classes existence). Finally, a set of *LLPs* refine properties in the source classes and their subclasses concerned with the factoring.
We showed how to preserve the covariance of the type between the classes and we verified the set of *LLPs*and their order of application enabling the creation of generalization class.

Class Removal: Property Identity and Order of LLPs
Removing a class in the middle of a class hierarchy involves a step where the inheritance links with the subclasses of the deleted class are removed. Because of the rules preserving the *Distinct Identity Invariant*, those of the properties that are concerned with rule *R3.4* have to be stored and deleted. Thereafter, eventually re–creating a property locally, this property receives an identical *pid* if a property with the same name still exists father up the inheritance path, a new *pid* otherwise.
Such properties may eventually solve repeated inheritance conflicts in a subclass of the deleted class. In this case, per subclass, *LLPs* that delete properties are run first, going down in the class hierarchy from the subclass, followed by *LLPs* deleting the inheritance links between the subclasses and the deleted class.
Hereafter, the option *reconnection* to the super-class(es) of the deleted class is assumed to be set. The re–definitions of the properties solving repeated inheritance are re–created in a specific order to avoid repeated inheritance conflicts, following the rule *R2.5.*. The order is bottom–up/transversal in the class hierarchy.

These methods take part in the maintenance of invariants *I2*, *I3* and *I4* while deleting a class. They are used in other *HLPs* algorithms and highlight the importance of the *LLPs order*.

4 Conclusion and Future Work

In this paper we presented new facilities for enhancing schema evolution that allow to change object database schemas in a declarative manner. We discussed issues on implementation on top the O_2 database.

At this stage, none of these new primitives are implemented in current commercial ODBSs. Related works usually address this topic through generalization, specialization or aggregation considerations [29, 17, 21, 28], general schema change management (widely addressed, [31]), view management [24] and schema change simulation through views [23, 7]. Generic and complex update operators for types are developed for this purpose. They are used to change, to extend and to integrate real or virtual schemas and bases.

The current available *HLPs* can be used interactively through an O_2 shell or a GUI. We plan to extend the *HLPs* with default behaviors and to focus on imported classes. This latter point would enhance the capacity for software reuse and schema integration.

Due to limited space, we did not address the notion of database update following a schema change. Database update is currently limited to those updates that are covered by the *conversion* and *migration functions* found in [13] and [12] and implemented in O_2. Readers interested in this aspect are referred to [8] for more details where it is explained how to perform data update following a class deletion.

As for implementation, prototypes were demonstrated at CEBIT 95 and BDA 95.

Concrete evaluation of advantages gained while designing with these primitives can be measured. The current implementation of the *HLPs* allows the designer to set options that register how many "equivalent" *LLPs* updates are called while performing an *HLP* update and what these *LLPs* are. We intend to use this reporting facilities together with specification of new metrics to evaluate schema change costs, including the database updates [14].

Since we consider simulation of schema change through views [7, 2] an important related issue for enhancing schema change, *HLPs* facilities have another potential application.

References

1. M. Adiba and C. Collet. *Objets et bases de données. Le SGBD O_2*. Traîté des Nouvelles Technologies, Série Informatique. Hermès, Paris, 1993.
2. S. Amer-Yahia. Mise à Jour des Données dans les Vues Objet. Master's thesis, Université Paris Dauphine, September 1995.

3. F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System — The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.
4. J. Banerjee, H.J. Kim, W. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object–Oriented Databases. In Umeshwar Dayal and I.rv Traiger, editors, *ACM–SIGMOD '87 Conference on Management of Data, Conference Proceedings*, pages 311–322, San Francisco, California, February 1987. ACM Press.
5. M. Blaha, F. Eddy, W. Premerlani, W. Lorensen, and J. Rumbaugh. *Object Oriented Modeling and Design*. Prentice–Hall International Editions, 1991.
6. G. Booch. *Object Oriented Design with Applications*. Benjammin/Cummings Publishing Co., Menlo Park, California, 1991.
7. P. Brèche, F. Ferrandina, and K. Kuklok. Simulation of Schema and Database Modification using Views. *In Proceedings of the 6th International Conference and Workshop on Database and Expert Systems Applications, Conference Proceedings*, September 1995.
8. P. Brèche and M. Wörner. How to remove a class in an object database system. In *In Proceedings of the 2nd International Conference on Applications of Databases, ADB '95*, San José, California, December 1995.
9. M. Christerson, I. Jacobson, P. Jonsson, and G. Òvergaard. *Object–Oriented Software Engineering. A Use Case Driven Approach*. Addison Wesley, 1992.
10. C. Delcourt and R. Zicari. Consistency Checker (ICC) for an Object Oriented Database System. In *Proceedings of ECOOP'91, European Conference on Object–Oriented Programming*, pages 97–117, Geneva, Switzerland, July 1991. Springer Verlag.
11. S. Even and M. Sakkinen. The safe use of polymorphism in the o2c database language – 5/94. Technical report, University of Frankfurt, FB 20 – Datenbanken und Informationssysteme, University of Frankfurt, Kettenhofweg 135 , 60325 Frankfurt am Main - Germany, March 1994.
12. F. Ferrandina, G. Ferran, J. Madec, T. Meyer, and R. Zicari. Database Evolution in the O_2Database System. In *Proceedings of the 21st International Conference on Very Large Databases*, September 1995.
13. F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 261–272, Santiago, Chile, September 1994. Morgan Kaufmann.
14. F. Ferrandina, T. Meyer, and R. Zicari. Measuring the Performance of Immediate and Deferred Updates in the Object Oriented Database Systems. In *OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, Austin, Texas, October 1995.
15. Inc. Itasca Systems. OODBMS Feature Checklist. Rev. 1.1. Technical Report TM–92–001, Itasca Systems, Inc., December 1992.
16. A. Kemper and G. Moerkotte. *Object Oriented Database Management. Applications in Engineering and Computer Science*. Prentice-Hall, 1994.
17. W. Klas, E.J. Neuhold, and M. Schrefl. Metaclass in VODAK and their Application in Database Integration. Technical Report P-90–09, GMD-IPSI, GMD-IPSI, Integrated Publication and Information Systems Institute. Dolivostr. 15, D-6100 Darmstadt, Germany, Sep 1990.
18. S. Mellor and S. Shlaer. *Object Lifecycles. Modeling the World in States*. Yourdon Press Computing Series. PTR Prentice–Hall, 1992.

19. O2 Technology. *O2C Reference Manual, Version 4.5.* O2 Technology, Versailles, France, November 1994.
20. O2 Technology. *The O_2C Reference Manual. Version 4.5., Released June 1994.* O2 Technology, Versailles, France, June 1994.
21. C.H. Pedersen. Extending Ordinary Inheritance Schemes to Include Generalization. In N. Meyrowitz, editor, *Proc. OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object–Oriented Programming Systems, Languages and Applications*, volume 24(10), pages 407–417, New Orleans, Louisiana, October 1989. ACM Press.
22. D.J. Penney and J. Stein. Class modification in the Gemstone OODBMS. In *Proc. OOPSLA'87, ACM SIGPLAN Second Annual Conference on Object–Oriented Programming Systems, Languages and Applications*, pages 111–177. ACM Press, October 1987.
23. Y.G. Ra and E.A. Rundensteiner. A transparent object–oriented schema change approach using view schema evolution. In *Proceedings of the International Conference on Data Engineering*, March 1995.
24. E.A. Rundensteiner. Multi View: A Methodology for supporting Multiple View Schemata in Object–Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 187–198, Vancouver, Canada, August 1992. Morgan Kaufmann.
25. B. Schiefer. *Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen.* Thesis, FIZ, Forschungsbereich Datenbanksysteme. DBS. Forschungszentrum Informatik. Universität Karlsruhe. Haid-und-Neu-Str. 10-14, D-76131 Karlsruhe, Dec 1993.
26. M.H. Scholl and M. Tresch. Meta Object Management and its Application to Database Evolution. In *Proc. 11th International Conf. Entity–Relationship Approach*, pages 299–321. Springer LNCS 645, Karlsruhe, Germany, October 1992.
27. M.H. Scholl and M. Tresch. Schema Transformations without Database Reorganization. *ACM SIGMOD Record 1993*, 22(1):21–27, March 1993.
28. A. Siebes and C. Thieme. Schema Integration in Object–Oriented Databases. In *Advanced Information Systems Engineering. Proceedings of the 5th International Conference, CAiSE'93*, number 685 in LNCS. Springer–Verlag, Paris, June 1993.
29. J.M. Smith and D.C.P. Smith. Database abstractions: Aggregation and Generalization. *ACM Trans. on Database Systems*, 20(6):105–133, June 1977.
30. GOODSTEP Team. The GOODSTEP Project: General Object–Oriented Database for Software Engineering Processes. In *Proc. of the Asia–Pacific Software Engineering Conference, Tokyo, Japan*, pages 410–420. IEEE Computer Society Press, 1994.
31. R. Zicari. A framework for schema updates in object-oriented database system. Technical Report 90-025, Politecnico di Milano, Dipartimento Di Elettronica, 1990.