HOW TO REMOVE A CLASS IN AN OBJECT DATABASE SYSTEM

P. Brèche M. Wörner

Datenbanken und Informationssysteme (DBIS)
Fachbereich Informatik - FB 20. J.W. Goethe-Universität
Kettenhofweg 135. 60325 Frankfurt am Main, Germany
{breche,mw}@dbis.informatik.uni-frankfurt.de

ABSTRACT

In schema design for object database systems (ODBS) a designer requires advanced facilities to change a schema through its life cycle. Removing information embedded in a complex class hierarchy is one of these advanced schema update primitives. Consequently to such a schema change, databases might need updates, guided by designers' requests. Most of the ODBSs disallow class deletions or restrict it to leaf classes without objects or lose objects after the schema update so far. The main contribution of this paper is to present a declarative primitive, *Remove*, which copes with these requirements.

For prototyping purposes, this article focuses on such a facility implemented as an application on top of the commercial ODBS O_2.

INTRODUCTION

To provide a schema designer with flexibility is one key feature for ODBS which is required by engineering design applications, especially in dynamically defining and modifying a database schema (changes on the fly). In an object environment, this means automating a whole update mechanism and interacting with the designer in case of complex changes. Moreover, schema update operations applied onto schemas have to cope with real design processes i.e. to provide different levels of abstraction, for instance adding a property as well as generalizing classes or removing classes in the middle of a class hierarchy. At the same time, it should facilitate propagation of update operations onto the database.

This article concentrates on a new schema update which *removes* a class in a class hierarchy, and propagates the change onto the database.

So far, commercial databases do not provide such a facility.

Once the semantics of the *remove* is fixed, its application to a *consistent* source schema should result in a new *consistent* schema. Strictly relating to the schema, preserving consistency depends on consistency definitions of the underlying ODBS and on the rules specific to the ODBS to maintain consistency. Moreover, consistency of other elements relying on the schema have to be maintained. Potentially, a schema update might corrupt the database(s) instance(s) as well as view(s) based on the schema and application programs running against the database.

At this point, the designer has to fill the gap between the functions currently provided by the ODBS environment and change requirements. This designer usually spends time finding, grouping and ordering together the appropriate basic primitives, from now on called Low Level Primitives (*LLPs*), to perform the desired update. To do it means to be aware of consistency definitions of the ODBS model, to find out the right *LLPs* accordingly, to foresee and to solve the intermediate conflicts which might appear. Moreover functions have to be written and run against the base to propagate the structural changes to the data. In addition, views and application programs have to be checked, eventually changed and re–compiled to run on top of the new schema/base. All these maintenance operations are time expensive and error prone.

The new declarative complex schema change primitive *remove*, from here on called High Level

Primitive (*HLP*) is developed to free the designer from performing mundane tasks.

After specifying the *semantics* of the primitive, *rules* are defined which preserve *consistency* of resulting schemas. Together with rules which maintain database consistency, this is the main contribution of this paper.

The work presented in this paper is part of the broader GOODSTEP project [18] under the ESPRIT III program. The goal of GOODSTEP is to enhance and improve the functionality of an object database system to yield a platform suited to applications such as Software Development Environments (SDEs). The baseline of the project is an existing European commercially available object database system O_2 [20, 19]. Thus, the data model, the implementation of the *remove* primitive and the *LLPs* are based on the current O_2 release [19].

The rest of the paper is organized as follows. First section introduces the *remove* strategy with a motivating example. The following section details this strategy and focuses on preserving schema consistency. We show after how to maintain the database consistency consequently to a *remove* update, using conversion and migration functions. A state of the art gives an overview on policies for class deletion implemented in current commercial ODBS and related work. Finally the last section concludes on concrete expectations from this work and future developments.

REMOVE A CLASS: MOTIVATING EXAMPLE AND GENERAL DEFINITIONS

Current ODBSs do not allow the deletion of a class (together with its instances) located in the middle of the class hierarchy, performing automatic class hierarchy and base restructuring. Hereafter, the *remove* primitive is introduced to fulfill this requirement. We are concerned with the instances of the removed class proposing a mechanism which triggers conversion and migration functions on objects.

An example first introduces our purpose to be later generalized in order to cover designer requirements.

Example

Consider the schema shown in figure 1 as part of a schema used in a carrying agency, modeling different kinds of vehicles: private vehicles and commercial vehicles.

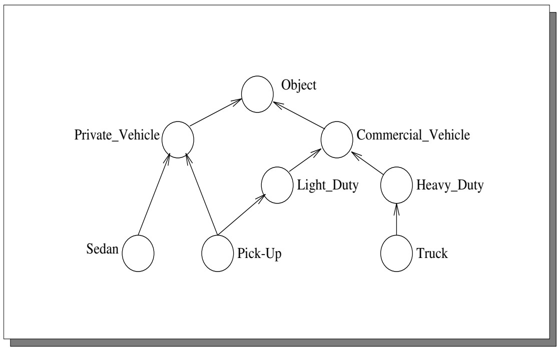

Figure 1: Carrying agency: source schema

Assume now - because of a change of law - light commercial vehicles (those with a weight between 2.5 and 7.5 tons) are viewed as private vehicles. This implies the deletion of class *Light_Duty*. To keep specific informations about this kind of vehicles means to locally re–define properties of class *Light_Duty* into class *Pick-Up*. Properties are propagated to the sub-class of the deleted class. Figure 2 shows the resulting schema.

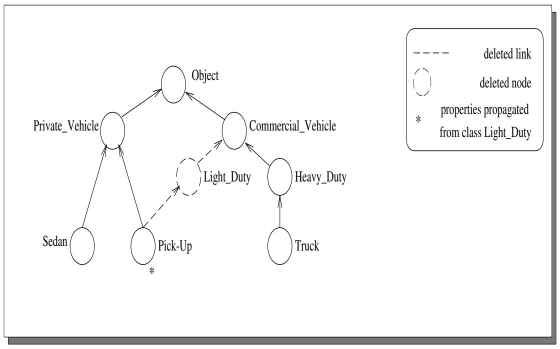

Figure 2: Carrying agency: target schema after deletion of class *Light_Duty*

After the deletion of class *Light_Duty*, class *Heavy_Duty* might be useless since a specialization of commercial vehicles is no longer needed. Therefore the designer deletes class *Heavy_Duty* too. Since trucks are still commercial vehicles, class *Truck* has to remain a sub-class of class *Commercial_Vehicle*. Thus, our designer decides to connect class *Truck* to class *Commercial_Vehicle* after the deletion of class *Heavy_Duty*. That is, class *Truck* is *reconnected* to class *Commercial_Vehicle*. Moreover, the locally defined properties of class *Heavy_Duty* are propagated to class *Truck* to preserve, as far as possible, information coming from the previous information model. Figure 3 shows the target schema reached after such a deletion of class *Heavy_Duty*.

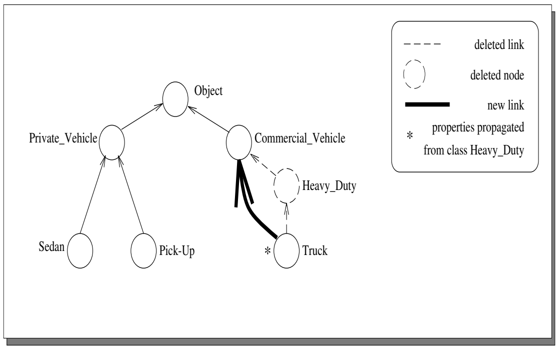

Figure 3: Carrying agency: target schema after deletion of class *Heavy_Duty*

In the former example, a pick-up is no longer a commercial vehicle, thus no link between class *Pick-Up* and class *Commercial_Vehicle* is required. In the latter example, a truck is still a commercial vehicle. Thus, the reconnection of class *Truck* to class *Commercial_Vehicle* is convenient.

<u>General strategy</u>

Let C_{del} be a class located in the middle of the class hierarchy, that is, with super-classes and sub-classes. In addition, C_{del} has specific features specified as local properties in the class. C_{del} could also be found in other classes of the class hierarchy as an aggregation. For simplicity in the general strategy, we focus hereafter on points concerned with the inheritance problems only.

Class deletion can be classified in a first approach as follows:

- delete a leaf class: since this is a *LLP* which is supported by most of the ODBS, we do not detail it.
- delete a class in the middle of the hierarchy: this can be performed either by deleting all the sub-classes of the deleted class or by preserving the sub-classes. The former case matches the deletion of a leaf class (starting from the bottom of the hierarchy) and is therefore not addressed. Only the latter case is described in the sequel.

Thus, assume the class C_{del} has to be deleted. The designer wants to retain specific information of the class C_{del} in its sub-classes. This information was inherited before through C_{del}. To be preserved, this information should be locally defined after in the direct sub-classes of C_{del}. To do it, the designer has to *propagate the properties* which where locally defined in C_{del}.

The designer could decide also that the sub-classes of C_{del} should also keep the information coming from the super-classes of C_{del}. Another designer more concerned with software re-use agrees, arguing that the classification expressed by the class hierarchy should remain "similar" after the class deletion. Thus, the direct sub-classes of C_{del} have to be reconnected to its direct super-classes. This is called the *reconnection.*

The examples of the previous Section point out these different options which might be required once deleting a class. This can be summarized with the willingness of preserving or losing information captured in class C_{del}. This includes information about static (class attributes) and dynamical (methods) properties as well as locally defined or inherited properties. Thus, deleting a class C_{del}, a designer might choose different options, combining *reconnection* to the super-classes of C_{del} and *propagation* of local properties of C_{del}, as follows:

1. You want the direct sub-classes to be reconnected to the class's parents and the reconnection to maintain C_{del} 's specific properties. Set the options *propagation* and *reconnection* which keeps the information locally defined in C_{del} and inherited through C_{del}. Only the properties locally defined in C_{del} and not redefined locally in the sub-classes are propagated. Each direct sub-class of C_{del} is reconnected to every direct super-class of C_{del}. This might lead to information redundancy but preserves the information.
2. You want to leave the sub-classes as they are. Do not set any options, *no propagation* and *no reconnection.* This loses the information captured by and through C_{del} . The direct sub-classes are not re–connected to the direct super-classes and the properties locally defined in C_{del} are not propagated to its sub-classes. *Dangling* sub-classes are automatically connected to the root class of the hierarchy.

3. You do not want the sub-classes of C_{del} to be reconnected to its class's parents but the sub-classes to maintain C_{del} 's specific properties. Set the options *propagation* and *no reconnection*. This preserves the information which was locally defined in C_{del}, propagating (copying) the properties into the direct sub-classes of C_{del} unless the direct sub-classes redefine them. Information inherited from the super-classes of C_{del} is lost. This might lead to a loss of information and/or to information redundancy.

4. You want the sub-classes to be reconnected to the class'parents without maintaining C_{del} 's specific properties. Set the options *reconnection* and *no propagation*. This loses the information locally defined in C_{del} but does keep the one inherited: the 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}.

The corresponding synopsis of the **Remove** primitive is expressed as follows:

```
remove [-p | -r | -pr | from] [class] Class_name
```

The *-p* option propagates the properties locally defined in C_{del} into the sub-classes, if any, whereas *-r* reconnects all the sub-classes of C_{del} to its super-classes, if any. For completeness, the *from* option is added which allows to remove a class and all its sub-classes.

Moreover, the *HLP remove* command helps the designer during the update phase, displaying the set of classes containing cross references to C_{del} (composite classes and named objects) for further refinements purpose.

Performing such a class deletion comes to construct an ordered set of *LLPs* deleting inheritance links and creating properties taking into account name conflicts, repeated inheritance conflicts, property identity conflicts and covariance conflicts. They are detailed in the following section.

CONSISTENCY AND DETAILED SEMANTICS FOR *REMOVE*

The schema change *remove* is an *HLP*, this is, at the logical level, an atomic schema update. It is implemented as an ordered set of *LLPs*, automatically built up regarding the options set by the user and the conflicts which are foreseen while performing the class hierarchy restructuring.

Next subsection presents the main features of O_2, especially those which are concerned with the preservation of schema consistency. The *LLPs* execution leads to a set of intermediate consistent schemas, solving conflicts which might appear during the restructuring phase. Typical conflicts resolution are given in this section. The second subsection accordingly, details the different options of the strategy highlighting the conflicts to be solved. Finally, the last subsection can present the algorithm which implements the *HLP remove*.

How to maintain consistency

Once the semantics of the *remove HLP* is specified, one has to build up the set of *basic atomic* updates, the *LLPs*, which can carry out the schema transformation. This set of *LLPs* transforms out a source schema into intermediate successive *consistent* schemas, up to the expected target schema. This target schema is the one implicitly required by the user and specified as a shortcut through the declaration of the *HLP*.

To keep a schema consistent two items of interest are highlighted hereafter: how to preserve property identifiers and how to solve repeated inheritance conflicts when deleting a class. We will first summarize roughly some properties of the O_2 data model. Definitions of the O_2 schema invariants and the O_2 rules to preserve them follow.

O_2 data model: main features

O_2 is an object oriented data model with the following main features concerned with schema evolution:

- data might be *values* (instance of types) or *objects* (instance of classes).
- the classes encapsulate the data defined by an underlying class type (tuple type, the C alike atomic types and the constructed types list, set and bag) together with methods which operate on the data.

- a class might inherit from different super-classes.

- the inheritance mechanism is defined on a covariant sub-classing relationship, based on the sub-typing relationship.

- Methods (attribute resp.) with same names can be re-defined in sub-classes. The object determines its own behavior (polymorphism) at runtime (late binding) to discover the actual class of the receiver object and thus to choose the method (attribute resp.).

- a property (method, attribute) is identified uniquely along an inheritance path by its *property identifier* – from now on *pid* – independently from its "external" name (can be renamed).

- an O_2 method is specified by its signature and its body, this is the code executed to perform a service. It can be re-defined in different classes along an inheritance path. This is known to be unsafe in the context of late binding [9].

- import–export: schemas and bases collect definitions and data which might be shared through exported and imported declarations.

- finally, persistence is carried out through naming as opposed to class extension.

Readers interested in other and detailed aspects of O_2 are referred to [3, 2, 15].

Schema consistency

The structural consistency of an O_2 schema is specified through the following constraints, so–called invariants of the database system, to be preserved after each schema update.

These invariants are almost standard [4, 17] and known as **I1** Class Hierarchy Invariant, **I2** Distinct Name Invariant, **I3** Distinct Identity Invariant, **I4** Full Inheritance Invariant, **I5** Domain Compatibility Invariant, **I6** Access Compatibility Invariant (this deals with the visibility of a global type of a class, attribute and method) and **I7** Class Specification Invariant (this is what is specified at the schema design time and reflects the available *LLPs*).

O_2 provides specific rules which cope with potential conflicts to preserve these invariants. For the purpose of limitation, the rules preserving the *pid* and solving *repeated inheritance* conflicts (*RI*) are presented only. The other rules similar to the one found in *Orion* or *Gemstone* are not detailed.

R1 Inheritance link deletion

To delete an edge BA (A direct super-class of B) is not allowed if there exists at least one property p_2 locally re-defined in B or in one of its sub-classes which 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 a schema change.

This rule preserves *I3*.

R2 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 has redefined the property definition, the property is inherited only once from the class origin.

Corollary, if on one path the property has been re-defined, the property has to be locally re-defined in class C.

This rule preserves *I2* as well as *I3*, *I4* and *I5*.

Property identity, repeated inheritance and order of *LLPs*

Removing a class in the middle of the class hierarchy, includes a step where the inheritance links with the sub-classes of the deleted class are removed. Because of the rules preserving the *Distinct Identity Invariant*, those of the properties which are concerned by rule *R1* have to be *stored and deleted*. This causes a first constraint in the order of the *LLPs* application. Afterwards, eventually re–creating locally a property, this property receives the same *pid* if a property with the same name still exists going up along the inheritance path or a new *pid* otherwise.

Moreover such properties might eventually solve repeated inheritance conflicts in sub-classes of the deleted class. In this case, the *LLPs* which delete the properties are run first on each sub-class, going down (in a topological order) in the class hierarchy from a direct sub-class, followed by the *LLPs* which delete the inheritance links between the direct sub-class and the deleted class.

If the option *reconnection* 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 rule *R2*. The order of the creation is bottom–up/transversal in the class hierarchy.

These methods are fundamental in the maintenance of the invariants *I2*, *I3* and *I4* while deleting a class. They are used in other *HLPs*' algorithms as well [6] and highlight the importance of the *order* of the *LLPs* application.

Refinement of the strategy

Hereafter for each option of the strategy, we specify what could occur and the option chosen. This is concerned mainly with conflicts due to inheritance. This is completed by a paragraph on aggregation and naming problems.

- the option *no reconnection* is set. Hereafter, the option *propagation* has no influence.

  Let C_{sup}, C_{del} and C_{sub} be classes of a schema in a sub-classing relationship such as $C_{sub} \prec C_{del} \prec C_{sup}$. While removing class C_{del} without reconnection of its direct sub-classes to the direct super-classes may cause questions regarding where to reconnect these sub-classes. If C_{sub} has several direct super-classes in the source schema, deleting C_{del} just removes the inheritance link between C_{del} and C_{sub} in the target schema. In contrast, if C_{del} is the only direct super-class of C_{sub}, deleting C_{del} makes C_{sub} a potential *dangling* class in the class hierarchy. That is, C_{sub} is no more rooted. By default, since the O_2 schema invariant [6] *Class Hierarchy Invariant* is preserved, C_{del} is automatically reconnected as direct sub-class of the root class *Object*. The following semantics is envisaged to handle case of dangling sub-classes:

  – C_{sub} is reconnected to *Object* as a default behavior.

  – since the sub-classes of C_{del} are semantically "dependent" of C_{del}, the dangling direct sub-classes of C_{del} are deleted as well as their sub-classes.

  – inheritance link(s) is forced; all or some of the direct super-classes of C_{del} might be selected. In figure 4, the target schema 3 illustrates a reconnection of C_{sub} and C_{sup}. Although this policy is contradictory to the *no reconnection* statement, it is assumed that the designer did not foresee it and then acts as a reminder. An interactive dialog with the designer is provided to perform this choice.

  Nevertheless, this reconnection might cause further *repeated inheritance* conflicts which are detailed in the sequel in the context of reconnection.

The designer may choose one of these options stated above through a GUI. The according target schemas are illustrated in figure 4.

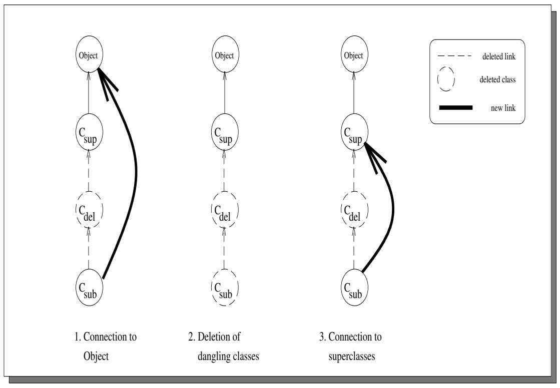

Figure 4: Dangling sub-classes: the different issues.

- the option *reconnection* is set. Once reconnecting the direct sub-classes of C_{del} to its direct super-classes, two specific source schema structures might lead to *repeated inheritance* conflict or *superfluous inheritance* creation. These cases are addressed in the sequel.

  1. how to solve repeated inheritance conflict due to class reconnection:

     In figure 5, C_{del} inherits a property p from, ultimately, a single ancestor class C_{sup}but through two different inheritance paths. This is a case of *repeated inheritance*. O_2 solves the conflict inheriting p from C_{sup}in C_{del}. Moreover, in our figure this property p is redefined in between C_{del} and C_{sup} in C' and C''. This causes a conflict which is solved in O_2 by redefining the property p locally in C_{del}. This avoids the ambiguity of two implementations of essentially the same property.

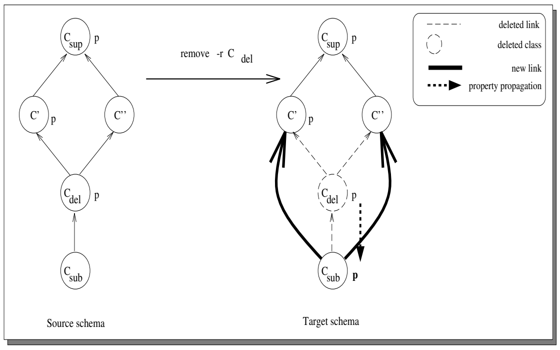

Figure 5: Option reconnection: how to solve repeated inheritance conflict.

In our example, p is not defined in the direct sub-class C_{sub} of C_{del}. Removing the class C_{del} without reconnection would cause any problem. If the option *reconnection* is set, a repeated inheritance conflict appears in C_{sub}. This forces propagation (redefine locally) of p in C_{sub} if the option *no propagation* is set to preserve the schema consistent. Only those properties which solve repeated inheritance are propagated in our strategy. This makes the difference with the option *propagation* where all (!) properties locally defined in C_{del} are redefined in C_{sub}.

2. how to come up with superfluous inheritance

   Propagation of properties has no influence hereafter. The source schema in figure 6 contains classes C_{sup} sub-classed by C' and C_{del}, themselves sub-classed by C_{sub}. Once C_{del} is

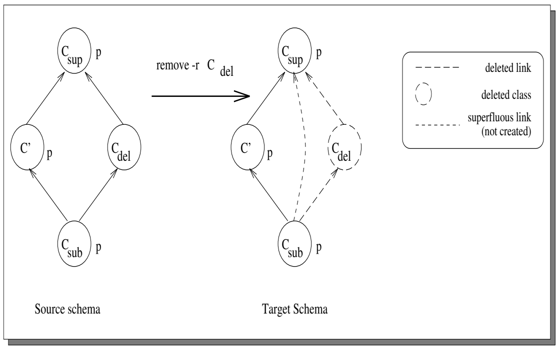

Figure 6: Option class reconnection: the superfluous inheritance.

deleted, an inheritance link between C_{sup} and C_{sub} is created because of the reconnection option. At that time the schema contains a so–called *superfluous inheritance* [14]. C_{sub} inherits properties from both C_{sup} through C' which is a sub-class of C_{sup} and directly from the newly created inheritance link between C_{sub} and C_{sup}. The schema is still consistent but since the transitivity of inheritance is not considered, this leads to poor model design. Our policy avoids in such a case the creation of such a link between C_{sup} and C_{sub}. The resulting schema is the target schema in figure 6.

- the option *propagation* is set.

  All the properties locally defined in C_{del} are locally redefined in the direct sub-classes. The signature and visibility are preserved when propagating. If such a property, recognized by its *property identifier* is already locally defined in a direct sub-class, the property is not propagated to this sub-class.

- the option *no propagation* is set.

  In O_2, it is not possible to have in the same class an attribute and a method of the same name and the same signature. The attribute takes precedence over the method. In figure 7 the method p in super-class C_{sup} is redefined with the name and signature in sub-class C_{del}.

  According to the encapsulation of the object, the behavior of the receiver to the message `obj->p` would change if the object is in the extension of C_{sup} or C_{del}. The former one would execute the method core whereas the latter one would get the value of the attribute. Once deleting class C_{del} with the

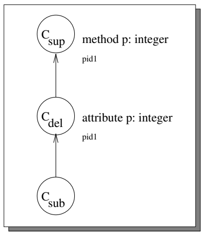

Figure 7: Option no propagation: method–attribute conflict

option *reconnection*, class C_{sub} becomes a direct sub-class of class C_{sup}. After this update, the method would replace the attribute in C_{sub}, executing a method instead of returning the attribute value. The new schema is still consistent but changed semantically and might lead to run–time errors. A warning message is displayed to advice the designer and let him further change the code of the applications running on the schema.

- the option *propagation* and *reconnection* are set.

  Before going into details, the way O_2 governs properties has to be explained. Each class property (attribute or method) is identified internally by a *property identifier*. This *pid* gives an unambiguous way of recognizing a property. Along an inheritance path in the class hierarchy, a property which is locally defined in a class at the specification time with the same name as a property in its direct super-class keeps the same *pid*. This holds also if the property is renamed in a class, as in figure 8 where p from C_{del} is renamed as p' in C_{sub}.

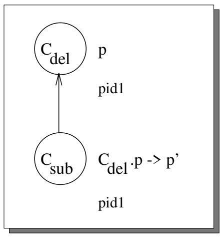

Figure 8: Class renaming and *pid*.

  In sub-classes of C_{sub}, the property p is accessed through its new name p' only.

  In figure 9, p is defined in C' and renamed as p_1 in C_{sub}. The same property is locally defined in C_{del}. They have all the same pid pid_1. At the same time, a new property p is specified in C_{sub}. Thus, in C_{sub}, the property p "takes place" (user point of view) of the inherited property p from C_{del}, but gain a new pid pid_2. Renaming only alters the name of an inherited or locally defined property but preserves the visibility and signature of the property.

  C_{del} is now deleted with the option *reconnection*. The information concerning the renaming is lost at that time although we would like to keep this renaming in C_{sub} and its sub-classes (since p_1 is used to access this property in the code dealing with C_{sub} and its sub-classes). Our strategy is to define p_1 locally in C_{sub}. This leads to a new pid creation in C_{sub}. Further redefinitions or renamings of p_1 in sub-classes of C_{sub} has to be adapted as well.

  In case of p is in a repeated inheritance solved in C_{del}, it is necessary to preserve the name of p renamed as p_1 in C_{sub} as well as to propagate the property of C_{del} with its original name as in figure 9. Property p

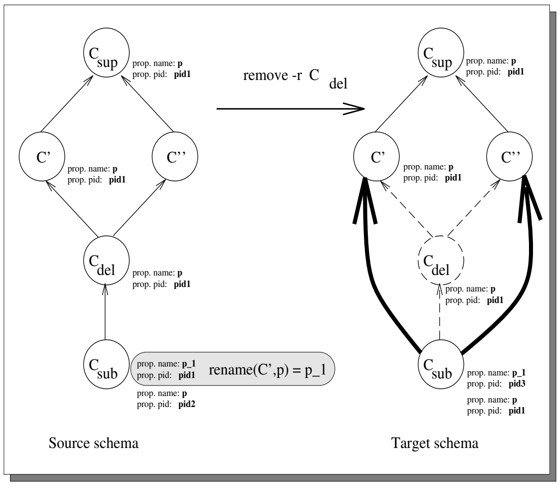

Figure 9: Options –rp: propagation of property with renaming

  is therefore propagated to C_{sub} in order to solve the repeated inheritance.

- to complete the strategy.

  It might happen that the deleted class is referred in other classes as an aggregation or in named objects or values as part of their type specifications.

  Regarding the aggregation, the C_{del} deletion breaks the reference to C_{del}. Since O_2 allows the declaration of a type which is not already known from the system, the type for any definition involving the C_{del} name becomes *undefined*. The type definition of a class C containing such an aggregation with C_{del} becomes *incomplete* and is called a *shadow* class. Moreover, in O_2 *undefined* types are constrained to become, latter at their definition, *class* types. In addition, to preserve covariance, an *undefined* type cannot be subtyped. Accordingly, to delete class C_{del} requires schemas which do not have a class C, in aggregation with C_{del}, which does specialize C_{del} into one of its of sub-classes. Otherwise, this would lead to the inconsistency where class C exists with an attribute[1] becoming *undefined* which is redefined (strictly) in a sub-class of C by a sub-class of C_{del} . By default, the C_{del} deletion is forbidden.

[1]For simplicity, we consider the case of a tuple type for C.

Regarding the named objects, the O_2 persistence is carried out by naming. Once you delete a class definition, the objects instances of the deleted class should be removed from the root(s) of persistence which store them. It requires deleting the root of persistence corresponding to the C_{del} 's extension and the removal of instances of C_{del} if any from the C_{del} super-classes's extensions. Otherwise instances of C_{del} have to be migrated *before* the class deletion. This latter case is detailed in section on database update The objects of roots of persistence containing references to instances of C_{del} become incomplete. Their value for this reference has to be changed to a *nil* value if the corresponding instance of C_{del} was not migrated.

Regarding the methods, application programs, or functions which use C_{del} in their code, the resulting schema inconsistencies are managed according to specific compiler options. They include automatic recompilation of the code, a consistency check issuing warning messages in case of inconsistencies, or disregarding possible inconsistencies.

Broken references can be checked looking at each class and type definition including C_{del} in their description and displaying the information[2]. The compiler options mentioned above are used to trigger warning messages.

Implementing class Remove in O_2

To perform the class removal, the *LLPs* which carry out the "equivalent" change are performed in a specific order taking into account remarks of section on consistency This complete order is given by the following steps:

1. resolve aggregation conflicts,
2. resolve the *repeated inheritance* conflicts in C_{del},
3. delete the properties inherited through C_{del} which are redefined in its direct sub-classes and their sub-classes,
4. delete the inheritance links between C_{del} and its sub-classes,
5. delete class C_{del},
6. if option *reconnection* is set:

   recreate the inheritance links between the direct sub-classes and the direct super-classes of C_{del},
7. redefine the properties of the sub-classes deleted in the second step,
8. redefine the properties of the sub-classes deleted in the first step,
9. if option *propagation* is set:

   redefine the properties which where locally defined in C_{del} in its direct sub-classes.

[2] Since the search is time cost expensive, it is not implemented.

Each action in the previous list maps a set of *LLPs*, e.g *delete property* is enforced by the *LLP*

`delete attribute` *attr_name* `in class` *class_name*.

The following algorithm covers all these phases throughout six main steps.

START Algorithm
if not *from* option **then**

$\mathcal{STEP}$ 0 /* Get the classes having an aggregation with C_{del}. Check the specialization of C_{del} along an inheritance path */ **{**

if C_{del} as an aggregation is specialized going down along an inheritance path **then ABORT remove**. /* means that an undefined type will be specialized with a non compatible type */

if reconnection **then**

$\mathcal{STEP}$ 1 /* Get properties of C_{del} solving RIs. These are candidates for solving RIs in sub-classes of C_{del} after a reconnection and/or propagation */ **{**

- **if** C_{del} is the lowest class of a repeated inheritance (RI) **then**
  {
  - determine those properties locally defined in C_{del} which are solving repeated inheritance conflicts (RICs). Store those properties in a list *props_solving_RICs*.
  - /* solve conflict: is it possible to propagate all properties stored in *props_solving_RICs* to solve RI */
    if a direct sub-class of C_{del} has a property with the same name but from a different origin **then ABORT remove**. /*

means that the properties have different internal identifiers (pid) */

}

- **for each** property of *props_solving_RICs*
{
  - /* collect sub-classes with further RIs caused by reconnection and forced propagation */
    if property has to be redefined in some sub-classes of C_{del} to solve RICs occuring after the reconnection **then** store name of the sub-classes and the property's name in a list *where_what_to_redef*. The order of the properties in *where_what_to_redef* reflects the order of later redefinitions. This is, a bottom up application of further LLPs.

  }

}

/* from now on the deletion is assumed to be allowed */

$\mathcal{STEP}$ 2 /* delete inheritance links between C_{del} and its direct sub-classes */
For each direct sub-class of C_{del} a list *redef_after_relink* is defined in which the type description (in case of attributes) or signature (in case of methods), the pid and the access right of these properties can be stored. The order of the properties in *redef_after_relink* reflects the order of further redefinitions through LLPs used in $\mathcal{STEP}$ 4.
for each IL between class C_{del} and one of its direct sub-classes
{

- /* solve problems of delete an inheritance link (IL) where properties are redefined in subclasses */
  for each locally defined property of a subclass of C_{del} which loses its connection to its origin if the IL is deleted (from now on it is said that the property is no longer "visible")
  {
  - store the property in *redef_after_relink* belonging to the current direct sub-class of C_{del}.
  - delete the property.

  }
- delete the IL. Those of the sub-classes of C_{del} with C_{del} as a unique direct super-classes are marked (these are the potentially *dangling* classes).

}

$\mathcal{STEP}$ 3 /* reconnect the direct sub-classes to the direct super-classes */
for each pair of direct sub-class/direct super-class of C_{del}
{

- **if** reconnection **then**
  {
  - create an IL between these classes /* is done as the first step to avoid problems with the pids of the properties to be redefined. */
  - drop the IL between the marked subclasses and Object if any

  }
  $\mathcal{STEP}$ 4
  /* property redefinitions in the sub-classes. First, redefine the properties deleted because of previous IL deletions, then create the properties visible at this stage which solve new repeated inheritance. This avoids RIs when relinking */
- Redefine all the properties stored in list *redef_after_relink* belonging to the direct subclass. Their description is taken from *redef_after_relink*. /* define attributes to solve RICs */
- **if** reconnection **then** redefine the properties of *where_what_to_redef* now visible again in the direct sub-class and its sub-classes. Their description is taken from the properties with the same name defined in C_{del}.

}

$\mathcal{STEP}$ 5 /* propagation: new property refinements in the sub-classes */

if propagation **then** {

- **for each** direct sub-class of C_{del}
  {
  - propagate all the locally defined properties of C_{del} not already locally defined in the direct sub-class. These properties are redefined with the same name, access rights and types or signatures. In case of no reconnection the pid of the newly created property is different from the pid of the property to be propagated.

  }

}

$\mathcal{STEP}$ 6 /* Final Deletion of C_{del}*/
delete class C_{del}

END Algorithm

DATABASE UPDATE AFTER CLASS DELETION

O_2 does not provide default class extensions. Class extensions may be built through roots of persistence, so–called *names* in O_2. From now on, the O_2system is assumed to simulate local extension.

Data may have been created and accessed under control of the schema which has to undergo a change. These data must not be lost or corrupted otherwise by the schema change. When deleting a class in the middle of the hierarchy two kinds of alterations, eventually combined, might occur:

- types of sub-classes of C_{del} changed in the target schema: for instance they do not inherit attributes of C_{del} or of its superclasses if the option p and/or r is not set. Consequently, the objects of these classes should be restructured to map these types. Moreover, because the *HLP remove* is implemented as an equivalent set of *LLPs*, intermediate schemas could contain objects not conformed to intermediate class types. These intermediate non significant changes should be "masked".
- objects of the deleted class are lost by default: they could be eventually migrated to sub-classes of the deleted classes according to the wishes of the schema designer.

To summarize, deleting a class requires to i) *adapt the format* of the existing data in order to conform to the new type definitions in the resulting schema and ii) to have the capability of *migrating* objects.

Conversion and migration functions

The current developments of O_2 provide a user, as a consequence of schema modifications, with two facilities: *conversion functions*[3] (c.f.) and *migration functions* (m.f.) [12, 11, 10].

Henceforth, the notion of *schema change* means *atomic* schema update. All the O_2*LLPs* are atomic.

[3] Other object database systems using a similar approach are GemStone [8] and ObjectStore [7]. Default conversion function capabilities exist in the current O_2release.

The *c.f.* refers to the possibility for an object to conform to its new type consecutively to a structural class update. It takes as input parameters the old and new schema class definitions and when executed transform the object of the database to conform to the new schema. It covers deletion, addition, modification of attributes in a class. A default *c.f.* is automatically built checking the differences between the *initial* type of a class and its *final* type after the schema change. The comparison performed on the attributes is carried out regarding the *pid* and not the *name* of the attribute. This is important for our further discussion. A user can specify its own *c.f.*, overwriting a default *c.f.*. To perform a user–defined *c.f.*, a designer needs to associate it to a schema change within a so–called *modification block* (see [10]). Each *modification block* concerns one class only and potentially its subclasses through an inheritance mechanism. It is performed in a *two phase* step: the schema change phase where the schema transformation is declared and the *c.f.* phase where the conversion function(s) is written. The *c.f.*(s) is logically executed at the end of the modification block statement.

A *m.f.* refers to the possibility for an object to change its class during its lifetime. This kind of migration is particularly useful when deleting a class of which the designer wants to retain the objects. In this case, *before* the deletion of the class, objects have to be migrated appropriately. Migration of objects is obtained associating a migration function to a class and is responsible for migrating part of the objects of the class extension to sub-classes only. Objects migration is restricted to sub-classes, because of well–known constraints of covariance based data models as O_2. Otherwise, it might lead to run-time type errors. As opposed to a *c.f.*, a *m.f.* is declared as a stand–alone function, like a common O_2 function. It has impact on one class only. Such a function is logically executed at the end of the function body declaration (see examples in [10]).

The real execution of the *c.f.* and *m.f.* is done following a *lazy database transformation* strategy [12].

An atomic update triggers automatically a database update, e.g. with trigger mechanism, according to the *kind* of schema update performed. A classification of the taxonomy of basic schema updates, regarding the changes on the static part of a class and on the class hierarchy, induces the specific propagation on data, roughly as follows:

- type modification: objects have to conform to the new type specification,
- type creation: objects <u>may</u> migrate to this new type specification,
- type deletion: objects <u>may</u> migrate to other existing definitions (sub-classes).

<u>How to use c.f. and m.f. mechanism for class deletion</u>

Since at the logical level, the *remove* update is atomic, any kind of data update due to type changes occurring during class removal should be carried out *after* the complete hierarchy restructuring. The objects should be therefore converted according to their final types after the restructuring. In addition, *remove* is implemented as a set of specific ordered *LLPs*.

Let $\mathcal{S}$ be the set of all consistent schemata and (S,S') $\in \mathcal{S} \times \mathcal{S}$. Let $\mathcal{DB}_S$ and $\mathcal{DB}_{S'}$ the set of all potential states a database might have with these schemata (resp.) and be $(\sigma, \sigma') \in \mathcal{DB}_S \times \mathcal{DB}_{S'}$ the database states of S and S'.

A schema/database evolution is represented as a pair $< u : S \rightarrow S', r : \sigma \rightarrow \sigma' >$ where u is an *atomic* schema update mapping S into S' and r is the consequent reorganization, mapping the state σ of S into σ' of S'.

Here, u is instantiated with *remove* and r is an according set of *c.f.* and *m.f.* (default or not) updating consequently the data. At the implementation level, the *HLP* u is replaced by a *valid* ordered sequence of n *LLPs* u_i such as the left side of schema/database evolutions: $< u : S \rightarrow S', \ldots >$ becomes

$< (u_1 : S_1 \rightarrow S'_1, \ldots, u_i : S_i \rightarrow S'_i, \ldots, u_n : S_n \rightarrow S'_n), \ldots >$

where S_1 is the source schema S and S_n the target schema S'.

We decompose the reorganization r into the set of intermediate reorganizations r_i mapping with schema updates u_i such as

$< u_i : S_i \rightarrow S'_i, r_i : \sigma_i \rightarrow \sigma'_i >$

This allows us to understand the data reorganization r during a class removal.

Hereafter, we take again our example of the *Carrying Agency*, from another point of view, slightly modifying the class hierarchy: it starts from class Vehicle (Object is omitted), classified as *Land_Vehicle* and *Water_Vehicle*, further refined their respective sub-classes. The attributes of the classes are detailed for our purpose as in figure 10. Two kinds of intermediate schema

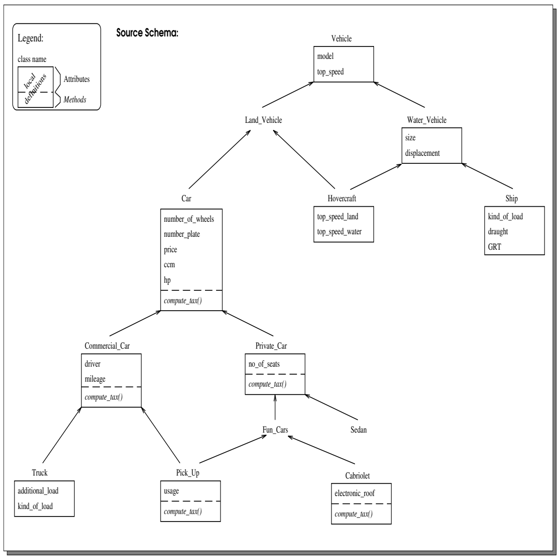

Figure 10: Hierarchy of vehicles: source schema.

changes responsible for data loss during the deletion process are addressed hereafter:

- the successive intermediate updates on types of sub-classes of C_{del} which leave unchanged the type of these sub-classes before and after the process,
- the intermediate updates which change the *pid*s of attributes of sub-classes of C_{del} along the deletion process.

As an example let us consider the three classes *Car*, *Commercial_Car* and *Truck*, and *objectT* an object of class *Truck*. Each of these classes has the attribute *price*, inherited or not, with the same *pid* at time t_0. *objectT* has the value `150.000` for *price*.

The class *Commercial_Car* is removed from the hierarchy with the options *no propagation* and *reconnection*. An "equivalent" set of *LLPs* is triggered to carry out this class deletion. At time t_1, a *LLP* u_1, "`delete inherit` *Truck Commercial_Car*," first deletes the inheritance link between *Truck* and *Commercial_Car*. The attribute *price* is no more inherited in class *Truck* and the class type *Truck* loses *price*. Since every time a class type changes, a default *c.f.* is generated which converts the concerned objects to their current type, *objectT* consequently loses the data `150.000`. At time t_2, class *Commercial_Car* is deleted whereas at t_3 the *LLP* u_3, "`create inherit` *Truck Car*", reconnects *Truck* to *Car*. The attribute *price* becomes inherited in *Truck* and a default *conversion function* is generated which converts *objectT* again, initializing *price* with the default value `0`. This is what happens during the intermediate r_i reorganizations as illustrated in Figure 11 if nothing is done. Since the type change is transparent for a user, the initial value of *price* at t_0 should be preserved along the intermediate changes to appear identical at the end of the schema restructuring.

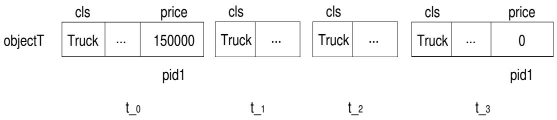

Figure 11: Evolution of Truck's object: the attribute *price*

To get the right data reorganization, the deletion of the inheritance link at time t_1 with the update u_1 ($\mathcal{STEP}$ 2 in algorithm of section on implemention) is performed in a *modification block*. This modification block concerns class *Truck* and contains the update u_1 deleting the inheritance link between *Truck* and *Commercial_Car*. Moreover, temporary attributes are created which store the values of the inherited attributes. A conversion function handles it, copying the value of lost attributes into temporary attributes. For instance, , attribute *price*, is transformed out into *xxx_price* as follows:

```
begin modification in class Truck;
/* update u_1 */
delete inherit Truck Commercial_Car;
/* creation of temporary attributes */
create attribute xxx_price in class Truck;
...
conversion functions;
  conversion function convert_truck1(
    old:tuple(model: string,
            top_speed: integer,
            ...
            price: integer,
            ...
            kind_of_load: integer))
  in class Truck
  {
    ...
    self->xxx_price= old.price;
    ...
   };
end modification;
```

After intermediate updates not concerned with class *Truck*, the inheritance link is created again, at time t_3, with the update u_3, ($\mathcal{STEP}$ 3 in the algorithm of section implementation). It is also embedded in a *modification block*. A conversion function copies the values of temporary attributes in corresponding newly inherited attributes (whereas the temporary attributes are removed) as follows:

```
begin modification in class Truck;
create inherit Truck Commercial_Car;/* update u_3 */
/* creation of temporary attributes */
delete attribute xxx_price in class Truck;
...
conversion functions;
  conversion function convert_truck3(
    old:tuple(xxx_model: string,
            xxx_top_speed: integer,
            ...
            xxx_price: integer,
            ...
            kind_of_load: integer))
  in class Truck
  {
    ...
    self->price= old.xxx_price;
    ...
   };
end modification;
```

Figure 12 figures out these data reorganizations which "inhibit" all the intermediate data changes while "inverse" updates occur.

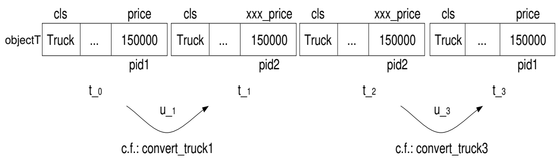

Figure 12: New evolution of Truck's object: the attribute *price*

Let us now start removing class *Commercial_Car* from the hierarchy with the options *propagation*, *reconnection* within a *modification block*. The attribute *driver* is considered hereafter. The *pid*s of the attribute are identical in class *Commercial_Car* and *Truck* at time t_0 in the source schema. At time t_1, a *LLP* deletes the inheritance link between the two classes. *driver* is no more inherited in class *Truck*. At time t_2, class *Commercial_Car* is deleted whereas at t_3, *Truck* is reconnected to *Car*. At time t_4, a *LLP* "**create attribute** *driver* **in class** *Truck*" propagates the attribute *driver* in class *Truck*. Since the attribute does not exist along the inheritance path going down through *Truck*, it receives a new *pid*. The final type of class *Truck* does contain the attribute *driver* but with a new *pid*. Since the difference of class types regarding the attributes is checked also regarding their *pid* [10], in any case, a default *c.f.* is generated which converts *objectT* and gives it a default value. Figure 13 illustrates these data reorganizations.

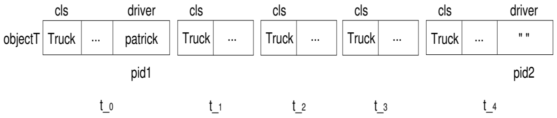

Figure 13: Evolution of Truck's object: the attribute *driver*

Again, although the class type changed from the system perspective, it remains identical from the designer's point of view who checks only the external information available, that is, the name of the attribute. To preserve the values along such an intermediate reorganization $r_i : \sigma_i \rightarrow \sigma_i'$, due to schema updates u_i which change the *pid* of "identical" attributes, we need to check all of those attributes which changed their *pids* without changing their names[4]. The initial values at time t_0 are copied as final values at t_4 with the same method as above. Temporary attributes are created and further deleted at times $t1$ and $t4$ (resp.) while generated conversion functions copy the values. Regarding the example, this process would be applied on attribute *driver* with the value *patrick*.

More generally, this method holds when deleting a class C_{del} for all the attributes in the sub-classes C_{sub_i} of C_{del} which are temporary lost during the process. These attributes are redefined after the schema change, identically or with a new *pid*. The following cases for attributes of sub-classes are covered:

- attribute inherited from C_{del} , option *propagation* set,
- attribute inherited from a super-class of C_{del} , option *reconnection* set,
- attribute inherited from C_{del} and locally re-defined,
- attribute inherited from a super-class of C_{del} and locally re-defined,
- attribute solving a RI conflict, option *reconnection*,
- previous cases mixed.

[4] It is noticed that renaming does not create problems since the attribute name only changes.

In the algorithm, at $\mathcal{STEP}$ 2, for each sub-class where it occurs, a modification block is created. It contains the schema update causing the removal of the attribute (delete inheritance link or delete attribute resp.). In addition, a schema update creates a temporary attribute while a corresponding *c.f.* performs a copy of the attribute value in the newly created attribute. Objects of the sub-classes can be converted accordingly. Latter in the process, at $\mathcal{STEP}$ 3 and $\mathcal{STEP}$ 4, the deleted attribute appears again (inheritance link creation or attribute creation) in the sub-class. A modification block is created for each concerned sub-class, embedding the update. Similarly, additional schema updates delete the temporary attribute while a *c.f.* copies the attribute value from the temporary attribute to the newly appearing attribute. Objects of the sub-classes can be converted again and recover the initial values of their temporary deleted attributes.

Since *c.f.*s can be *inherited* in sub-classes optimizations are performed.

Accordingly, our complete process includes now a set of modification blocks which "encapsulate" schema updates touching those attribute values of sub-classes which have to be saved. Each modification block addresses one sub-class of C_{del} only.

In the previous examples we used *c.f.*s to solve internal problems converting objects of sub-classes of C_{del} which changed their type classes during the schema restructuring. This was mainly because the *HLPs* are built up as application on top of O_2. No intervention of the designer is required in these extensions.

So far, to remove a class C_{del} leads to the loss of its objects. The user would like to have the possibility of migrating objects of C_{del} to other classes of the schema. For that purpose, the *remove* declaration is extended as follows:

```
remove [-- p|--r|-- pr | from]
[class] Class_name
[ migrate objects to
```

Sub_Class_name [{ *condition* }]
[, to *Sub_Class_name* [{ *condition* }] ...]]

This allows us to migrate objects of C_{del} to its sub-classes $Class\_name_{sub}$ only, according to conditions expressed in { `condition` }. The restriction to the sub-classes of C_{del} avoids well–known possible problems of run–time type errors occurring if objects were allowed to migrate freely to any class in the schema. The filter *condition* contains restricted O_2C code (no method calls, no names).

From now on, the schema of the example is slightly modified, including the attribute *weight* in class *Vehicle*. Thus, removing class *Commercial_car* with the options *reconnection*, *propagation*, the user migrates the objects of *Commercial_car*: those which have a weight greater than `1.5` tons to *Truck* and the others to *Pick_Up*. The corresponding declaration is expressed as follows:

```
remove -pr class Commercial_Car
migrate objects
to Truck {if (self- >weight≥1.5) },
to Pick_Up { if (self- >weight<1.5) }
```

This declaration is handled by generating a *migration function* attached to the class to be deleted. This function is executed first, before the schema restructuring dealing with the class removal. The following *m.f.* is automatically generated to carry out the previous declaration:

```
migration function migrate_commercial_car
          in class Commercial_Car
  {
   if  (self->weight >= 1.5)
          self->migrate("Truck")
   if  (self->weight < 1.5)
          self->migrate("Pick_Up")
  };
```

Logically the migration is executed before the schema restructuring implied by the class deletion. The schema restructuring is then performed within a modification block as a set of *LLPs* together with the automatically generated conversion functions as in figure 14.

Physically, the transformations are performed in the same order at the object *access* time in the *lazily* strategy.

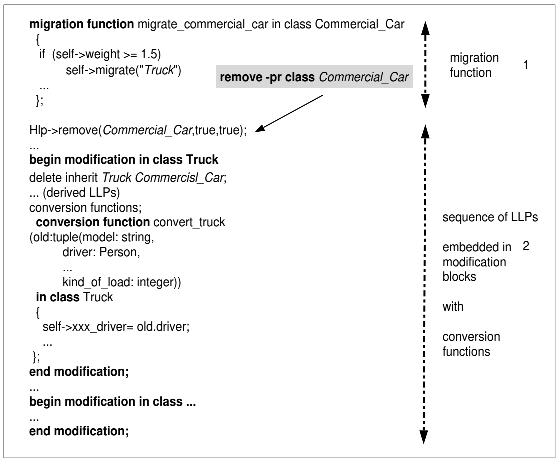

Figure 14: Complete process of data update

RELATED WORK

Hereafter, the commercial ODBSs Itasca/Orion, ObjectStore, O_2 and Versant are under consideration in our study on the *remove*.

Class Deletion in Itasca/Orion

The deletion of a class can be performed interactively through a GUI, simply selecting the class to be removed.

Itasca ([13], page 27 – Drop a Class) allows the deletion of a class located in the middle of a class hierarchy. The instances of the deleted class are *automatically removed*, that is, lost. The sub-classes of C_{del}, if any, are preserved but lose C_{del} as their superclass. If C_{del} was the only super-class of these sub-classes, they gain C_{del}'s super-classes as their immediate super-classes. This means that the sub-classes of C_{del} are *automatically reconnected* to the direct super-classes of C_{del}.

When deleting class C_{del}, the properties locally defined in C_{del} and inherited in the sub-classes, are lost in these sub-classes. If a sub-class of C_{del} loses such a property p that was selected over a conflicting property in another super-class of that sub-class, the sub-class now inherits the alternate definition of p. The alternate property is given by the following rule used by Itasca (ORION resp.) to maintain its invariants: if two or more super-classes of a class C have a property with the same name but distinct origin, the property selected for inheritance is that from the **first** super-class among conflicting super-classes [4]. Consequently, an instance of a sub-class of

C_{del} which loses its present values/behavior for p because of the C_{del} deletion, inherits the default value/behavior under this new definition of p.

If the class C_{del} being dropped is presently the domain of the instance attribute $attr_1$ of some other class (an aggregation with C_{del}), $attr_1$'s domain becomes the first super-class of the class C_{del}.

Moreover, regarding the propagation of changes, when an instance of a class is dropped, all objects that referenced it will reference a non–existent object. This would require further updates of the applications using these objects.

Class Deletion in Objectstore

Schema evolution is not interactive in Objectstore. The designer has to specify in a file the target schema to be reached starting from the source schema. This means that the designer has to be able to envisage the complete change occurring while performing a deletion.

Thus, one might delete a class in the middle of the hierarchy only explicitly specifying the changes occurring in the class inheritance hierarchy ([16], page 351), if any.

Deleting a class C_{del} results in the deletion of all its instances during the instance initialization phase. Pointers and references to objects so deleted provoke an error ([16], page 362).

Class Deletion in O_2

A class deletion can be performed in different ways in O_2: through a class browser GUI, interpreted O_2-commands or a $C/C++$ program.

The current release of O_2 [19] allows the deletion of a leaf class. A class in the middle of the hierarchy can be deleted as well but with all its sub-classes.

The deletion of a class which is referenced by other classes or type definitions is allowed. The class definition is kept as a forward reference and definitively disappears when there are no references to it. The objects of the deleted class become inaccessible in every base governed by the current schema. The other classes, named objects or values using the deleted class in their type specification become *incomplete*. This is, for a class C having an aggregation with C_{del} in $attr_1$, $attr_1$'s domain becomes *undefined* (as opposed to Itasca where it becomes the first superclass of C_{del}) and C becomes incomplete. C is then called a *shadow* class. The designer may further specify a new domain (a class domain only) for that attribute $attr_1$.

Class Deletion in Versant

Versant allows the deletion of a leaf class only. The instances of the deleted class are lost.

CONCLUSION

This paper presented how to handle a class deletion in the middle of a class hierarchy together with its consequences on the database. This is one of the facilities which are under development in the GOODSTEP project for enhancing schema evolution in ODBSs. After presenting a state of the art, we introduced the expectations of designers removing a class in a hierarchy. Further we detailed the semantics of the remove facility, emphasizing how schema consistency definitions are preserved. Together with schema consistency, database consistency is maintained according to designers' specifications.

As far as the implementation is concerned, the primitive *remove* is implemented on top of O_2. A first prototype was demonstrated at CEBIT 95 and BDA 95.

The concrete expectation from this work is first to improve declarativity of such a complex schema change. This can be measured by the number of *LLPs* which perform an update "equivalent" to the *remove* update and database change performed by a designer in such a SE process. The current implementation of the *HLP remove* allows the designer to set options for that purpose. This provides hints for new metrics in SE processes.

The implementation of the database update consequently to schema change is ongoing work, integrating *conversion* and *migration functions* in the *HLPs* framework.

We plan to extend this facility with a default behavior and to cover imported classes. This latter point would provide better software reuse and integration capacity for federated databases.

Moreover, the integration of the *remove* primitive into a view mechanism [1] (similarities with the view command *hide*) is under consideration. This participates to a **C**omputer **A**ided **S**chema **U**pdate tool suitable for change management, currently under development at the University of Frankfurt, including schema change simulation [5].

References

[1] S. Abiteboul, C. Delobel, and C. Souza dos Santos. Virtual Schemas and Bases. In *Proceedings of the International Conference on Extending Database Technology*, March 1994.

[2] Michel Adiba and Christine Collet. *Objets et bases de donnees. Le SGBD O_2.* Traite des Nouvelles Technologies, Serie Informatique. Hermes, Paris, 1993.

[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] P. Brèche, F. Ferrandina, and K. Kuklok. Simulation of Schema Change using Views. In N. Revell and A.Min. Tjoa, editors, *Proc. of the 6th International Conference on Database and Expert Systems Applications*, number 978 in LNCS, London, England, September 1995. Springer Verlag.

[6] P. Brèche and M. Wörner. Schema Update Primitives for ODB Design. Technical Report 07/95, Fachbereich Informatik, Universität Frankfurt, May 1995.

[7] Object Design Inc. *ObjectStore User Guide, chapter 9.* Object Design Inc., 1993.

[8] R. Bretl et al. The GemStone Data Management System. In W. Kim and F.H. Lockovsky, editors, *Object–Oriented Concepts, Databases and Applications*, chapter 12. ACM Press, 1989.

[9] 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.

[10] F. Ferrandina, G. Ferran, J. Madec, T. Meyer, and R. Zicari. Database Evolution in the O_2Database System. In U. Dayal, P.M.D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, September 1995. Morgan Kaufman.

[11] 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.

[12] F.Ferrandina, T.Meyer, and R.Zicari. Correctness of Lazy Database Updates for an Object Database Systems. In *Proceedings of the 6th International Workshop on Persistent Object Systems.* Springer Verlag, September 1994.

[13] Inc. Itasca Systems. ITASCA Distributed Object Database Management System. Technical Summary for Release 2.2. Distributed at Object World 1994, Frankfurt am Main, Germany, 1993.

[14] B. Meyer. *Object–Oriented Software Construction.* Prentice Hall, Hertfordshire, England, 1988.

[15] O2 Technology. *The O_2C Reference Manual. Version 4.5., Release June 1994.* O2 Technology, Versailles, France, June 1994.

[16] Inc. Object Design. *ObjectStore – User Guide: DML. Objecstore Release 3.0 for UNIX Systems.* Object Design Inc., Dec 1993.

[17] D.J. Penney and J. Stein. Class modification in the Gemstone OODBMS. In *OOPSLA 87*, pages 111–177, 1987.

[18] 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.

[19] O2 Technology. *O2C Reference Manual, Version 4.5, Release November 1994.* O2 Technology, Versailles, France, November 1994.

[20] R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O2*, pages 146–182. Morgan Kaufmann, San Mateo, CA, 1992.