Schema and Database Evolution in the O_2 Object Database System

Fabrizio Ferrandina
Thorsten Meyer
Roberto Zicari

J. W. Goethe-Universität
FB 20, Robert Mayer Straße 11–15
60054 Frankfurt am Main, Germany

Guy Ferran
Joëlle Madec

O_2 Technology
7, Rue du Parc de Clagny
78000 Versailles, France

Abstract

When the schema of an object-oriented database system is modified, the database needs to be changed in such a way that the schema and the database remain consistent with each other. This paper describes the algorithm implemented in the new forthcoming release of the O_2 object database for automatically bringing the database to a consistent state after a schema update has been performed. The algorithm, which uses a deferred strategy to update the database, is a revised and extended version of the screening algorithm first sketched in [7].

1 Introduction

When the schema of an object-oriented database system is modified, the database needs to be changed in such a way that the schema and the database remain consistent with each other. The decision how to change the database is mainly an application-specific issue. This paper focuses on two aspects of the new release of the O_2 system. The first one is how the designer specifies the way objects in the database have to be updated as a consequence of a (or a series of) schema modification(s). The second aspect covers the specifications of the data structures and the algorithm used by O_2 for automatically bringing the database to a consistent state after a schema update has been performed.

The algorithm uses a deferred strategy to update the database, and it is a revised and extended version of the screening algorithm first sketched in [7].

Choosing to implement database updates as deferred updates poses some interesting implementation problems when ensuring the correctness of the implementation as it will be explained in the rest of the paper.

1.1 The O_2 Database System and How to Change its Schema

The main structure of an O_2 schema consists of a set of classes related by inheritance and/or composition links. An O_2 schema contains the definition of types, functions, and applications, while an O_2 *base* groups together *objects* and *values* which are created to conform to a schema.

An *object* has an identity, a value, and a behavior defined with its methods. Objects are class instances and values are type instances. A given object can be shared (referenced) by several entities (an entity is either an object or a value). By default, objects and values created during program execution are not persistent. To become persistent, an entity must be directly or indirectly attached to a *name*, i.e., a persistent root belonging to the schema.

A *class* definition consists of a type definition and a set of methods. A *type* is defined recursively from atomic types (integer, boolean, char, string, real, ...), classes, and constructors (tuple, set, list, ...). Methods are coded using the O_2C or the C++ language which allows to express manipulations on persistent as well as non-persistent entities.

In O_2, encapsulation is provided at different levels. First, properties (attributes and methods) are private to their class by default. Programs are encapsulated into

Proceedings of the 21th VLDB Conference
Zürich, Switzerland, 1995

applications. Finally, encapsulation is provided at the schema level as elements of a schema cannot be used by another schema. In order to increase reusability, O_2 provides an import/export mechanism.

Schema modifications can be performed in O_2 either in an incremental way using specific primitives (e.g. by adding or deleting attributes in a class) or by redefining the structure of single classes as a whole [12]. The O_2 schema manipulation primitives available in the O_2 product are briefly presented below [12, 19]:

1. creation of a new class
2. modification of an existing class
3. deletion of an existing class
4. renaming of an existing class
5. creation of an inheritance link between two classes
6. deletion of an inheritance link between two classes
7. creation of a new attribute
8. modification of an existing attribute
9. deletion of an existing attribute
10. renaming of an existing attribute

No matter how a class is modified, O_2 performs only those schema modifications that keeps the schema consistent [5].

The rest of the paper is structured as follows: in Section 2 we present from a user perspective how to define and use *conversion functions* as a means to instruct the system on how to change objects in the database as a consequence of a schema change. In addition to updating objects as a consequence of a schema change, O_2 allows to modify the structure of individual objects by moving them from one class to another independently from any schema change. This is described in Section 2.3 as *object migration*. Implementation details on conversion functions are given in Section 3 and 4. In particular, in Section 3.1 we illustrate the problems of implementing conversion functions, and in Section 3.2 we present the data structures used in the implementation. The detailed algorithm for implementing conversion functions as deferred database updates is presented in Section 4. In Section 5 we review relevant related work and compare our approach with existing ones. Finally, in Section 6, we present the conclusions.

2 Database Updates in O_2

In this section we describe the functionalities that have been added in the new release of O_2 for automatically updating the database after a schema has been modified.

The semantics of updating the database after a schema change depends on the application(s) which use(s) the schema. The basic mechanism to update the database is very simple: the designer has the possibility to program so called *conversion functions* which are associated to modified classes in the schema and define how objects have to be restructured. If no conversion functions are provided by the designer, the system provides *default conversion functions* where no programming is required. Instead, default transformation rules are applied to objects of modified classes.

Similar concepts to user-defined database conversion functions can be found in GemStone[3], ObjectStore[13], OTGen[10], whereby Versant [18] and Itasca [9] offer features that are similar to default conversion functions only. The definition and modality of use of conversion functions is explained in Sections 2.1 and 2.2.

The main design issue when implementing database (user-defined or default) conversion functions, is *when* such functions have to be executed, that is when the database has to be brought up to a consistent state wrt. the new schema.

We had two possible strategies to choose [6, 7]: an *immediate* strategy, where objects in the database are updated in any case as soon as the schema modification is performed, and a *deferred* strategy, where objects are updated only when they are actually used. The two above strategies have advantages and disadvantages [6, 7]; in O_2 we have supported *both* strategies and gave the designer the possibility to select the one which is most appropriate for his/her application domains. The implementation details are presented in Sections 3 and 4.

2.1 Default Database Transformations

In this section we describe what we called *default* database conversion functions. If no user-defined conversion functions are specified (see Section 2.2), the system transforms the objects in the database using default transformation rules. When a class in the schema is modified, the system compares each attribute of the class before and after the modification of the class and transforms the values of the object attributes according to the default rules as follows[1]:

- An attribute defined in a class before its modification and non present in the class after the modification (i.e. a deleted attribute) is ignored.
- An attribute which is not present in a class before its modification and present after its modification (i.e. a new attribute) is initialized with default initial values (i.e. 0 for an integer attribute, nil for an attribute referring to a class, etc.).
- An attribute present in both the class before the change and after the change is transformed according to the rules in Table 1.

[1] Note that, in O_2, after a class modification has been performed, two attributes are considered the *same attribute* if they have the *same name*.

attribute initial type attribute final type	int (i)	char (c)	bytes (b)	real (r)	string (s)	boolean	list	unique set	set	tuple	class_name
int	■	atoi(c)	atoi(b)	C cast	atoi(s)	0 / 1					
char	C cast	■	b[0]		s[0]	'0' / '1'					
bytes	sprintf(i)	"c"	■	sprintf(r)	■	"false" / "true"					
real	C cast		atof(b)	■	atof(s)	0.0 / 1.0					
string	sprintf(i)	"c"	■	sprintf(r)	■	"false" / "true"					
boolean	if i != 0 then true else false			if i != 0.0 then true else false		■					
list							list_to_list()	unique_set_to_list()	set_to_list()		
unique set							list_to_uniqiue_set()	unique_set_to_unique_set()	set_to_unique_set()		
set							list_to_set()	unique_set_to_set()	set_to_set()		
tuple										▤	
class_name											▨

□	initialize the modified attribute with the system initial value of the attribute final type
■	the value remains unchanged
▒	the transformation is done using a C cast or a C library function
□	the transformation depends on the domains of the constructors and is obtained recursively
▤	the fields of the tuple are transformed individually in the same way as attributes of a class
▨	if the final type is a superclass of the initial type the value remains unaltered; otherwise nil

Table 1: Attribute default conversion.

In the table, *attribute initial type* refers to the type of an attribute before the class modification, whereas *attribute final type* refers to the type of the same attribute after the modification of the class. If, for instance, an attribute of a class is declared of type real (the *attribute initial type*), and after a schema modification its type is transformed to integer (the *attribute final type*), a C cast is applied which truncates the real value. For those attributes where an empty entry appears in Table 1, the system initial value for the final type is used.

Let us consider a simple example, the schema *Car_showroom* which we suppose has been defined at time t_0:

```
schema creation at time t0:

create schema Car_showroom;
class Vendor type tuple ( name: string,
                          address: tuple ( city : string,
                                           street : string,
                                           number : real),
                          sold_cars: list( Car ))
end;

class Car type tuple ( name: string,
                       price :  real,
                       horse_power : integer )
end;
```

Assume we only have one object in the database for class *Vendor*, with values: *name* = "Volkswagen"; *address* = tuple(city: "Frankfurt", street: "Goethe", number: 5.0); *sold_cars* = list([1]: Golf_id, [2]: Passat_id, [3]: Corrado_id); where Golf_id, Passat_id, Corrado_id are references to *Car* objects.

Suppose at time t_1 the class *Vendor* in the schema is modified as follows:

```
schema modification at time t1:
modify class Vendor type tuple (name: string,
                                address: tuple ( street : string,
                                                 number : integer),
                                sold_cars: set( Car ))
end;
```

In the modified class *Vendor*, the type of the attribute *address* is now a tuple where the tuple field *city* has been deleted, and the tuple field *number* has become an integer instead of a real. Moreover, the attribute *sold_cars* is now a set instead of a list.

Since no user-defined conversion function is associated to the modified class *Vendor*, a default conversion function is applied. The object of class *Vendor* in the database is then automatically converted as follows: the attribute *name* keeps the value "Volkswagen", the tuple field *number* of attribute *address* is transformed from 5.0 to 5, and the attribute value of *sold_cars* becomes the set(Golf_id, Passat_id, Corrado_id), i.e. without order among values.

2.2 User-Defined Conversion Functions in O_2

The schema designer can override the default database transformations by explicitly associating user-defined conversion functions to the class just after its change in the schema.

In this case, the update to a class in the schema is performed in *two phases*. The first phase is the update to the class, i.e. using schema updates primitives. This phase is called *class modification phase*. The second phase is when user-defined conversion function(s) are associated, i.e. *defined and compiled*, to the modified class(es). This second phase is called *conversion functions definition phase.*

We show the definition of user-defined conversion functions using the previous example. Assume at time t_2 the schema designer decides to delete the attribute *horse_power* in class *Car*, but to retain the information by adding the attribute *kW* in class *Car* instead. This can be done as follows:

```
schema modification at time t2:
begin modification in class Car;
delete attribute horse_power;
create attribute kW : integer;
conversion functions;
conversion function mod_kW (old : tuple(name:string, price:real,
                                 horse_power:real)) in class Car
{
        self->kW = round( old.horse_power / 1.36 );
};
end modification;
```

Two schema update primitives for class *Car* are used after the command `begin modification in class Car`. The command `conversion function` associates the user-defined conversion function *mod_kW* to class *Car* after the change. In the body of the conversion function, "->" returns the attribute value of an object. The input parameter `old` of the conversion function refers to a tuple value conforming to the type of class *Car* before the modification has been performed. The variable `self` refers to an object of class *Car* after its modification. In the conversion function the transformation is not defined for all the attributes of class *Car* but only for the attribute *kW*. This is because a default transformation is executed in any case on objects before a user–defined conversion function is executed. This simplifies the writing of user–defined conversion functions. In the example, there is no need to write trivial transformations such as:

```
self->name = old.name,
self->price = old.price.
```

These transformations are performed by the default conversions.

The command `conversion functions` is optional. If not present, the system transforms the database using default transformations instead. The command `end modification` specifies the end of the class(es) transformation. Conversion functions are *logically* executed at the end of a modification block. The *real* execution time of the conversion functions depends on the implementation strategy chosen as it will be described in Sections 3 and 4.

Suppose now the attribute *sales* is added to the class *Vendor* at time t_3 (see schema modification at time t_3 shown in the next column).

At time t_3 class *Vendor* has been modified as a whole with the primitive `modify class` instead of using the primitive `create attribute sales in class Vendor`. The user-defined conversion function associated to class *Vendor* stores in *sales* the sales turnover for the vendor.

```
schema modification at time t3:
begin modification in class Vendor;
modify class Vendor type tuple ( name: string,
                                 address: tuple ( street : string,
                                                  number : integer),
                                 sold_cars: set( Car ),
                                 sales : real )
end;
conversion functions;
conversion function compute_sales (old : tuple(
                         name:string,
                         address: tuple ( street : string, number : interger),
                         sold_cars: set( Car ))) in class Vendor
{
        o2 Car c;
        for (c  in old.sold_cars) {
                        self->sales  += c->price;  }
};
end modification;
```

We should note in the example the difference between the conversion function *mod_kW* associated to *Car* at time t_2 and the conversion function *compute_sales* associated to *Vendor* at time t_3 . For the first one the value of the "updated" object is computed using only values locally defined to the object. The second conversion function instead uses the value of objects belonging to another class in the schema.

In [7] we have classified the above conversion functions as follows:

- *Simple conversion functions*, where the object transformation is performed using only the local information of the object being accessed (the conversion function *mod_kW* defined at time t_2).
- *Complex conversion functions*, where the object transformation is performed using objects of the database other than the current object being accessed (the conversion function *compute_sales* defined at time t_3).

This is an important distinction when implementing conversion functions as we will see in Sections 3 and 4.

Suppose we make a final schema modification at time t_4 by deleting the attribute *price* in class Car:

```
schema modification at time t4:
delete attribute price in class Car;
```

At time t_4 we did not associate any user-defined conversion function to class *Car*. The default conversion is then used for the transformation of the objects.

In Figure 1 we show a graphical representation of the schema modifications performed on the two classes. Classes connected by a solid arrow mean a modification has been performed on them, the label on the arrow indicate the presence of default or user-defined conversion functions.

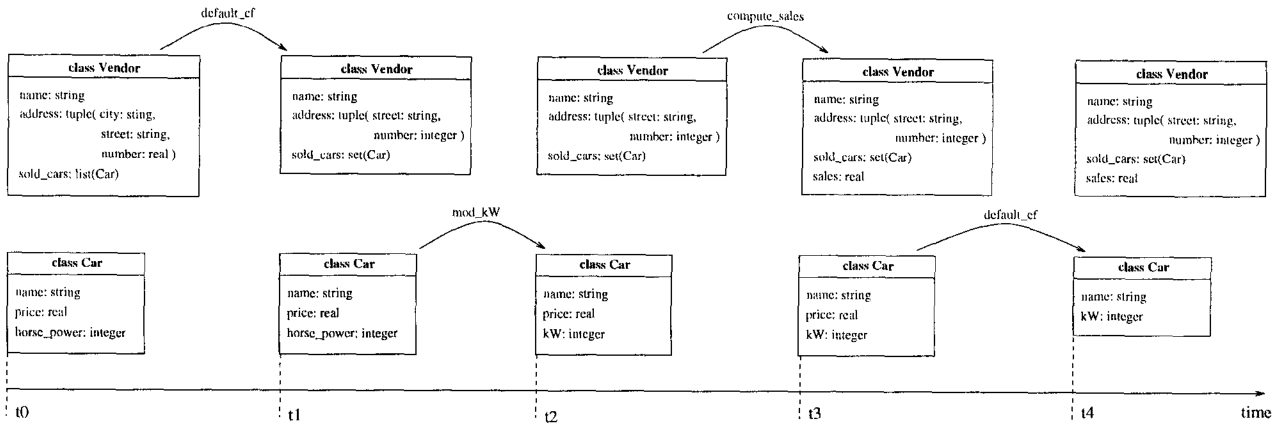

Figure 1: Schema evolution until time t_4.

The designer in O_2 has the possibility to specify the execution time for conversion functions. In particular, O_2 offers a command to execute conversion functions immediately, as follows:

```
transform database;
```

After the database transformation is completed, all objects in the database conform to the last schema definition. The default implementation modality for the execution of conversion functions is the deferred approach as described in Section 4.

So far, we have seen how objects of a class are updated as a consequence of a class modification using conversion functions. It is also possible in O_2 to update objects by migrating them to other classes in the schema. This is addressed in the next section.

2.3 User-Defined Object Migration Functions

Object migration refers to the possibility for an individual object to change its class during its lifetime. O_2 offers two ways to migrate objects, i) either a single object can change its class, or ii) an entire class extension (or a part of it) can be migrated to another class.

We start looking at the first possibility and then we consider class extensions.

We have defined a system method `migrate()` associated to the root class *Object* which, when invoked for a particular object, allows the object to migrate from its class to any of its subclasses (if any). In the method `migrate()`, the name of the target subclass must be given as an input parameter. We considered migration of objects to subclasses *only*, to avoid the possibility of run-time type errors if objects were allowed to migrate freely to any class in the schema.

Notwithstanding this limitation, this feature is particularly useful especially when: i) a new class is added to the schema and the existing objects of the class' superclasses need to be moved "down" to the new class, ii) a class is deleted and objects of that class must be retained by migrating them to subclasses.

The other possibility is to migrate an entire class extension (or a part of it) to other subclasses by means of a so called *migration function.*

The use of migration functions is explained using our example. Suppose at time t_5 the designer creates a new class *Sport_car* in the *Car_showroom* schema. After the creation of the class, he/she wants to migrate powerful cars, i.e. those cars with power kW >= 100, from class *Car* to class *Sport_car*. This can be achieved as follows:

```
schema modification at time t5:

class Sport_car  inherit Car
                 type tuple  ( speed : integer )
end;

migration function migrate_cars in class Car
{
        if ( self->kW >= 100 )
                self->migrate( "Sport_car" );
};
```

The migration function *migrate_cars* is associated to class *Car*. In the body of the migration function the system method **migrate** is called to migrate each object satisfying the selection condition to the subclasses *Sport_car.*

The example shows the importance of having object migration when new classes are added to the schema. Let us consider the case of the deletion of a class. Suppose the designer wants to delete class *Car*, but retain some of the objects in the database by moving them to another class. By creating class *Sport_car* and migrating *Car* objects to it, if the designer deleted class *Car* from the schema, he/she would lose only part of the objects, namely the ones whose attribute *kW* is lower than 100. Without migration there had been no chance to retain any object of class *Car.*

As in the case of conversion functions, migration functions can be executed either with an immediate or a deferred modality. By default, O_2 uses a deferred approach for the migration of objects. It is however possible to migrate objects immediately by explicitly calling the **transform database** schema command. More on this in Sections 3 and 4.

3 The Implementation of Database Updates in O_2

O_2 supports both the immediate and the deferred database transformation. However, the basic principle we followed whem we implemented the mechanism for database updates is the following: whatever transformation strategy is chosen for implementing a database transformation, there should be no difference for the schema designer as far as the result of the execution of the conversion functions is concerned [7]. From the above principle we derived the notion of correctness of a deferred database transformation, as first introduced in [7] and formally defined in [6]. A correct implementation of a deferred database transformation satisfies the following criteria:

The result of a database transformation implemented with a deferred modality is the same as if the transformation were implemented with an immediate modality.

The formal proof of correctness for the algorithm we will present in Section 4 is given in [6].

3.1 Deferred vs. Immediate Updates

In this section we present the data structures used in O_2 for supporting immediate and deferred database transformations. Since in O_2 the immediate database transformation is implemented using the deferred one, in the rest of the section we will mainly concentrate on the implementation details for deferred database transformations.

In Section 2.2 we have made the distinction between simple and complex conversion functions. The reasons for that is that implementing *complex* conversion functions for a deferred database transformation requires special care [7]. To explain why, consider in our usual example two objects v of class *Vendor* and c of class *Car* conforming to the respective class definitions at time t_2 (see Figure 1). Object v refers to c through the attribute *sold_cars*. If object c were accessed by an application at time t_a, with $t_4 < t_a$, the system would transform the object to conform to its last class definition deleting the attribute *price* from it. If, at time t_b, with $t_a < t_b$, object v is accessed, v will be restructured as well and its new value will be computed by applying the conversion function *compute_sales*.

The problem is that *compute_sales* accesses object c via the attribute *price*. But c now does not have anymore all the information required for the transformation of v because it has lost the attribute *price* when it was transformed at time t_a. In this special case, the execution of *compute_sales* would result in a run-time type error. In general, using default values as described in Section 2.1 for the restructured object v does not solve the problem, as it could result in an incorrect database transformation.

Let us consider again the database at time t_2 and assume the immediate database transformation had been used to transform objects v and c. If at time t_3 the system had transformed the object v immediately by executing the conversion function *compute_sales*, no run-time type error would have occurred because at time t_3 the object c accessed by the conversion function would have had the attribute *price*. The deletion of *price* at time t_4 would therefore not affect the execution of previously defined conversion functions. This is the correct transformation of the database.

In Section 3.2 we will present in detail the data structures and in Section 4 the algorithm used in O_2 for implementing simple and complex conversion functions using deferred database updates which guarantees a correct database transformation. The basic idea is to physically retain the deleted or the modified information in the object in a so called *screened part*. This implementation strategy is commonly known with the name of *screening* [1]. Applications running against the database do not have access to the screened information, but conversion functions, instead, have access to the screened information in order to perform a correct database transformation.

When some information is deleted and/or modified in the schema, it is only screened out, but not physically deleted in the database. When, for instance, a deletion of an attribute (or a change in the type which would correspond to a deletion and an addition of the same attribute) is performed, the update is not physically executed on the object structure but simply a different representation of the object is presented to applications. Using screening, O_2 manages the different representations of an object, one representation visible to applications and one representation visible to conversion functions only.

3.2 Data Structures

The physical format of an object, i.e. as it is internally stored in the database, contains two parts: *the object header* and *the object value*, the value itself being composed of an *existing value* and a *screened value* (see Figure 2).

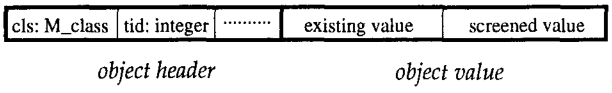

Figure 2: Structure of an O_2 object.

The object value part is used for storing values that reside within the object, such as attribute values. The object header contains, among other info, the *identifier of the object's class descriptor* (`cls`) and the *type entry identifier* (`tid`) according to which format the object itself is stored. Each of these two can be viewed as somewhat special fields in the physical format of the object.

The main principle in the implementation of deferred updates is to keep track of the evolution of the schema. The O_2 *schema manager* manages a persistent symbol table containing schema components such as class definitions, type definitions, etc..

A simple integer variable called *schema state* is associated to each schema. The *schema state* is incremented every time a class in the schema undergoes a change.

All components of a schema are *internally* maintained as meta-objects. Each class in the schema is internally represented by a *class descriptor* which can be considered as an instance of the O_2 class *Meta_class* illustrated in Figure 3.

```
class Meta_class  type tuple (
      :
      sch        : integer,                /* schema-id where the class is defined */
      name       : string,                 /* name of the class */
      visib      : char,                   /* access mode = public, read, private */
      type       : Meta_type,              /* type of the class */
      properties : list (Meta_property),   /* attributes and methods */
      parents    : list (Meta_class),      /* direct superclasses */
      children   : list (Meta_class),      /* all subclasses */
      ancestors  : list (Meta_class),      /* all superclasses */
      ispartof   : list (Meta_class),      /* classes with a component of this class */
      :          :
      cur_tid    : integer,                        /* current tid of this class */
      history    : list (Meta_history_entry), /* type history of this class */
      :  )
end;
```

Figure 3: The Meta_class definition for describing classes in the schema.

The class Meta_class contains all the information related to a class, i.e. its name, its type, its visibility (private vs. public), the list of its parents classes, etc.. In particular, to implement a deferred database transformation, each class descriptor contains a field `cur_tid` which is used for testing whether an object in the database conforms to the last class definition in the schema or not. Another important information in the class descriptor is stored in the field `history`, the list of *history entry descriptors* containing the information of the class as it was defined *"in the past"*.

A history entry descriptor can be considered as an instance of the class *Meta_history_entry* (see Figure 4) and contains the following fields:

- the type entry identifier `tid`, a simple integer number, which helps in identifying to which entry an object of the class belongs to. When a class undergoes a change, the *schema state* is assigned to the `tid`,
- the type `type` which corresponds to the type of the class visible by applications,
- the type `ex_type` which corresponds to the extended type of the class including the screened information,
- the entry `struct` which contains a list of *property entry descriptors*,
- a field `cf` which contains a reference to a *conversion function descriptor* that is used to convert objects to conform to a subsequent entry in the history,
- a field `mf` which contains a reference to a *migration function descriptor* that is used to migrate objects to conform to the appropriate entry in the history of a subclass[2].

A property entry descriptor belonging to the `struct` list of a history entry descriptor can be considered as an instance of the class *Meta_property_entry* (see Figure 4). It contains the following information:

- the `pid` of the attribute; the reason for using such an identification is that the external name of a property can be changed without affecting the identity of a property,
- the `sch_state`, i.e the state of the schema when the attribute has been created. The information (pid, sch_state) identifies an attribute in a non ambiguous way.
- the `offset` of the attribute, i.e. the physical position of the attribute in the object itself,
- the `type` of the attribute,
- the `status` of an attribute indicating whether the given attribute can be accessed by both application and conversion functions (in this case the value is set to *existing*), or by conversion functions only (in this case the value is set to *screened*).

The last two components of a history entry descriptor, `cf` and `mf`, are the descriptors of a conversion and a migration function which can be considered as instances of the classes *Meta_conversion* and *Meta_migration* (see Figure 4). In a conversion function descriptor, the field `next_state` indicates to which entry in the class history the conversion function stored as a binary file in the field `function` is supposed to transform objects. The same applies for a migration function descriptor. The `sch_state` field indicates the state of the schema when the migration function has been associated to the class. The `sch_state` information is used by the system to determine to which history entry of a subclass an object has to be migrated[2].

Recall the example we presented in Section 2. Figure 5 illustrates the class descriptor of *Car* after the migration function *migrate_cars* has been defined at time t_5.

The field `cur_tid` of the class is equal to 5 and corresponds to the *schema state* just after the migration function *migrate_cars* has been associated to the class. The field `history` points to a list of four history entry descriptors, whereby the one with `tid` = 0 identifies the original class information when it has been created at time t_0. The following history entry descriptors identify the information of the class after each class modification or after

[2] In Section 4 we describe how O_2 infers the history entry in the target class when executing a migration function.

```
class Meta_history_entry type tuple (        class Meta_property_entry type tuple (   class Meta_conversion type tuple (    class Meta_migration type tuple (
      tid       :  integer,                        pid        :  integer,                     next_state :  integer,                 sch_state  :  integer,
      type      :  Meta_type,                      sch_state  :  integer,                     function   :  Meta_binary )            function   :  Meta_binary )
      ex_type   :  Meta_type,                      offset     :  integer,               end;                                  end;
      struct    :  list(Meta_property_entry),      type       :  Meta_type,
      cf        :  Meta_conversion,                status     :  { existing, screened } )
      mf        :  Meta_migration )          end;
end;
```

Figure 4: The meta-definitions for the different descriptors in the class history.

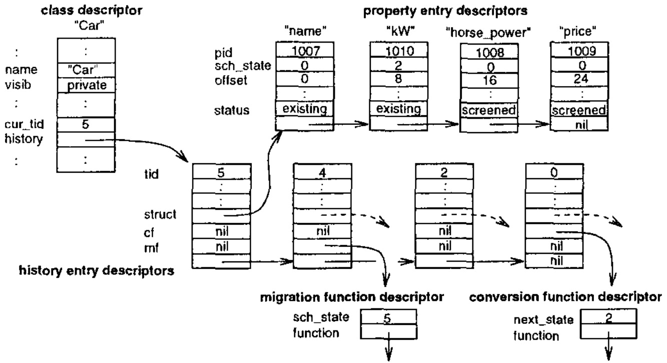

Figure 5: Descriptor of class *Car* along with its history.

the association of a migration function to the class. For reasons of readability, we show only the **struct** information related to the first history entry descriptor whose **tid** = 5. The field **struct** points to the list of property entry descriptors belonging to the class. The first two property entry descriptors refer to attributes in the class which are visible by application. This can be recognized by the value "existing" in the **status** field. The last two property entry descriptors refer to screened attributes visible by conversion functions only. It is important to note that screened attributes are physically stored *always* after visible attributes, i.e. their offset in the chunk of memory representing an object is always greater than the one of a visible attribute.

The conversion function descriptor for *mod_kW* and the migration function descriptor for *migrate_cars* are associated to the appropriate history entry descriptors.

4 The Deferred Database Update Algorithm

We first introduce some definitions. The most recent entry in a class history is called *current history entry.* An entry in a class history is called *input relevant* if this entry holds a conversion or a migration function. The current history entry is defined as input relevant as well. From now on, the class descriptor of a class X is referred to as X*_desc.*

When a new class X is created, the schema manager of O_2 instantiates a class descriptor with the appropriate information, i.e. the name of the class, the list of parent and ancestor classes in the hierarchy, etc.. In particular, the field **cur_tid** is initialized with the *schema state* associated to the schema and a first entry is created in the history of the class.

After a modification is performed on a class X, the *schema state* is incremented and a new entry in the class history is created for the modified class X and for all of its subclasses which have effectively undergone a modification. The newly created entry becomes the current history entry and its **tid** is initialized with the *schema state.* For those subclasses where no modification have taken place (e.g. because an attribute has been added to X which is already present in its subclasses), no new entry in the class history is created. If a conversion function is associated to the class modification, the schema manager instantiates and initializes a conversion function descriptor and assigns it to the **cf** field of the history entry descriptor which chronologically precedes the current history entry. The **function** field of the conversion function descriptor contains a pointer to the binary code of the conversion function. The field **next_state** contains the **tid** of the current history entry.

The same happens for a migration function. When a migration function is associated to a class X, the *schema state* is incremented and a new entry in the history of X is created. The newly created entry becomes the current history entry and its **tid** is initialized with the *schema state.* The schema manager instantiates and initializes a migration function descriptor which is then assigned to the **mf** field of the history entry descriptor which chronologically precedes the current history entry. The **function** field of the migration function descriptor contains a pointer to the binary code of the migration function. The field **sch_state** contains the **tid** of the current history entry.

4.1 Basic Deferred Update Algorithm

The algorithm used by O_2 when an object o of class X is accessed by an application is the **Deferred Update Algorithm** shown in Figure 6.

The algorithm first checks whether an object conforms to its last class definition in the schema. If yes, the object can be used by the application which accessed it without being first transformed. If not, O_2 identifies the appropriate history entry descriptor in the history of class X to which object o conforms to. Three alternatives are then possible: i) the history entry descriptor contains a migration function which implies a *possible* migration of o to a subclass of X, ii) the history entry descriptor contains a conversion function which implies that o *must* be restruc-

Deferred Update Algorithm

```
while (o->tid <> o->cls->tid ) do                    /* o is not in current format */
   for ( X_his_desc in o->cls->history where X_his_desc->tid == o->tid)
      { break };                    /* find the history entry descriptor to which o conforms */
   if (X_his_desc->mf <> nil) then    /* a migration function has to be applied */
      apply the migration function X_his_desc->mf->function;
      if (object o has not been migrated) then
            modify the tid of the object to correspond to the tid
            belonging to the chronologically following entry;
      endif;
   else           /* a default or user-defined conversion function has to be applied */
      copy the value of o in a variable old;            /* old is used by the cf's */
      if (X_his_desc->cf <> nil) then  /* an user-defined cf has to be applied */
            restructure o to conform to the entry in the history whose
            tid corresponds to X_his_desc->cf.next_state;
            apply the default conversion function;
            apply the conversion function X_his_desc->cf->function;
            o->tid = X_his_desc->cf.next_state;
      else                                      /* a default cf has to be applied */
            restructure o to conform to the next input relevant entry in
            the class history;  /* entry with a migr. or user-def. conv. function */
            apply the default conversion function;
            update  the tid to correspond to the one found in the
            next input relevant entry;
      endif;
   endif;
endwhile;
```

Figure 6: The deferred update algorithm.

tured to conform to a more recent entry in the history of class X, iii) the history entry contains neither a conversion nor a migration function; object o *must* be restructured and reinitialized using a default conversion function to conform to the next input relevant entry in the history. Note that, due to how class descriptors are maintained by O_2, no entry will ever contain both a conversion and a migration function.

4.2 Implementing Complex Conversion Functions

The deferred update algorithm presented before works fine if only *simple* conversion functions have been defined when evolving the schema. In case of *complex* conversion functions, instead, the transformation of objects accessed by complex conversion functions must be stopped before reaching the state corresponding to the current history entry to avoid database inconsistencies or run-time type errors [7].

Suppose that a complex conversion function *cf* associated to a history entry with tid $= i$ of a class X transforms objects of that class to conform to a history entry with tid $= j$, where $j > i$. If other objects are accessed by *cf*, their transformation should not be propagated up to the current history entry, but it must be stopped at a history entry which is the one *visible* by the conversion function *cf* at the time it was defined. The concept of visibility is modeled by the tid's attached to each entry in the history of a class.

The nth history entry of a class Y in the schema is *visible* by *cf* if:

Y_desc ->history[n]->tid <= j

and the chronologically subsequent entry (if any)

Y_desc ->history[n-1]->tid > j

where history[n] indicates the nth history entry descriptor in the history list of a class and history[n-1] indicates the entry that chronologically follows history[n][3].

In order to stop the transformation of objects to the visible history entry O_2 maintains a *stack associated to each application*. Before the execution of an application or of a conversion function, the system pushes in the stack the appropriate entry number signaling up to which entry in the history an object has to be transformed (the actual *schema state* for the application, or a smaller number for a conversion function). This number is removed from the stack after the execution of a conversion function or the execution of an application.

The correctness of the deferred update algorithm has formally been demonstrated in [6].

Reconsider the example in Section 2.2 where the complex conversion function *compute_sales* accesses objects of class *Car* to perform the computation of the vendor's turnover. Since the conversion function *compute_sales* is supposed to transform objects of class *Vendor* to conform to the history entry with tid $= 3$[4], the schema manager of O_2 pushes the value 3 on the stack. When an object c of class *Car* is accessed by the conversion function, c is transformed to conform to the history entry visible by *compute_sales*, i.e. the one with tid $= 2$.

4.3 Implementing Object Migration

If an object o conforming to the history entry descriptor of class X with tid $= i$ has to migrate to a target class Y due to the presence of a migration function descriptor, the deferred update algorithm executes the migration function stored in the mf field of the history entry descriptor. When migrating an object, the schema manager of O_2 must decide to which history entry of the target class Y a migrated object has to conform to. This is not necessarily the current history entry of Y because between the definition of the migration function and its execution, class Y might have been changed. The schema manager of O_2 identifies the history entry of the target class Y as the one whose tid j is the greatest satisfying the condition $j <= s$, whereby s is the value stored in the field sch_state of the migration function descriptor, i.e. the

[3] It might happen that objects accessed by a cf have a tid $\geq j$. In this case no transformation is triggered on them because they are already containing the information needed by cf.

[4] After the modification of class *Vendor* at time t_3, the *schema state* is equal to 3.

state of the schema at the time the migration function has been defined.

As shown in Section 2.3, the real migration of an object is performed by the execution of the system's method *migrate* which is called within a migration function.

The method migrate, when executed on an object o which has to migrate from class X to class Y, is responsible for the following:

- copy the value of o in a variable *old*;
- find the appropriate target history entry where o has to be migrated;
- restructure o to conform to the target history entry of class Y;
- perform the default transformation on o using the information present in *old*;
- update the class identifier `cls` in the header of o to be the one of the target class;
- update the type identifier `tid` in the header of o to be the one of the target history entry of class Y;

4.4 Implementing Class Deletions

So far, we discussed how to transform objects in the database when a class in the schema has been modified. Another important issue is how O_2 implements a class deletion.

Basically, when using the deferred database transformation, there is no way to control when objects are accessed by applications, i.e. when conversion functions are effectively executed. In particular, the execution of a *complex* conversion function might require the information of objects whose class has been deleted in the schema. Further, since migration of objects is implemented using a deferred modality as well, objects of a deleted class can be migrated to subclasses of the deleted class.

To accomplish a deferred database transformation when classes are deleted in the schema, the deletion of a class is not physically performed, but classes are only screened out from being used by applications. Only conversion and migration functions are allowed to access the information of screened classes. If class *Car* were deleted from the schema *Car_showroom*[5], the schema manager of O_2 would only set the field `visib` of the class descriptor to "deleted". This would imply that conversion functions accessing objects of class *Car* can still read the information needed for the transformation.

4.5 Optimization Issues

There is no need to screen all deleted classes but only those ones whose objects might be accessed by complex conversion functions or by migration functions. Therefore, O_2 internally maintains a so called *dependency graph* associated to each schema which allows the schema manager to understand when deleted classes have to be screened. The dependency graph is defined as follows:

Definition: The **dependency graph** G is a tuple (V, E), extended by a labeling function $l : (V \times V) \rightarrow A$. V is a set of class–vertices, one for each class in the schema. E is a set of directed edges (v, w) $v, w \in V$. A is a set of attribute names and the special value "mf" which identifies a migration function. An edge (v, w) indicates that there exists at least one complex conversion function associated to class w which uses the value of objects of class v or that a migration function is associated to class v which migrates objects to class w. The function *l(v,w)* returns the names of the attributes of class v used by conversion functions associated to class w and/or "mf" if objects have to be migrated to class w.

Evolution of the schema implies changing the dependency graph associated to the schema. By looking at the dependency graph it is possible to identify when classes have to be screened due to a definition of a complex conversion function or a migration function.

The use of the graph is shown with our usual example, the *Car_showroom* schema. In Figure 7 the evolution of the dependency graph for the schema *Car_showroom* from time t_0 till time t_5 is illustrated. The conversion function defined at time t_1 uses only local defined attributes, therefore no edge appears in the graph. At time t_3, the edge is added to the graph because of the definition of the complex conversion functions *compute_sales*. At time t_5, a new edge is added to the dependency graph due to the definition of the migration function *migrate_cars*.

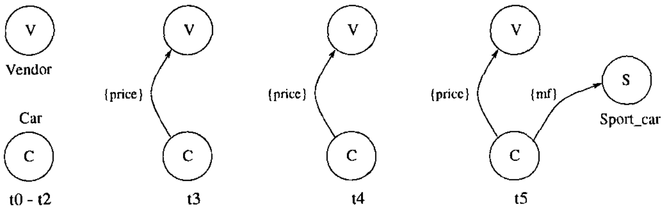

Figure 7: Evolution of the dependency graph of schema *Car_showroom*

The dependency graph has to be checked by the system in the following cases: i) a class is modified along with a complex conversion function, ii) a class is deleted from the schema, iii) a migration function is associated to a class. If, for instance, class *Vendor* is deleted from the schema, the schema manager of O_2 recognizes that there is no outgoing arrow for class *Vendor* in the dependency graph and therefore the class can be really removed along with its extension.

If no space optimization is taken into account when using screening, i.e. if the information is never deleted in the objects, the size of the database risks to grow contin-

[5] Note that in the current version of O_2 only leaf classes can be deleted. To delete class *Car* would therefore imply to first remove the link with its subclass *Sport_car*.

uously.

The schema manager of O_2 optimizes it by physically delete the information in those objects which will never be accessed by any complex conversion function. This can be easily obtained by checking the dependency graph. Objects of classes which do not have any outgoing arrow in the dependency graph should not contain screened attributes because no conversion function will ever use them. Objects of classes which have an outgoing arrow in the dependency graph contain only screened attributes whose name appears to be returned by the labeling function associated to the arrow. Moreover, every time the immediate database transformation is launched, O_2 transforms all the objects to conform to the last schema definition. After the transformation, the system deletes the edges in the dependency graph and therefore the screened part can be dropped from all the objects in the database. As a consequence of the immediate database transformation, all histories of the class descriptors are updated to contain only one history entry, namely the current history entry with the information of the class as visible by applications.

4.6 Implementing Immediate Database Updates

In O_2, the immediate database transformation is implemented using the algorithm defined for the deferred database transformation. When the designer specifies the schema command:

```
transform database;
```

the schema manager of O_2 launches an internal tool which is responsible to access all objects in the database which are not up to date. When accessed, objects are transformed according to the algorithm defined for the deferred database transformation.

The tool follows basically two strategies for accessing objects which are not up to date. If class extensions are maintained by the system[6], extensions of updated classes have to be iterated to access all objects of that class. If extensions are not maintained by the system, the tool accesses objects in the database starting from appropriate roots of persistence and following the composition links between objects.

As already mentioned, after an immediate database transformation, the dependency graph is updated. Further, the history of all classes is deleted and the deleted part of screened objects is dropped.

5 Related Work

Not all available ODBSs provide the feature of adapting the database after a schema modification has been performed [15, 16]. For those that do it, they differ from each other in the approach followed for updating objects. Some commercial systems support the possibility to define object versions to evolve the database from one version to another, examples are *Objectivity* [14] and *GemStone* [3]. *Objectivity* does not provide any tool to automatically update the database, besides providing object versions. The designer has to write a program which reads the value *old_val* of objects of the old version, computes the new value *new_val* and assigns it to the correspondent objects of the new version. The program can be written in order to transform the database both immediately and lazily. *GemStone*, instead, provides a flexible way for updating object instances. It provides default transformation of objects and the possibility to add conversion methods to a class. Conversion methods can update objects either in groups (for instance the whole extension of a class) or individually. The transformation of the database is performed in a deferred mode but manually, i.e. objects are transformed on demand only when applications call the transformation methods. The problems pointed out in this paper do not occur when versioning is used because objects are never *transformed*, but a new version is created instead. Therefore the information for the transformation of an object can always be found in its correspondent old version.

On the other hand, the majority of the existing commercially available systems do not use versioning for updating the database. Applications can run on top of the schema as defined after the last modification. Instances are converted either immediately or lazily. *ObjectStore* [13] makes use of the immediate database transformation. So called *transformation functions*, which override the default transformation, can be associated to each modified class. Objects are not *physically* restructured, but a new object (conforming the definition of the modified class) is created instead. The transformation function reads the value in the old object and assigns it (after having made some modification on it) to the new object. All references to the object have to be updated in order to point to the newly created object. This technique resembles the one used by those systems providing versions, the only difference being that, after the transformation, the old objects are discarded. Deferred transformation of objects is provided in systems like *Itasca* [9] and *Versant* [18]. They both do not provide the user with flexible conversion functions like the one presented in the paper. Instead, they have the possibility to override a default transformation assigning new constant values to modified or added attributes of a class.

Among research prototype systems, *Avance* [2], *CLOSQL* [11], and *Encore* [17] all use object versioning. *Orion* [1] uses a deferred approach where deletion of attributes is filtered. Information is not physically deleted, but it is no more usable by applications. No conversion functions are provided to the schema designer.

[6] Note that O_2 does not automatically maintain extensions associated to classes. It is the responsibility of the designer to inform the system if extensions are to be kept or not.

In summary, if we consider those database systems using a deferred database transformation, then no one is currently offering conversion functions like the one presented in this paper.

Updating the database using only default transformation of objects is clearly not flexible and powerful enough.

6 Conclusions and Future Work

In this paper we have discussed how the new release of the O_2 object database has been enhanced to offer an automatical database modification mechanism after a schema change. O_2 supports both the immediate database and the deferred database transformation, whereby the deferred transformation is used by default. We have described how O_2 transforms objects by means of default transformation rules and by means of user–defined conversion functions. We also described how to associate migration functions to classes in order to move objects "down" in the class hierarchy. Object migration is suitable both when new classes are added to the schema which are more appropriate classes for existing objects, and to retain objects in the database when classes are deleted from the schema. Finally, we presented the data structures used by O_2 for implementing the deferred database transformation and the algorithm used by the system to transform objects to conform to their last class definition to be properly accessed by applications.

We are currently evaluating the performance of the algorithm proposed in this paper and the ones defined in [7] using the OO7 benchmark [4]. We are defining an appropriate benchmark for analyzing the performance of immediate vs. deferred database updates [8].

Acknowledgments

The work presented in this paper is partially supported by the Goodstep project funded by the Commission of the European Communities under the ESPRIT III programme.

References

[1] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and Implementation of Schema Evolution in Object–Oriented Databases. In *Proc. of ACM SIGMOD Conf. on Management of Data*, San Francisco, CA, May 1987.

[2] A. Bjornerstedt and S. Britts. Avance: an Object Management System. In *Proc. of OOPSLA*, San Diego, CA, September 1988.

[3] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. H. Lockovsky, editors, *Object–Oriented Concepts, Databases and Applications*, chapter 12. ACM Press, 1989.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. ACM SIGMOD Conf.*, pages 12–21, Washington, DC, USA, May 1993.

[5] C. Delcourt and R. Zicari. The Design of an Integrity Constaint Checker (ICC) for an Object-Oriented Database System. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, number 512 in Lecture Notes in Computer Science, Geneve, Switzerland, July 1991. Springer.

[6] F. Ferrandina, T. Meyer, and R. Zicari. Correctness of Lazy Database Updates for an Object Database System. In *Proc. of the 6th Int'l Workshop on Persistent Object Systems*, Tarascon, France, September 5-9, 1994.

[7] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, pages 261–272, Santiago, Chile, September 12-15, 1994.

[8] F. Ferrandina, T. Meyer, and R. Zicari. Lazy Database Updates Algorithms: a Performance Analysis. Technical report, J.W. Goethe Universität, 1995. In preparation.

[9] Itasca Systems, Inc. *Itasca Systems Technical Report Number TM-92-001. OODBMS Feature Checklist. Rev 1.1*, December 1993.

[10] B. S. Lerner and A. N. Habermann. Beyond Schema Evolution to Database Reorganization. In *Proc. of the ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) and Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, pages 67–76, Ottawa, Canada, October 21-25, 1990.

[11] S. Monk and I. Sommerville. A Model for Versioning Classes in Object–Oriented Databases. In *Proc. of the 10th British National Conf. on Databases*, Aberdeen, Scotland, July 1992.

[12] O_2 Technology. *The O_2 User Manual, Version 4.5*, November 1994.

[13] Object Design Inc. *ObjectStore User Guide, Release 3.0, chapter 10*, December 1993.

[14] Objectivity Inc. *Objectivity, User Manual, Version 2.0*, March 1993.

[15] J. E. Richardson and M. J. Carey. Persistence in the E language: Issues and Implementation. *Software – Practice and Experience*, 19(12):1115–1150, December 1989.

[16] B. Schiefer. Supporting Integration & Evolution with Object-Oriented Views. *FZI-Report 15/93*, July 1993.

[17] A. H. Skarra and S. B. Zdonik. Type Evolution in an Object–Oriented Database. In Shriver and Wegner, editors, *Research Directions in Object–Oriented Programming*, pages 393–416. MIT Press, Cambridge, MA, 1987.

[18] Versant Object Technology, 4500 Bohannon Drive Menlo Park, CA 94025. *Versant User Manual*, 1992.

[19] 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 O_2*. Morgan Kaufmann, San Mateo, CA, 1992.