Simulation of Schema Change using Views

Philippe Brèche, Fabrizio Ferrandina, Martin Kuklok

J. W. Goethe-Universität, Fachbereich Informatik – FB 20
Robert Mayerstraße 11–15. D–60325 Frankfurt am Main

Abstract. In this paper we concentrate on the description of change management functionalities that have been added to the commercially available O_2 ODBMS to yield a platform suited for applications such as SDEs. A description of declarative high level schema update primitives is presented as well as mechanisms to propagate schema changes to the database. Particular emphasis is given to a mechanism that allows designers first to simulate schema and database modifications by means of object views, and second to materialize them to be propagated onto the real schema and base.
`Keywords`: schema and database evolution, object views, schema change simulation.

1 Introduction

The work presented in this paper is part of the broader GOODSTEP project 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: the O_2 ODBS [17, 13]. O_2 already includes many of the features required by SDEs. The project has identified enhancements to O_2 in order to make it a real software engineering database system.

The paper addresses how change management facilities in the O_2 database system have been improved to fulfil the requirements submitted by the software engineering partners of GOODSTEP [14]. In particular, the possibility to use the view mechanism as defined in [1] to *simulate changes* (both to the schema and the database) is presented.

Results that have been achieved in the project are the following: i) definition of complex declarative schema manipulation commands, ii) definition of conversion and migration functions for the *automatical* transformation of the database as a consequence of a schema change, and iii) simulation of schema changes by means of views[1].

With regards to the first point, new *complex* schema change primitives (from here on called *High Level Primitives – HLPs*) have been developed to help the designer in the modification of a schema. Therefore complex changes can be done more quickly and on a formal basis.

Conversion and migration functions enable the designer to decide which modification the database should undergo after a schema change.

Basic schema changes usually have lesser effect on a given schema than complex ones do have. Several complex primitives which are applied one after the other in

[1] This third task is currently still in progress.

one go might imply changes which then cannot be easily overviewed. Therefore it is necessary to support the designer in order to help him/her to capture the complexity of the complete change management process. Capacity-preserving or capacity-removing schema changes [16] can be simulated without any permanent effect to the underlying schema and bases. This simulation process, as a validation phase, should help to avoid to perform undesired changes.

We intend to glue these concepts together to achieve a change management environment suited to Software Engineering (SE) applications.

The rest of the paper is organized as follows: Section 2 presents the enhancement of schema evolution achieved through the declarativity of complex common operation performed during a schema design process. Section 3 illustrates how schema changes are propagated to the database by means of designer-defined conversion and migration functions. Finally, Section 4 shows how a view definition mechanism as it is defined in [2, 1] is used as a simulation tool for schema modification purposes.

2 High Level Primitives for Schema Evolution

2.1 HLPs: Functional Architecture and General Semantics

Currently, existing ODBSs perform schema changes by means of primitives closely related to the respective data model both on a syntax point of view and on the kind of updates they address. Taxonomies of schema update primitives can be found in [3, 11]. They usually differ because of the underlying data model. They may be considered as a *first level* of schema update primitives (from now on they are referred as *Low Level Primitives - LLPs*). It is recognized [4] that designers building database schemas suited for SE applications require a more *abstract level* to update a schema. Clear examples are primitives allowing generalization, specialization, or merge operations performed on classes in the schema. These *High Level Primitives* should be independent of the underlying data model and should be based on a restricted set of LLPs [17].

A schema modification through an *HLP* is a declarative transformation of a source schema into a target schema and acts as a shortcut for several *LLPS*. Such an operation is carried out preserving the O_2 schema invariants such as the Class Lattice Invariant, Distinct Name Invariant ...

The detailed semantics and implementation of the *HLP* module is beyond the scope of this paper. The reader is referred to [5] for more details. Hereafter the most important *HLPs* are presented along with a brief explanation of their semantics.

Abstraction-Generalization: This primitive generalizes two source class definitions into a lowest common superclass definition and refines it according to the source classes involved. This is performed by collecting the common properties of the source classes. Source classes are reconnected to the class of generalization.

Abstraction-Specialization: This primitive specializes one source class into several subclasses (which automatically become subclasses of the source class). The newly created classes are placed in the class hierarchy by specifying their other direct superclasses and subclasses.

Merge Classes: This primitive merges two source classes into one new class definition, deleting the source classes and consequently restructuring the class hierarchy.

Reclassification: This primitive reclassifies a class, i.e. repositioning in the class hierarchy. The new set of superclasses and subclasses of the class is to be declared. Changes due to this reclassification are propagated along the class hierarchy.

Remove: This primitive removes a class in the middle of the class hierarchy.

Abstraction-Aggregation: This primitive creates an aggregation class which is built up out of several source classes. The designer can specify the super- and subclasses of the resulting class to place it in the class hierarchy, define properties of aggregation links (e.g. cardinalities), and establish reverse-links (e.g. is_part_of). For instance, starting from the class definitions of Monitor, System_Box, Mouse, Keyboard, and so forth, one could think of introducing a class Microcomputer with the attributes Mouse, Keyboard.

Abstraction-Decomposition: This primitive acts in the opposite way as the Abstraction—Aggregation primitive does. It creates classes out of decomposition from a set of attributes addressed in a source class (the source class might be of type tuple only). The reverse link is_a_part_of is automatically created in these classes (the classes have types of kind tuple). Only those of the attributes which are specified together with a class generate a class of decomposition.

2.2 *HLPs*' Advantages: Example on Specialization

The following example details a simple application of the *HLP* **Abstraction-Specialization**. The concept *Figure* is defined in a first step of the design as a generalization of the concrete classes *Point*, *Line*, *Spline*, *Polygon*, and *Circle*. These definitions result in the *source* schema of figure 1. Suppose the new notion of "dimension" of a *Figure* has to be introduced in the schema. The insertion of classes *Figure_Dim_0*, *Figure_Dim_1*, and *Figure_Dim_2* forces the designer to reclassify the former direct subclasses of class *Figure*. Such a reorganization is required because the specialization of *Figure* is not performed on a leaf class in the inheritance graph. The classes *Point*, *Line*, *Spline*, *Polygon* and *Circle* become subclasses of the newly added classes *Figure_Dim_0*, *Figure_Dim_1*, and *Figure_Dim_2*. The resulting schema, *target* schema in figure 1, shows the class hierarchy after such a schema modification. The *HLP* the designer has to use in order to transform this source schema into the target schema is expressed as follows:

`specialize class` *Figure* `into class`

Dim_0 `sub(`*Point*`)`, *Dim_1* `sub(`*Line* , *Spline*`)`, *Dim_2* `sub(`*Polygon* , *Circle*`)`

The *specialize* abstract schema update is performed in a declarative way but may be replaced by an *equivalent* ordered sequence(s) of *LLPs*. The *HLP* does this mapping automatically, avoiding possible mistakes occurring in a complex process of identifying, composing and applying these *LLPs*.

An optimized set of *LLPS*, equivalent to the here above specialization declaration gives a hint of such an advantage. This sequence of *LLPs* is applied onto the same source schema to achieve the same target schema.

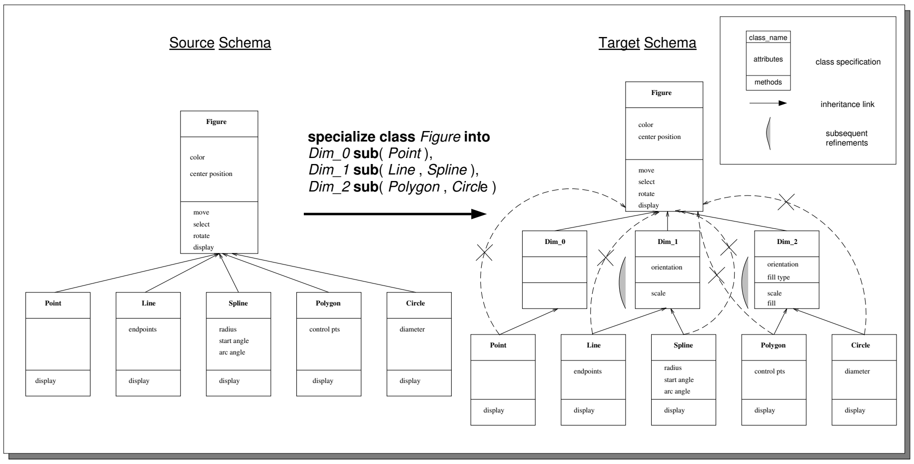

Fig. 1. Specialization of class "Figure" with the Dimensions.

```
create class Dim_0 inherit Figure end;
create inherit Point Dim_0;
delete inherit Point Figure;
create class Dim_1 inherit Figure end;
create inherit Line Dim_1;
delete inherit Line Figure;
create inherit Spline Dim_1;
delete inherit Spline Figure;
create class Dim_2 inherit Figure end;
create inherit Polygon Dim_2;
create inherit Polygon Figure;
create inherit Polygon Dim_2;
delete inherit Circle Figure;
```

Section 4 highlights the importance of being able to simulate such abstract updates to cope with the complexity of change propagation. The following section addresses how schema changes are propagated to the database.

3 Propagation of Schema Modifications to Instances

In GOODSTEP, as a consequence of a schema change, the database is modified with the help of designer-defined *conversion functions*[2] which take the old and new schema class definitions as input parameters and when executed transform the objects of the database to conform to the new schema.

There are mainly two strategies for implementing database conversion functions: immediate and lazy. In the first case, all objects of the database are updated immediately after the definition of the conversion functions. In the second case, objects are updated only when used (i.e. conversion functions are executed only when objects are effectively accessed by applications).

Since software engineering applications supposed to run on top of GOODSTEP are required not to be suspended when performing the propagation of changes in the database, we opted for implementing conversion functions as a lazy database transformation. The main problem encountered in implementing a lazy database transformation has been to ensure the equivalence of results with respect to the corresponding immediate database transformation [10, 9].

[2] The ODBSs GemStone [6] and ObjectStore [7] use a similar approach.

Object migration refers to the possibility for an object to change its class during its lifetime. This kind of migration is particularly useful when new classes are added to the schema which are more appropriate classes for existing objects. Migration of objects is also very useful when a class has to be deleted and the designer wants to retain objects of that class. Before deleting the class, objects have to be migrated appropriately.

3.1 Designer-defined Conversion Functions

The transformation of a database object means changing its internal structure and its value. Changing the object internal format is the task of the database system and it is done automatically without the need for manual assistance by the schema designer. Changing the object value is mainly an application-dependent task. The solution for the schema designer or application builder is to write a conversion function which tells the database systems how to change the value of the affected objects, and then to associate such a conversion function to the appropriate class that has been changed. If no conversion function for a modified class is provided, the system uses a default conversion function for the transformation.

Let us consider the designer has defined the following *CAD* schema:

```
class Circle type tuple (color: string,
                         center_position: real,
                         diameter: real)
end;

class Colored_Circle inherit Circle
end;
```

If the designer wants to modify this schema and changes the attribute *diameter* of class *Circle* from *diameter* to *radius*, he/she will have to write a conversion function that tells the system how to compute the value of *radius* and how to assign the newly computed values to all objects of class *Circle* already existing in the database. This is exactly what is done by the `mod_Circle` conversion function associated to the updated class *Circle* hereafter:

```
modify class Circle type tuple (color: string,
                                center_position: real,
                                radius: real)
with conversion mod_Circle {
     new->radius = old->diameter / 2;
}
end;
```

In the conversion function `mod_Circle`, `old` refers to an object conforming to the format of class *Circle* before the modification has been performed, whereas `new` refers to the same object after its modification. In the conversion function the transformation is not defined for all the attributes of class *Circle* but only for the attribute *radius*. This is because we assume that default conversion functions [8] are *always* executed on objects *before* a designer–defined conversion function is executed. This simplifies the writing of designer–defined conversion function.

In case of a modification to a class which is propagated to subclasses, it is often desirable to be able to associate a conversion function to the subclasses as well. For more details the reader is referred to [8].

Conversion functions can be associated to the basic primitives provided by O_2, as well as to all *HLPs* which modify the structure of classes in the schema.

3.2 Designer-defined Migration Functions

Object migration refers to the possibility for an object to change its class during its lifetime. Migration of objects is obtained through associating a migration function to a class which is responsible to migrate part of the objects of the class extension to subclasses only. The restriction to migrate objects to subclasses stems from the fact that the data model of O_2 is based on covariance and migration of objects to any class in the schema may lead to run-time type errors.

Let us assume that after the creation of class *Colored_Circe* the designer needs to migrate objects from class *Circle* to *Colored_Circle*. To perform the operation, the designer should provide a *migration function* as follows:

```
migration function migr_Circle in class Circle
{    if ((self->color <> ``´´) &&
         (self->color <> ``transparent´´))
     {
           self->migrate(``Colored_Circle´´);
     }     }
```

The public method `migrate` is provided with the root class *Object* belonging to every schema. This method is therefore inherited in all classes of the schema and can be invoked on each object in the database. Task of the method is to change the object's class identifier stored in the header of the object, to restructure and reinitialize the value of the object so that it conforms to the definition of the target class.

A migration function should be considered in the same way as a conversion function, i.e. for schema evolution purposes only. When defined and associated to a class, a migration function is supposed to be executed immediately, at least from a logical point of view. As explained in Section 3 problems with conversion functions apply for migration functions as well. The solution is to execute migration functions using a *lazy* modality, i.e. only when objects are accessed by applications. The reader is referred to [8] for implementation details on conversion and migration functions.

4 "Simulating" Schema Updates through Views

An alternative use of a view definition, which can be considered as a virtual schema definition, is to use it for *simulating some* classes of schema modifications. The responsible schema designer could, instead of invoking a sequence of schema changes, define a virtual schema, test if this virtual schema reflects his/her expected modifications, and then *materialize* the virtual schema in order to make it a real one. In our perspective, materializing a virtual schema means to translate the view definition commands into schema update commands, both basic primitives and

HLPs, and conversion/migration functions which have to be applied on the base schema from which the virtual one has been defined.

Using *declarative* **V**iew **D**efinition **L**anguage (VDL) statements, a schema along with the associated database(s) can be modified, avoiding the use of the underlying programming language, as it is with conversion functions, as far as possible.

4.1 The Process of Simulation and Materialization

The *simulation phase* starts with the execution of an O_2 command which invites the designer to specify a view by means of a script to be written in the *VDL* which should simulate the intended schema modification. Together with the definition of this view – the eventual future schema –, a corresponding *simulation base*[3] should be created which works as some kind of "scratchpad". Otherwise, modifications of data performed by applications running against the view might corrupt the production data which is generally not intended.

After the simulation base has been set (this is done implicitly with the creation of the simulation base), all database accesses which were launched through the view are bypassed to this new base. The virtual base then has to be filled either with an extract of the related production/source base or with a complete copy of it. The newly created view together with the source schema and the simulation base form a *simulation scope* illustrated in Figure 2.

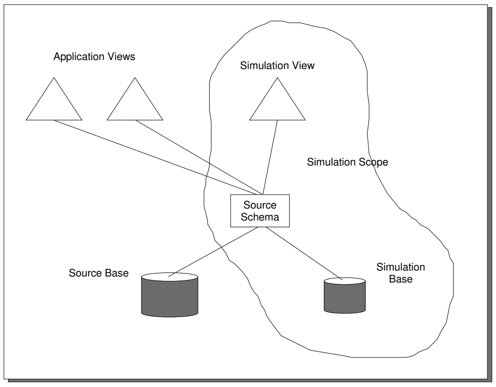

Fig. 2. The simulation scope.

After the instantiation of the simulation environment, the schema designers may work interactively with the simulation database, or they can define applications which run on top of the view. The advantage is that applications which run in production mode are not affected because the real database is not blocked.

It may be necessary to have several iterations on the process of defining a view which reflects all the desired features of the schema update. This is due to the fact, that schema designers might want to "play" with some alternatives before they decide which one to choose. It should even be possible to test several views in parallel. The simulation process ends either with no result, meaning that all views and the simulation bases have to be discarded, or a view definition has been

[3] Which is intended to be a portion or a *small* subset of the original base.

found to be the desired solution. In this case the simulation phase ends and the materialization phase has to be invoked. Once all designers are satisfied with the view established in the simulation process, it ought to be *materialized*.

Materialization in this context means to *transform* the source schema according to the view into a new target schema and to migrate or convert the affected objects in the source base so that they conform to the new structures present in the class hierarchy of the target schema. Materialization is a function which takes the original schema, the original base and the view – the selected VDL's script corresponding to the view – as input and produces a transformed schema and converts the base. The function has to invoke a translator, which maps the view definition together with the original schema definition into schema update primitives (both basic and *HLPs*), conversion and migration functions which are then automatically executed. These different steps are illustrated on the synopsis in Figure 3.

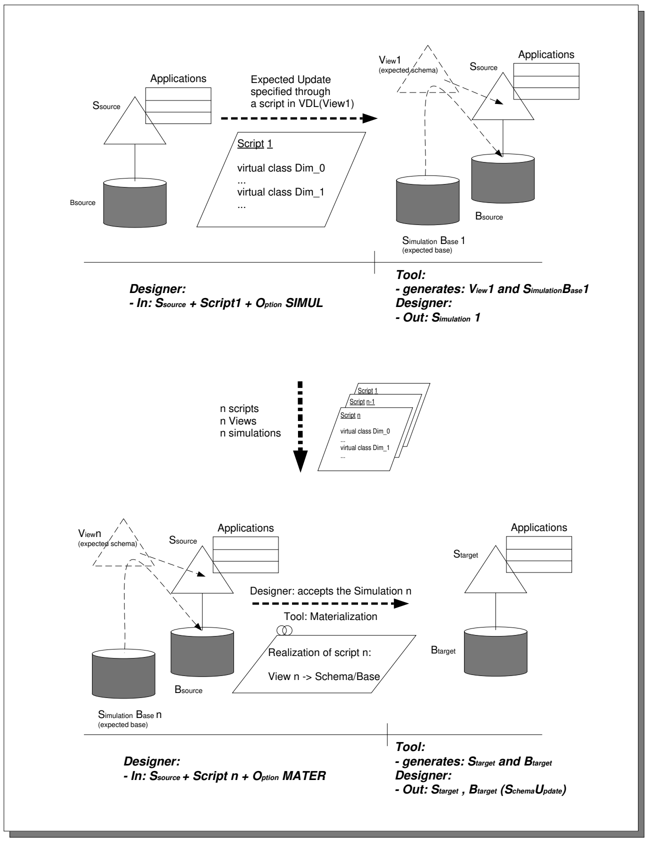

Fig. 3. Synopsys of Schema Change through Views.

4.2 Example Simulating Abstraction-Specialization

In this section a simulation example based on the operation Abstraction-Specialization is presented. The simulation of an operation starts by finding appropriate

VDL statements which realize a view simulating the desired changes/updates. Afterwards it ought to be possible to materialize the view or in other words to perform the schema change process physically. In addition it might be necessary to extract functions which convert or migrate affected objects, i.e. propagate the schema changes onto the base(s).

Let us consider a slightly modified version of the example from Section 2.3. The class *Figure* holds an additional attribute, *dim* of type `string`, giving information about an object's dimension. This information could be a one word description like "dim_1" or "dim_2". Furthermore, all instances of class *Figure* and of its subclasses are supposed to be stored in the persistent set *The_Figures*.

The intention of the designer in the Software Engineering (SE) process therefore is, to insert new classes into the schema which refine the concept of a "Figure" according to the dimension of the different figures kept in the database. These are the classes *Dim_0*, *Dim_1* and *Dim_2*.

The simulation step We can simulate the designer intention by creating the following view:

```
virtual schema CAD_view from schema CAD
virtual class Dim_0 includes Point where
      (select e from The_Figures where e.dim="dim_0")
virtual class Dim_1 includes Line, Spline where
      (select e from The_Figures where e.dim="dim_1")
virtual class Dim_2 includes Polygon, Circle where
      (select e from The_Figures where e.dim="dim_2")
```

With this definition the virtual classes would be populated with all objects in *The_Figures* satisfying the corresponding query and with those objects which would belong to the extension attached to one of the classes specified in the `includes` statement. For instance, *Dim_1* would hold those objects in *The_Figures* which contain the string "dim_1" in the attribute *dim* and those which are member of the extension of class *Line* or *Spline*. The positioning of the virtual classes in the inheritance hierarchy is automatically performed by the system [1].

The materialization step Assuming the designer is satisfied with such a view after the simulation phase, the materialization of that view can be triggered by means of the command: `materialize view CAD_view;`

Generally, the insertion of a class in the middle of a hierarchy refers to two design concepts at once. First, class *Figure* is **specialized** into the subclassses *Dim_0*, *Dim_1* and *Dim_2* and second the existing classes *Polygon*, *Circle*, *Spline*, *Line* and *Point* are **generalized** to the classes *Dim_2*, *Dim_1* and *Dim_0* respectively.

This development step is covered by the *HLP* **Abstraction-Specialization**:

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

According to the object's shape, members of class *Figure* have to be migrated to the newly created classes. The materialization has to call a migration function, which transfers the corresponding objects, namely those selected by the queries, to the specialized classes:

```
migrate objects of class Figure
{
  if ( self-> dim =="dim_0")
    self->migrate ("Dim_0");
  else if ( self->dim =="dim_1")
    self->migrate ("Dim_1");
  else if ( self->dim =="dim_2")
    self->migrate ("Dim_1");
}
```

This migration function might be deduced from the VDL statements at the materialization step. Migration functions should be provided whenever specialization occurs within the development process. This problem does not occur when dealing with generalizations since they only lead to new, abstract classes, i.e. class membership is not affected in these cases.

The materialization step ends with a new schema and new database corresponding to the intented update of the designer.

4.3 Can all schema modifications be simulated?

A very important question to be answered when performing simulations of schema modifications is the following: can the designer really simulate *all* kinds of schema modifications ? The answer is not trivial and depends strongly on how powerful the view mechanism used for simulating schema changes is. The simulation of schema modifications implies also the possibility to run applications on top of the simulation view which are allowed to read and update data in the simulation base.

Views as defined for relational databases are mainly information retrieval oriented, i.e. few possibilities are given to applications running against a view to update the real data. Views in an ODBMS allow both an efficient retrieval of information and the possibility for applications to update the real data [15, 12]. They are therefore much more suitable for simulation purposes which, de facto, imply data to be read but also to be modified.

In GOODSTEP we can define both read and write virtual properties of a class in a view. Read only virtual properties are not stored in the database and are computed on demand. Write virtual properties are stored in the simulation database to be able to update and retrieve them appropriately[4].

Consider the schema *CAD* presented in Section 3.1 where the attribute *radius* has been added to class *Circle* and its value has been initialized by a conversion function to be: `radius` = `diameter` / 2. The value of the newly added attribute clearly depends on the value of another attribute in the schema. This can be elegantly expressed by the view mechanism as follows:

```
            virtual read attribute radius: real in class Circle
has value  self->diameter / 2;
```

The fact that the virtual attribute is declared as a read attribute implies that its value is computed on demand, without being forced to store the value in the

[4] In special cases, for efficiency reasons, virtual information which is read only is stored as well.

simulation database. For write attributes this is no more valid and the value has to be stored in the simulation database. In fact, it is not always possible to infer a backward transformation in order to propagate the change to the source data[5].

Consider, for instance, primitives like

```
create class Rectangle
public type tuple (center_position: real,
                   height: real,
                   width: real)
end;

create attribute description: string in class Circle;
```

These schema modifications address the addition of information (class *Rectangle* and attribute *description*) which is independent from any other information in the schema. Using the GOODSTEP simulation mechanism, new virtual classes and attributes can be added in the simulation view. The initialization of the data is not computed from existing information, but default values are assigned to the new virtual class (an empty extension) and an initial default value to the new virtual attribute (an empty string “”). Simulation applications can update the value of virtual data which must be stored in the simulation database. As far as the deletion of information is concerned, it can be simulated using the *hide* command provided by the VDL. In the simulation database, the hidden information is either not present (in this case the simulation database is smaller than the source one) or filtered out by the view itself.

As already shown in the example in Section 4.2, *HLPs* like specialization or generalization of classes can be simulated elegantly using the VDL.

5 Conclusions

In this paper we presented the features which are under development in the GOODSTEP project for enhancing schema evolution in the O_2 database system. We first introduced the concept of high level primitives which improve the declarativity of how complex schema changes can be be performed. Further, we described how propagation of schema changes to the database can be obtained via designer-defined conversion and migration functions. We emphasized the need to perform the transformation lazily, i.e. without requiring application shutdown. Finally, we indicated how the designer can make use of the view mechanism for simulating schema changes. If the designer is satisfied with the simulations, the view is materialized mapping statements written in an extended VDL to both schema update primitives and conversion/migration functions.

As far as the implementation is concerned, high level primitives have been implemented as an application on top of O_2. On the other hand, conversion functions and migration functions have been implemented changing the Schema Manager and the Object Manager of the O_2 ODBMS itself. The implementation of the simulation mechanism is currently under development.

[5] Consider for instance the simple function $z = f(x, y)$. An update on z does not definitely determine the update on x and on y.

References

1. S. Abiteboul and A. Bonner. Objects and views. In *SIGMOD'91 Conference Proceedings, Int. Conf. on Management of Data*, San Francisco, California, Mar 1991. ACM Press.
2. E. Amiel, M. Bellosta, E. Dujardin, and E. Simon. Supporting Exceptions to Behavioral Schemas Consistency to Ease Schema Evolution in Oodbms. In *Proceedings of the 20th International Conference on Very Large Databases.* Morgan Kaufmann, September 1994.
3. J. Banerjee, H. Kim, W. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. In U. Dayal and I. Traiger, editors, *ACM–SIGMOD '87 Conference on Management of Data, Conference Proceedings*, pages 311–322, San Francisco, California, February 1987. ACM Press.
4. M. Blaha, F. Eddy, W. Premerlani, W. Lorensen, and J. Rumbaugh. *Object Oriented Modeling and Design.* Prentice–Hall International Editions, 1991.
5. P. Brèche and M. Wörner. Schema Update Primitives for ODB Design. Technical Report 07/95, Fachbereich Informatik, Universität Frankfurt, May 1995.
6. R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. Lockovsky, editors, *Object–Oriented Concepts, Databases and Applications*, chapter 12. ACM Press, 1989.
7. O. Design Inc. *Objectstore User Guide, chapter 9.* Object Design Inc., 1993.
8. F. Ferrandina, G. Ferran, J. Madec, T. Meyer, and R. Zicari. Database Evolution in the O_2Database System. In *Proceedings of the 21st International Conference on Very Large Databases*, September 1995.
9. 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.
10. 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.
11. I. Itasca Systems. OODBMS Feature Checklist. Rev. 1.1. Technical Report TM–92–001, Itasca Systems, Inc., December 1992.
12. C. Laasch, M. Scholl, and M. Tresch. Updatable views in object–oriented databases. In *Proc. of the 2nd International Conference on Deductive and Object–Oriented Databases*, number 566 in LNCS, Munich, Germany, December 1991. Springer Verlag.
13. O2 Technology. *O2 Database User Manual, Version 4.0.* O2 Technology, Versailles, France, June 1992.
14. O2Technology, Inria, Cefriel, U. of Frankfurt, and U. of Grenoble. Final requirements for the database repository. Goostep project 6115. Goodstep report. GO.8R.1.0. Technical report, Goodstep consortium, March 1994.
15. E. Rundensteiner. Multi View: A Methodology for supporting Multiple View Schemata in Object–Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 187–198, Vancouver, Canada, August 1992. Morgan Kaufmann.
16. M. Scholl and M. Tresch. Schema Transformations without Database Reorganization. *ACM SIGMOD Record 1993*, 22(1):21–27, March 1993.
17. 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.