Management of Schema Versions and Versions of Schema Instance in a Multiversion Database [1]

Marie-Jo Bellosta*
Robert Wrembel**
Geneviève Jomier*
*University Paris Dauphine
Paris, France
e-mail {bellosta, jomier}@lamsade.dauphine.fr
**The Franco-Polish School of New Information and Communication Technologies
Poznan, Poland
e-mail wrembel@efp.poznan.pl

Preliminary Version

1 Introduction

An object database is defined by a schema and an instance of the schema. The schema describes the structure of the data stored in the database, including classes, types associated with classes, inheritance relationships between classes, and method interfaces. The instance of the schema contains all objects stored in the database, and method implementations. Expressiveness of object database models is the main reason of their wide success in the CAD/CAM, CSCW and CASE applications. Moreover, these applications require database versioning in order to view and manage different states of the modeled universe. Such databases are called *multiversion*. The main problem of multiversion databases is to recognize which versions of different multiversion entities, i.e. classes or/and objects, are consistent, i.e., go together. As shown in [CJ90], a user is quickly lost if he needs to manage explicitly consistency between all versions of both classes and objects without some efficient support from the system.

Existing systems use two approaches to provide versioning mechanism. In the first one [SG94], [CJ90], [Kat90], the versioning mechanism is applied only to objects and not to the database schema. In the second one, the versioning mechanism is applied either to separate classes, [TK89], [TOC93], [MS93], or to sets of classes [SZ86], [ALP91], or to the entire schema [BKKK87]. The objects stored in the schema instance have versions associated with their class versions. Both approaches are not satisfactory since they provide no solution to manage together versions of the schema and versions of the schema instance.

[1] This research is partially supported by: INRIA-Rocquencourt, the grant No. 6548 from the French Embassy in Poland and the Polish Committee for Scientific Research (KBN) concerning the scientific cooperation, and the grant No. PB 1173/T11/95/09 from the Polish Committee for Scientific Research (KBN).

In this paper, we propose a formal model of multiversion database systems. It is based on the *database version* approach [CJ90], [CJK91]. This approach constitutes the first step to the management of both schema versions and versions of schema instances. The authors introduced the database version mechanism and focused only on the versioning mechanism applied to complex objects. We extend this approach to take into account any type constructors and inheritance relationships. We propose a set of functions which constitute the kernel of the database version approach to manage both schema versions and versions of schema instances. The main advantages of this approach are extensibility and adaptability. Moreover, this model offers a rigourous framework for the development of new multiversion functionalities such as evolution support, view mechanism, query processing and integrity constraints.

This paper is organized as follows. Section 2 defines a monoversion object database model. This model is used for illustration purposes, and our formal definition of the *database version approach* can be applied on others object database models. Section 3 briefly outlines the *database version* approach and presents how this approach manages the inheritance relationship. Section 4 formally defines the *database version* model; only structural aspects are considered. Section 5 concludes the paper.

2 A Monoversion Object Database Model

This section presents a monoversion object database model that is the basis of the formal definition of *database version* approach described in Section 4. Because there is a lack of a unified object database model, we introduce a notation for classes, types and methods. Then, we define the notion of a *well-formed* class hierarchy and a *well-formed* set of methods which constraints the semantic of the object model. The main criterion guiding these definitions is the static type checking, i.e., no run-time error can occur. Finally, we conclude by the definitions of a schema and a schema instance. The notations used in the paper are inspired by the formalism presented in [ABDE96], [AHV95] and [ADL91].

2.1 Preliminaries

Object and Class. An *object* represents a real world entity. An object has an identifier called *oid* and a *value*. An identifier is unique and invariable. The set of oid's is denoted as $\mathcal{O}id$. A value may be atomic or complex. This complex value may involves oids, i.e., references to objects. The set of values is denoted as $\mathcal{V}alue$. Similar objects are grouped together by describing their common characteristics in a *class*. Thus, a class forms a template form which individual objects, called *instances*, can be created [KM94]. Classes are components of a database schema. The set of classes is denoted as $\mathcal{C}lass$.

Example 2.1 Consider a schema of geometric figures. This schema contains three classes: *Figure*, *Circle* and *Ellipse*. A figure, an ellipse, and a circle have a color and a center position. Additionally, an ellipse and a circle have a diameter. Moreover, an ellipse has an additional diameter, called lateral diameter and an orientation indication. Finally, the *Circle* class has two instances : $circle_1$ and $circle_2$. □

Structural type. A *Structural type* may be associated with a class to describe the *structure* of this class. The set of structural types is denoted as $StructuralType$. A structural type is either *atomic* or *complex*. *Atomic* structural types are: integer, float, boolean, char or string. *Complex* structural types are represented using different *type constructors* such as tuple, set and list; the two later constructors are called *collections*. The definition of complex structural types are:

1. **A tuple type** is denoted as $T = [\ldots, a_i\colon \tau_i, \ldots]$, where a_i is the ith distinct attribute name of T and τ_i is the type of this attribute which may be a structural type or a class.
2. **A set type** is denoted as $T = \{\tau\}$ where τ is the type of its elements which may be a structural type or a class.
3. **A list type** is denoted as $T = <\tau>$ where τ is the type of its elements which may be a structural type or a class.

The structural types and the classes are in the object database schema. When we have a structural type or a class, we group them through the notion of *type*. The set of types is denoted as $\mathcal{T}ypes$.

Example 2.2 The structural type *FigureType* associated with the class *Figure* is a tuple type where the type of the attribute *color* is *string*, and the type of the attribute *centerPoint* is the class *Point*. Moreover, we define the structural type *Design* as a set of figures:

$$FigureType = [color\colon string, centerPoint\colon Point]$$

$$Design = \{Figure\}.$$

□

Subtyping relationship. Given T_1 and $T_2 \in \mathcal{S}tructuralType$, a subtyping relationship between T_1 and T_2 where $T_2 \leq T_1$ is defined as follows:

1. If T_1 is a tuple type, $T_1 = [a_1{:}\tau_{11}, a_2{:}\tau_{21}, \ldots, a_n{:}\tau_{n1}]$ then T_2 has to be a tuple type defined as $T_2 = [a_1{:}\tau_{12}, a_2{:}\tau_{22}, \ldots, a_n{:}\tau_{n2}, \ldots, a_p{:}\tau_{p2}]$, where n $\leq$ p and $\forall i \in \{1, \ldots, \text{n}\}$ $\tau_{i2} = \tau_{i1}$.
2. If T_1 is a set type, $T_1 = \{\tau_1\}$, then T_2 has to be a set type defined as $\{\tau_2\}$, where $\tau_2 = \tau_1$.
3. If T_1 is a list type defined as $<\tau_1>$ then T_2 has to be a list type defined as $<\tau_2>$, where $\tau_2 = \tau_1$.

The subtyping relationship is constraint by the assumption of the static type checking, as shown in [KM94] and [ABDE96].

Example 2.3 The subtyping relationships *CircleType* $\leq$ *FigureType* and *EllipseType* $\leq$ *CircleType* and *EllipseType* $\leq$ *FigureType*, are deduced from the definitions of $EllipseType$ and $CircleType$:

$$CircleTye = [color\colon string, centerPoint\colon Point, diameter\colon float]$$

$$EllipseType = [color\colon string, centerPoint\colon Point, diameter\colon float, lateralDiameter\colon float, orientation\colon integer]$$

Moreover, we create the following collection types, $Figures = \{Figure\}$ and $Circles = \{Circle\}$. We deduce that $Figures$ may be a subtype of $Design$. Since the element type of $Circles$ and $Figures$ are not the same, $Circles$ is not a subtype of $Figures$. $\Box$

Inheritance relationship. An *inheritance relationship*, noted as $\prec$, is a partial order on *Class* that forms a direct acyclic graph - DAG. The class *Object* is the root of this hierarchy and it defines the common properties of all the objects. Given two classes C_1 and $C_2 \in \mathcal{C}lass$, if $C_1 \prec C_2$ then C_2 is a *superclass* of C_1 and C_1 is a *subclass* of C_2.

The function $superClass$ from $\mathcal{C}lass$ to $\mathcal{C}lass$ returns all the *direct* superclasses of a class, defined as:
$\forall\ c \in \mathcal{C}lass$, $superClass(c) = \{\ c_i \mid c_i \in \mathcal{C}lass$ and $c \prec c_i$ and $\nexists\ c_j \in \mathcal{C}lass$ with $c \prec c_j \prec c_i\}$.

Well-formed class hierarchy. A class inheritance hierarchy is a 3-tuple:

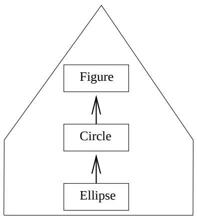
$$< \mathcal{C}lass, representation, \prec>, \text{ where:}$$

- $\mathcal{C}lass$ is the set of classes;
- $representation$ is a function from $\mathcal{C}lass$ to $\mathcal{S}tructuralType$ defined as :
  $\forall\ c \in \mathcal{C}lass,\ representation(c) = \tau$ where $\tau \in \mathcal{S}tructuralType$;
- $\prec$ is an inheritance relationship on $\mathcal{C}lass$.

A class hierarchy is *well-formed* if it abides to the *structural consistency rule* defined as:
let $b,c \in \mathcal{C}lass$; $c \prec b \Rightarrow representation(c) \preceq representation(b)$.
This rule means that a class c is the subclass of b if the type associated with c is a subtype of the type associated with b.

Note that, this rule implies that the *Object* class has only methods. For instance, if the *Object* class has a type using the type constructor *tuple*, then all its subclasses have to use the same type constructor.

Example 2.4 The class hierarchy, defined in Figure 1, is well-formed. All the subclasses of class *Figure* respect the structural consistency rule. Two inheritance relationships are defined: $superClass(Circle)$= $Figure$ and $superClass(Ellipse)$= $Circle$. In this case, all the properties of class *Figure* are implicitly inherited by its subclasses. $\Box$




```
class Figure
type tuple [ color : string,
             centerPoint : Point ]
 ...
end;
```

```
class Circle inherit Figure
type tuple [ diameter : float ]
 ...
end;
```

```
class Ellipse inherit Circle
type tuple [ lateralDiameter : float,
             orientation : integer]
 ...
end;
```

Figure 1: The Class Hierarchy of Geometric Figures

Operation. An operation m represents an action that can be performed by the class instances of some classes in a class hierarchy.

For instance, the *geometric figure* class hierarchy may have operations such that $move$, $copy$, $equal$, $changeColor$, or $resize$.

Built-in operations. Each structural type associated with a class has a set of *built-in operations* that enable to manipulate (i.e., access and update) the state of instances of that class. For our purpose, we consider a subset of the operations defined in the ODMG object model [RC94].

For instance, any collection has the following *built-in operations*: $insert\_element$, $remove\_element$ or is_empty.

Set of Methods. A set of *methods*, $\{(interface\ of\ m_k,\ implementation\ of\ m_k) \mid k \in \{1, \ldots, \mathrm{n}\}\ \}$, corresponds to an operation m where k denotes the kth method, $interface\ of\ m_k$ is $m_k(\tau_k^1, \ldots, \tau_k^n) \rightarrow \tau_k$ and $implementation\ of\ m_k$ is the code which realizes this method. We assume that a unique implementation is associated with a method interface. We denote the type of the ith formal argument as τ_k^i, and the type of the result as τ_k. The *signature* of the method m_k is $(\tau_k^1, \ldots, \tau_k^n)$. The set of method interfaces is denoted as $\mathcal{M}Interface$ and the set of method implementations is denoted as $\mathcal{M}Implementation$. The first one is an element of the schema while the second one is an element of the schema instance [AHV95].

Example 2.5 Suppose that we wish to implement the operation *copy* on *Figure*, *Circle*, and *Ellipse*. Since *copy* is implemented differently for each of these classes, three methods are defined with the following interfaces:
$copy_1(Figure) \rightarrow\ Figure$; $copy_2(Circle) \rightarrow\ Circle$; $copy_3(Ellipse) \rightarrow\ Ellipse$. □

Built-in methods. *Built-in methods* perform built-in operations on each structural type. Note that the structural consistency rule avoids run-time error when using these built-in methods, as shown in [KM94].

For instance, all sets have the following built-in methods whose interfaces are defined so that: $insert\_element(T, T_1)$; $remove\_element(T, T_1)$; $is\_empty(T) \rightarrow Boolean$.

Operation Invocation. An *operation invocation* is denoted as $m(\tau_1, ..., \tau_n)$, where m is the operation, $(\tau_1, \ldots, \tau_n)$ is the signature of the invocation and the τ_i's represent the types of variables or operation invocations passed as arguments.

For instance, *copy* $(circle_1)$ denotes the invocation of the operation *copy* on the instance $circle_1$.

Substitutability. Given two classes C_1 and C_2, an instance of C_2 can substitute an instance of C_1 if and only if $C_2 \preceq C_1$.

The substitutability property is the basis of inclusion polymorphism [CW85]. It relaxes the condition of type equality on variable assignments or parameter passing to take into account an inheritance relationship.

Precedence relationship. A *precedence relationship* is a partial order defined on a set of methods [ADL91]. We choose the inheritance relationship between the type of the first signature argument to determine the partial order. Note that this type is always a class.
Given two method interfaces $m_i(\tau_i^1, \ldots, \tau_i^n) \rightarrow \tau_i$ and $m_j(\tau_j^1, \ldots, \tau_j^n) \rightarrow \tau_j$, m_i is more specific than m_j, denoted as $m_i < m_j$, if $\tau_i^1 \preceq \tau_j^1$.

For instance, consider again the method interfaces of the operation *copy*:
$copy_1(Figure) \rightarrow\ Figure$ and $copy_2(Circle) \rightarrow\ Circle$. We determine that $copy_2 < copy_1$ since $Circle$ is a subclass of $Figure$.

Method resolution. The method resolution is the process that selects the method to execute an invocation of an operation. The selection of the corresponding method follows a two-step process

[Laj93]. At compile-time, the compiler determines a set of applicable methods. Based on the types of the invocation arguments known at compile-time, it uses the substitutability property between the invocation arguments and the arguments of method interfaces. It generates an error if no applicable method maps onto the invocation. At run-time, the precedence relationship between applicable methods is used to select what is called the *most specific applicable* method, *MSA* for short. Note that this selection is based only on the first argument : there is no check that the other arguments of the invocation are subtypes of the corresponding formal arguments of the selected method.

The function $interface$ from $\mathcal{C}lass$ to $\mathcal{M}Interface$ returns all the interfaces of the applicable methods to an instance of a class:
$\forall\, c \in \mathcal{C}lass,\ interface(c) = \{\ m_k(\tau_k^1, \ldots, \tau_k^n) \ \rightarrow\ \tau_k\ \}$, where $m_k(\tau_k^1, \ldots, \tau_k^n) \ \rightarrow\ \tau_k \in \mathcal{M}Interface$ and m_k is defined locally in c or in a superclass of c.

Example 2.6 Consider the operation *equal* on the *geometric figures* class hierarchy. Since equality is defined differently for these classes, three methods are needed. Their interfaces are :
$equal_1(Figure, Figure) \rightarrow\ Boolean$, $equal_2\ (Circle, Figure) \rightarrow\ Boolean$
and $equal_3(Ellipse, Figure) \rightarrow\ Boolean$.
On the invocation $equal\ (Circle, Ellipse)$, the method resolution will select method $equal_2$ based on the first argument. $\Box$

Well-formed set of method interfaces. A set of method interfaces, denoted as $\mathcal{M}Interface$, is *well-formed* if it obeys to the following rules:

1. **Contravariance of input arguments.** Given two method interfaces $m_i(\tau_i^1, \ldots, \tau_i^n) \ \rightarrow\ \tau_i$ and $m_j(\tau_j^1, \ldots, \tau_j^n) \ \rightarrow\ \tau_j$ where $m_i < m_j$ then $\forall k \geq 2,\ \tau_j^k \preceq \tau_i^k$
2. **Covariance of returned argument.** Given two method interfaces $m_i(\tau_i^1, \ldots, \tau_i^n) \ \rightarrow\ \tau_i$ and $m_j(\tau_j^1, \ldots, \tau_j^n) \ \rightarrow\ \tau_j$ where $m_i < m_j$ then $\tau_i \preceq \tau_j$
3. **Unicity of the most specific applicable method.** Given an invocation $m(\tau_1, ..., \tau_n)$, if there is an applicable method, then there exists a Most Specific Applicable method and this method is unique.

The above rules guarantee that no type error can occur during the execution of a method code. The *covariance of return argument* is called consistency in [ADL91]. The *contravariance of input arguments* was originally developed for subtyping of functions [Car84], and has been extended to subtyping of partially targeted methods in [McK92] and [Dan90].

Example 2.7 The following set of method interfaces is well-formed:
$\{equal_1(Figure, Figure) \rightarrow\ Boolean,\ equal_2\ (Circle, Figure) \rightarrow\ Boolean,$
$equal_3(Ellipse, Figure) \rightarrow\ Boolean,\ copy_1\ (Figure) \rightarrow\ Figure,$
$copy_2\ (Circle) \rightarrow\ Circle,\ copy_3(Ellipse) \rightarrow\ Ellipse\ \}$. $\Box$

Root of Persistence. A *Root of persistence*, *root* for short, is a schema element which confers persistence to objects: an object reachable by a root becomes persistent. In this way, persistence is also conferred to an object if this object is referenced by another persistent object. This policy has two main advantages :

1. An object may be temporary or persistent;

2. Persistence is independent of the class definition [Deu91].

A root may be an instance of a class or a type, and some roots may have the same domain. The set of roots is noted as $\mathcal{R}oot$. Finally, all persistent objects and all method implementations form an *instance of schema.*

The function *instanceOf* from $\mathcal{R}oot$ to $\mathcal{T}ype$ returns the type of a root which may be a structural type or a class:
$\forall\ r \in \mathcal{R}oot$, *instanceOf(r)* $= \tau$ where $\tau \in \mathcal{T}ype$.

Example 2.8 The schema of geometric figures has five roots of persistence: the circle drawn by Joe, named $JoeCircle$, is an instance of the class $Circle$; the logo of the Olympic Games has five circles and is an instance of the structural type $Circles$; the designs of Tom and $Mary$, named $TomDesign$ and $MaryDesign$, are instances of the structural type $Design$; the design of Tom, named $TomDesign$, is an instance of the structural type $Figures$:

name JoeCircle: Circle;
name OlympicGames: set(Circle);
name TomDesign: set(Figure);
name MaryDesign: set(Figure);
name TomMotif: Figures; □

2.2 Definition of a Schema

Using the formalism presented in [AHV95], a *schema S* is a 5-tuple:

$$S =< \mathcal{C}lass, representation, \prec, \mathcal{R}oot, \mathcal{M}Interface > \text{ where}$$

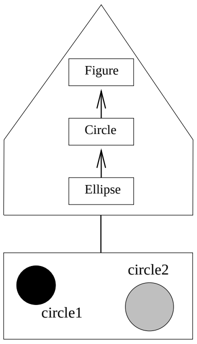
- $\mathcal{C}lass$ is the set of classes;
- $representation$ is a function from $\mathcal{C}lass$ to $\mathcal{T}ype$;
- $\prec$ is a partial order on $\mathcal{C}lass$;
- $< \mathcal{C}lass, representation, \prec >$ is a well-formed class hierarchy;
- $\mathcal{R}oot$ is the set of roots of persistence;
- $\mathcal{M}Interface$ is a set of well-formed method interfaces.

For instance, the figure 2 presents a schema with a well-formed class hierarchy and a set of well-formed methods.

2.3 Definition of a Schema Instance

Given a schema S = $< \mathcal{C}lass, representation, \prec, \mathcal{R}oot, \mathcal{M}Interface >$, an *instance of the schema* S, denoted as I, is a 4-tuple:

$$I =< assignment, value, extent, implementation >, \text{ where}$$




```
class Figure
type tpule [ color : string,
             centerPosition : Point ]
    method copy (FIgure): Figure,
    method equal (Figure, Figure),
    method changeColour (Figure, Color),
    method resize (Figure, float)
end;

class Circle inherit Figure
type tuple [ diameter : float ]
    method equal (Circle, Figure),
    method resize (Circle : float),
    method move (Circle : Point)
end;

class Ellipse inherit Circle
type tuple [ lateralDiameter : float,
             orientation : integer ]
    method equal (Ellipse, Figure)
end;
```

Figure 2: The Database of Geometric Figures

- *assignment* is a function from $\mathcal{C}lass$ to $\mathcal{O}id$ which assigns an object identifier *oid* to an object: $\forall\ c \in \mathcal{C}lass\ assignment(c) = oid$, where $oid \in \mathcal{O}ID$ with $\mathcal{O}ID = \cup \{\ assignment(c) \mid c \in \mathcal{C}lass\ \}$;
- *value* is a mapping from $\mathcal{O}id$ to $\mathcal{V}alue$ which maps each *oid* to a *value*: $\forall\ oid \in \mathcal{O}id\ value(oid) = v$, where $v \in \mathcal{V}alue$;
- *extent* is a function from $\mathcal{R}oot$ to $\mathcal{O}id$ which returns all the objects attached to a root of persistence: $\forall\ r \in \mathcal{R}oot\ extent(r) = \{\ oid\ \}$, where $oid \in OID$
- *implementation* is a function from $\mathcal{M}Interface$ to $\mathcal{M}Implementation$ which assigns an implementation to a method interface of m_k with respect to its signature and its return argument.

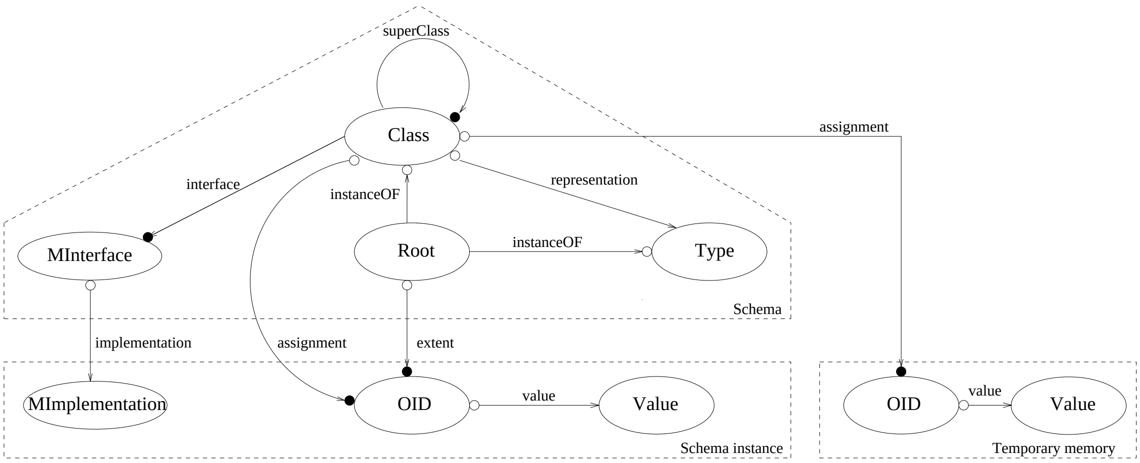


Figure 3: *Monoversion Database Model*

Figure 3 summarizes the previous functions (drawn as link) between the components (drawn as ovals) constituting a schema and its schema instance. A link may be optional (white circle) or mandatory. For example, since a root may have no object reachable through it, the $\mathcal{R}oot$ extremity of the *extent* link is a white circle. A black circle represents a multiple cardinality of a component in a link. For example, since a class has many method interfaces, the $\mathcal{M}Interface$ end of the *interface* link is a black circle. Moreover, we add a temporary memory to show that the transient objects are not in the schema instance.

3 The Database Version Approach

This section gives the main principles of the *database version* approach presented in [CJ90], [CJK91]. Then, the notions of a database version and multiversion database are formally defined. Finally, the database version approach is applied to manage the versioning of schema. We present (i) the choice of the meta-information used to manage the association between the types associated with class versions and to manage the inheritance relationship between class versions; (ii) the algorithm computing the function $superClass$ to take into account class versioning.

3.1 An overview of the Database Version Approach

The database version approach has been developped to provide a way to implement efficiently versions in general purpose DBMS. One of its main features which distinguishes it from other approaches is the distinction between the logical level, seen by users, and the physical level, managed by the system and hidden from users.

At the logical level, a multiversion database is seen as a set of *Database versions*, or DBV, identified by a database version identifier. Each database version, DBV_i, represents a consistent description of the modelled word. A database version is composed of one version of the schema and one version of the schema instance associated with this schema. Database versions are created by logical copying, called *derivation* : DBV_i is derived from the DBV_j either because the schema is versioned, or the schema instance is versioned, or both. The database versions may evolve independently from each other. This means that a database version may be updated or deleted without side effect on other database versions.

As a version of object/class may have the same value in several database versions, the conventional concept of *version of object/class* is split in two new concepts: a *logical version* of an object/class represents the version of this object/class as it seen by users in a given database version; a *physical version* of object/class is used by the system to store the value of all logical versions of this object/class having the same value. So, each version of schema consists of one logical version of each class while each version of schema instance consists of one logical version of each object. They are identified by a pair (object/class identifier, database version identifier).

At the physical level, the version management system allows to manage efficiently as many database versions as needed. Implementation details appear in [CJ90].

Example 3.1 In Figure 3.1, the database of geometric figures is a multiversion database, composed of three database versions:

- DBV_1 is the initial database version. Its schema is composed of the three class versions: $(Figure, DBV_1)$, $(Circle, DBV_1)$, and $(Ellipse, DBV_1)$. The associated version of its schema instance has two object versions : $(circle_1, DBV_1)$ and $(circle_2, DBV_1)$.
- DBV_2 is derived from DBV_1 to store new versions of objects, e.g., $(circle_2, DBV_2)$. Its schema remains unchanged.

- DBV_3 is derived from DBV_1 to store the new version of the schema. In this database version, class $Figure2D$ is added as the direct superclass of $Circle$ and the subclass of $Figure$. These modifications lead to conversion of all the objects of class versions $(Circle, DBV_3)$ and $(Ellipse, DBV_3)$, resulting in creation of new object versions, i.e., $(circle_1,\ DBV_3)$ and $(circle_2,\ DBV_3)$. Note that this database version constitutes an *alternative* of database version DBV_2.

□

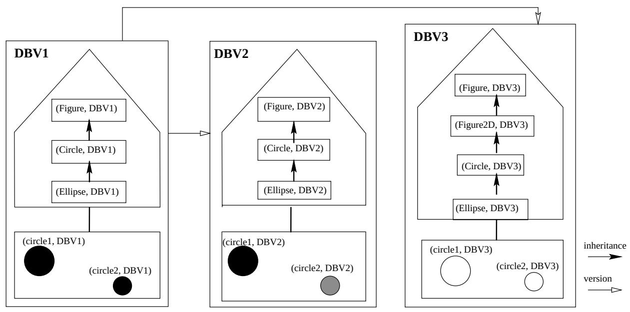


Figure 4: *Database Version Derivation*

Example 3.2 The Figure 5 illustrates the logical and physical level in showing the sharing of physical object/class in two cases:

1. In DBV_2, only the schema instance changes in comparison with DBV_1. At the schema level, the class versions share the same physical class, i.e., *PhysicalClassFigure1*, *PhysicalClassCircle1* and *PhysicalClassEllipse1*. At the level of the schema instance, logical objects *($circle_1$ DBV_1)* and *($circle_1$ DBV_2)*, share the same physical object, i.e., $physicalCircle_{1,1}$. The first version of $circle_2$, i.e., *($circle_2$, DBV_1)*, was updated to version *($circle_2$, DBV_2)*, thus a new physical version of this object is created, i.e., $physicalCircle_{2,2}$.

2. In DBV_3, both the schema and its schema instance change in comparison with DBV_1. At the schema level, $Figure2D$ is introduced in DBV_3 between its superclass $(Figure,\ DBV3)$ and its subclass $(Circle\ DBV_3)$. Physical class version *PhysicalClassFigure2D1* is implicitly created by the system. Class $(Circle,\ DBV_3)$ inherits all properties of its direct superclass, thus a new version is created, i.e. *PhysicalClassCircle2*. The $(Figure,\ DBV_3)$ class is unchanged thus both logical classes, *($Figure$, DBV_1)* and *($Figure$, DBV_3)*, share the same physical class version, named *PhysicalClassFigure1*. At the level of the schema instance, the circle objects have to be converted with respect to the new representation of class version $(Circle,\ DBV_3)$: the logical versions, *($circle_1$, DBV_3)* and *($circle_2$, DBV_3)*, have two new physical versions $physicalCircle_{1,3}$ and $physicalCircle_{2,3}$.

□

The user works in a multiversion database by selecting the database version of interest, DBV_i. After the selection of DBV_i, the user can treat it as a standard database without versions, querying and updating it. Operations on DBV_i are translated into actual physical operations by the version

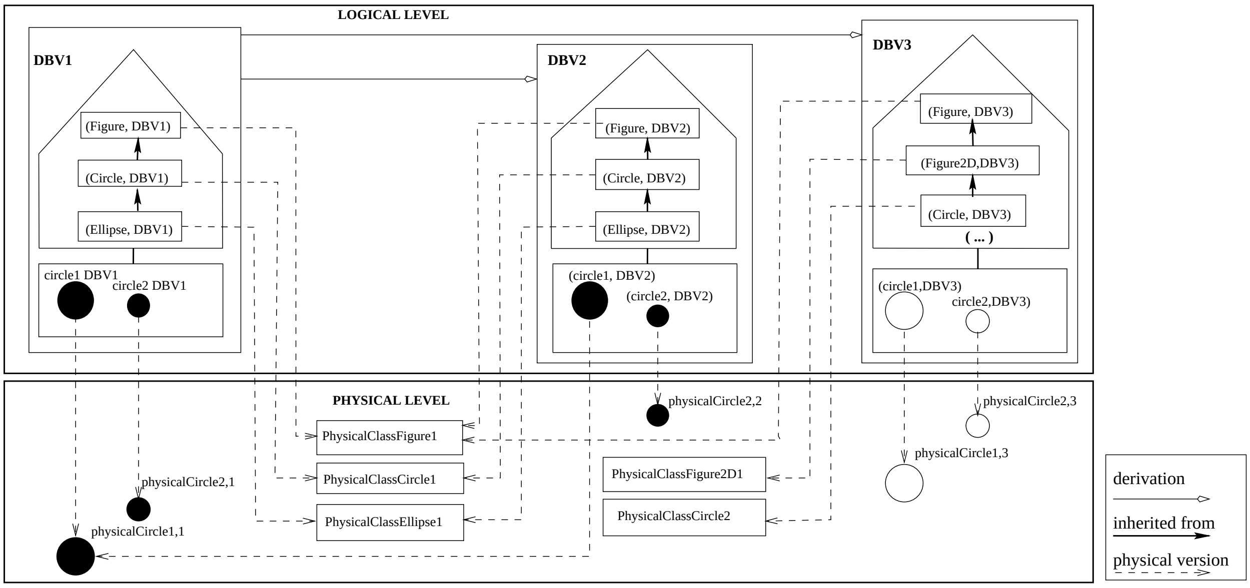


Figure 5: *A Multiversion Database*

management system, but the translation is transparent to the user. This transparency is one of the main advantages of the mechanism, and is absent in other existing versioning mechanisms.

Example 3.3 The type and the inheritance relationship of class $Circle$ is defined on its logical version in the specific database version, DBV_3, using the class definition as in a monoversion database :
Class $Circle$ inherit $Figure2D$
type tuple [diameter : float]
method equal(Circle, Figure),
...
end; □

3.2 Formal Definition of a Database version and a Multiversion database

Database Version. Given the set of schema versions - $\mathcal{S}chemaVersion$ and the set of versions of the schema instance - $\mathcal{S}chemaInstanceVersion$, a database version DBV is a couple defined as follows:

$$DBV = (sv, siv) \text{ where}$$

- $sv \in \mathcal{S}chemaVersion$,
- $siv \in \mathcal{S}chemaInstanceVersion$,
- given function *instantiation* from $\mathcal{S}chemaVersion$ to $\mathcal{S}chemaInstanceVersion$. This function maps one and only one version of the schema instance to one and only one version of the schema:
  $\forall\ sv \in \mathcal{S}chemaVersion\ instantiation(sv) = siv$ where $siv \in \mathcal{S}chemaInstanceVersion$.

Multiversion database. A *multiversion database*, denoted as $MVDB$, in the *database version* approach is defined as a 2-tuple:

$$MVDB =< \mathcal{D}atabaseVersion, assignment_{DBV} > \text{ where}$$

- $\mathcal{D}atabaseVersion$ is the set of database versions;
- $assignment_{DBV}$ is a function from $\mathcal{D}atabaseVersion$ to $\mathcal{DBVI}d$ that assigns an identifier to a database version:
  $\forall\ DBV \in \mathcal{D}atabaseVersion\ assignment_{DBV}(DBV) = dbvid$ where $dbvid \in \mathcal{DBVI}d$.

3.3 Type and Inheritance in the Database Version Approach

The main assumption guiding transparent management between the logical and physical level is that different logical versions of a class having the same value share the same physical version. The physical version stores this value which consists of a structural type description and an inheritance definition. The problem lies in the choice of the information used to guarantee a transparent and efficient management of classes without creating superfluous physical versions. We show that the solution is the same as the management of complex objetcs [CJ90]. That means, the references used in structural types and inheritance relationship are only the class identifiers. The system dynamically infers the logical version of classes participating in the structural type definition or in the direct superclasses of a logical version of a class. Then, it can access the physical version associated with each logical class version. These physical versions are the basis of transparent management of schema versions. For instance, the system can verify that the structural consistency rule between the physical version of the current class and the physical versions of its superclasses.

To illustrate our purpose, we focus on the algorithm which computes the superclasses of a logical class version named lvc. This alhorithm returns the physical versions associated with each logical class versions of the lvc superclasses. The first step investigates its physical version and gets the class identifiers of its superclasses by function $superclass_{\phi}$. The second step infers the logical version of each class identifiers using two functions: $logicalVersion$ which gives the logical version of a class from a class identifier and a database version identifier; DBV returns the database version identifier from a logical class version. Finally the physical version associated with each logical versions are searched.

$superClass(lvc)$
input: a logical version of a class in a database version, lvc
output: collection of the physical versions of the superclass of lvc in a DBV
Step 1:/*searches the identifiers of the direct superclass of lvc */
  pvc $\leftarrow$ ($physicalVersion(lvc)$) ;
  superclassIds $\leftarrow$ $superclass_{\phi}$(pvc);
Step 2:/*infers the logical versions and searches the physical version associated with each of them*/
  for each $cid_i \in superclassIds$ do
    $lvc_i \leftarrow logicalVersion(cid_i, DBV(lvc)$;
    $pvc_i \leftarrow physicalVersion(lvc_i)$;
    $insert\_element$ (result, pvc_i)
  end do ;
  return result ;
  end superClass

Example 3.4 In Figure 5, the logical version of class *Ellipse*, in the database version DBV_1, is associated with its physical version i.e., *PhysicalClassEllipse1*. This physical version contains the class identifier of class $Circle$ as direct superclass reference, identified by cid_2. In database version DBV_2, the physical version of the logical version of class *Ellipse* remains unchanged. The invocation of *superClass* on the logical version of class *Ellipse* determines its physical version, *Physical Ellipse1*. Then the algorithm searches the superclass references, i.e., the collection composed of the identifier of class $Circle$, cid_2. It infers the logical version in the database version DBV_2 : (cid_2, DBV_2). From this logical version, it can access the physical version of the logical version of class $Circle$, i.e., *PhysicalClassCircle1*. □

4 Multiversion Object Data model

This section presents the multiversion object data model. First, we define formally the notions of *multiversion*, *logical version* and *physical version* of an object and a class. Then, we specify the notion of a well-formed class hierarchy to take into account the versioning of classes. Finally, we define the schema version and the version of its schema instance.

4.1 Versions of Objects and Classes

Multiversion object. A *multiversion* object, $mvObject$ for short, has a unique identifier called *mvoid* and a set of *logical* versions. We note the set of *mvoid* as $\mathcal{M}vOid$.

Example 4.1 In Figure 5 $circle_1$, identified by $mvoid_1$, is a multiversion object with three logical versions: ($circle_1$ DBV_1), ($circle_1$, DBV_2), and ($circle_1$, DBV_3). □

Logical version of a *mvObject*. The *logical version* of a $mvObject$, $lvObject$ for short, represents its object version in a specific database version. This *logical version* is identified by a unique pair (*mvoid*, *dbvid*), denoted $lvoid_{mvoid,dbvid}$, where *dbvid* is the database version identifier.

The *logicalVersion*$_{mvo}$ function from $\mathcal{M}vOid \times \mathcal{D}bvId$ to $\mathcal{M}vOid \times \mathcal{D}bvId$ returns the logical version of a $mvObject$ in a specific database version:
$\forall$ $mvoid \in \mathcal{M}vOid$ and $\forall$ $dbvid \in \mathcal{D}BVId$, $logicalVersion_{mvo}$ $(mvoid, dbvid) = lvoid_{mvoid,dbvid}$ where $lvoid_{mvoid,dbvid}$ = $(mvoid,$ $dbvid)$.

Example 4.2 The invocation of $logicalVersion_{mvo}$ on multiversion object $circle_1$ in database version DBV_3, identified by $dbvid_3$, returns the logical version equal to the ($mvoid_1$,$dvid_3$) pair: $logicalVersion_{mvo}$ $(mvoid_1, dbvid_3) = lvoid_{mvoid_1,dbvid_3}$ with $lvoid_{mvoid_1,dbvid_3} = (mvoid_1, dbvid_3)$. □

Physical version of a *lvObject*. The *physical version* of a $lvObject$ has a unique identifier, denoted as *oid*, and a *value*. $lvObject$ has a unique physical version but this version may be shared by different logical versions of the same $mvObject$. Now, we define three functions on a $lvObject$ and its physical version:

The *physicalVersion*$_{lvo}$ function from $\mathcal{M}vOid \times \mathcal{D}bvId$ to $\mathcal{O}id$ maps each $lvObject$ to a unique physical version.:
$\forall$ $lvoid_{mvoid,dbvid} \in \mathcal{M}vOid \times \mathcal{D}bvId$, $physicalVersion_{lvo}$ $(lvoid_{mvoid,dbvid}) = oid$ where $oid \in \mathcal{O}id$ and $\forall$ $oid_j \in \mathcal{M}vOid$, if $physicalVersion_{lvo}$ $(lvoid_{mvoid,dbvid}) = oid_j$ then $oid = oid_j$.

The $\boldsymbol{value_\phi}$ function from $\mathcal{O}id$ to $\mathcal{V}alue$ maps each physical version of a $lvObject$, identified by oid, to a value of correct domain:
$\forall\ oid \in \mathcal{O}id,\ value_\phi(oid) = v$ where $\exists\ v \in \mathcal{V}alue$ with $v \in representation(Class(oid))$
and $\forall\ v_j \in \mathcal{V}alue$, if $value_\phi(oid) = v_j$ then $v = v_j$.

The $\boldsymbol{value_{lvo}}$ function from $\mathcal{M}vOid \times \mathcal{D}bvId$ to $\mathcal{V}alue$ computes the mapping of a given logical version of a multiversion object using its physical version; $\forall\ lvoid_{mvoid,dbvid} \in \mathcal{M}vOid \times \mathcal{D}bvId$,
$value_{lvo}\ (lvoid_{mvoid,dbvid}) = value_\phi(physicalVersion_{lvo}(lvoid_{mvoid,dbvid}))$.

Example 4.3 The invocation of $physicalVersion_{lvo}$ on the logical version of $circle_1$, in database version dbv_3, identified by $lvoid_{mvoid_1,dbvid_3} = (mvoid_1,\ dbvid_3)$, maps this logical version of $circle_1$ to its physical version, identified by oid_{13}, as shown in Figure 5. Moreover, the invocation of $value_\phi$ on this physical version (oid_{13}) maps the white circle identified by oid_{13} to its value. Finally, the invocation $value_{lvo}$ on the logical version of $circle_1$ in the database version dbv_3, identified by $lvoid_{mvoid_1,dbvid_3}$ returns also the same value:

$physicalVersion_{lvo}(lvoid_{mvoid_1,dbvid_3}) = oid_{13}$;
$value_\phi(oid_{13}) = valOid_{13}$;
$value_{lvo}(lvoid_{mvoid_1,dbvid_3}) = valOid_{13}$. □

Multiversion class. A *multiversion* class, $mvClass$ for short, has an identifier called $mvcid$ and a set of *logical* versions that reflect the same kind of entities of the real world [CJK91]. We note the set of multiversion classes as $\mathcal{M}ultiversionClass$, and the set of $mvcid$ as $\mathcal{M}vCid$.

The $\boldsymbol{ident_{mvc}}$ function from $\mathcal{M}ultiversionClass$ to $\mathcal{M}vCid$ returns the identifier of a *multiversion* class. This function is used to the mapping between the set of $\mathcal{M}ultiversionClass$, i.e., the user interface, and $\mathcal{M}vCid$, i.e., the system interface:
$\forall\ mvclass \in \mathcal{M}ultiversionClass,\ ident_{mvc}\ (mvclass) = mvcid$ where $mvcid \in \mathcal{M}vCid$.

The $\boldsymbol{assignment_{mvc}}$ function from $\mathcal{M}vCid$ to $\mathcal{M}vOid$ assigns an identifier to a multiversion object:
$\forall\ mvcid \in \mathcal{M}vCid,\ assignment_{mvc}(\ mvcid) = mvoid$ where $mvoid \in \mathcal{M}vOid$
with $\mathcal{M}vOid = \cup\ \{assignment_{mvc}(mvcid) \mid mvcid \in \mathcal{M}vCid\ \}$

Example 4.4 In Figure 5, class $Circle$, identified by $mvcid_2$, is a multiversion class with three logical versions: ($mvcid_2$, DBV_1), ($mvcid_2$, DBV_2), and ($mvcid_2$, DBV_3). Moreover, the invocation of $ident_{mvc}$ on the $Circle$ class returns its identifier, i.e., $mvcid_2$. Finally, the invocation $assignment_{mvc}$ on the identifier of the class $Circle$ assigns the multiversion object identifier of the $circle_1$, i.e., $mvoid_1$:

$ident_{mvc}(Circle) = mvcid_2$;
$assignment_{mvc}(mvcid_1) = mvoid_1$. □

Logical version of a *mvClass*. The *logical version* of a $mvClass$, $lvClass$ for short, represents its class version in a specific database version. This *logical* version is identified by a unique pair ($mvcid$, $dbvid$), denoted $lvcid_{mvcid,dbvid}$, where $dbvid$ is the database version identifier. We define five functions on a $lvClass$: the first one computes a logical version; next two are the basic functions that return the database version in which exists the logical version of the multiversion class, and the identifier of the multiversion class respectively; the fourth one, computes a logical version of a *lvObject* using a *lvClass*; The fifth one implies the creation of a new multiversion object, and then computes its logical version using the database version of the *lvClass*.

The *logicalVersion*$_{mvc}$ function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{M}vCid \times \mathcal{D}bvId$ returns the logical version of a *mvClass* in a specific database version:
$\forall\ mvcid \in \mathcal{M}vCid$ and $\forall\ dbvid \in \mathcal{D}BVId$, $logicalVersion_{mvc}\ (mvcid, dbvid) = lvcid_{mvcid,dbvid}$ where $lvcid_{mvcid,dbvid} = (mvcid, dbvid)$.

The *dbV*$_{lvc}$ function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{D}bvId$ returns the database version of the *lvClass*:
$\forall\ lvcid_{mvcid,dbvid} \in MvCid \times \mathcal{D}bvId,\ dbV_{lvc}(lvcid_{mvcid,dbvid}) = dbvid.$

The *mvClass*$_{lvc}$ function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{D}bvId$ returns the identifier of the multiversion class of a *lvClass*:
$\forall\ lvcid_{mvcid,dbvid} \in MvCid \times \mathcal{M}vCid,\ mvClass_{lvc}(lvcid_{mvcid,dbvid}) = mvcid.$

The *logicalAssignment* function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{M}vOid \times \mathcal{D}bvId$ infers an identifier of a logical version of a *mvObject* using a multiversion object identifier and the database version of the *LvClass*:
$\forall\ lvcid_{mvcid,dbvid} \in \mathcal{M}vCid \times \mathcal{D}bvId,\ logicalAssignment(\ lvcid_{mvcid,dbvid}) =$
$logicalVersion_{mvo}(mvoid,\ dbV_l(lvcid_{mvcid,dbvid}))$ where $mvoid \in \{\ assignment_{mvc}(mvcid)\ \}$.

The *assignment*$_{lvc}$ function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{M}vOid \times \mathcal{D}bvId$, computes an object identifier of a *lvObject* using the assignment function on *mvClass* and the database version of the logical class:

$$\forall\ lvcid_{mvcid,dbvid} \in \mathcal{L}vCid,\ assignment_{lvc}(lvcid_{mvcid,dbvid}) = lvoid_{mvcid,dbvid} \text{ where } lvoid_{mvcid,dbvid} = logicalVersion_{mvo}(\ (assignment_{mvc}(\ mvClass_l(lvcid_{mvcid,dbvid}))),\ dbV_{lvc}(lvcid_{mvcid,dbvid})).$$

Example 4.5 The invocation of $logicalVersion_{mvc}$ on the multiversion *Circle* class in the database version DBV_3, identified by $dbvid_3$, returns the logical version $lvcid_{mvcid_2,dbvid_3}$ identified by the pair $(mvcid_2, dbvid_3)$. Then, the invocation of *logicalAssignment* on this logical version, identified by $lvcid_{mvcid_2,dbvid_3}$ computes the identifier of the multiversion $circle_1$ object in the database version DBV_3. Finally, the identification of a logical version of a new circle created in the database version DBV_3 processes in two steps: the first step assigns a new identifier - $mvoid_3$ - to the new multiversion object by the invocation of $assignment_{mvc}$ on the multiversion class *Circle*; the second step computes the identifier of the multiversion object in the database version DBV_3 with the invocation of $logicalVersion_{mvo}$ on the new identifer $mvoid_3$ and the identifier of DBV_3:

$logicalVersion_{mvc}\ (mvcid_2,\ dbvid_3) = lvcid_{mvcid_2,dbvid_3}$ with $lvcid_{mvcid_2,dbvid_3} = (mvcid_2, dbvid_3)$;
$logicalAssignment(\ lvcid_{mvcid_2,dbvid_3}) = lvoid_{mvoid_1,dbvid_3}$ with $lvoid_{mvoid_1,dbvid_3} = (mvoid_1, dbvid_3)$;
$assignment_{lvc}(lvcid_{mvcid_1,dbvid_3}) = lvoid_{mvoid_3,dbvid_3}$, with $lvoid_{mvoid_3,dbvid_3} = (mvoid_3, dbvid_3)$.

□

Physical version of a *lvclass*. The *physical version* of a *lvClass*, *pvClass* for short, is a class which has a unique identifier, a type as representation, and a set of methods as behavioral definition.

This class is managed by the system which assigns to it a name and an identifier. This physical version groups physical versions of $lvObjects$ having the same type and the same behavior. We note the set of physical classes as $\mathcal{P}vClass$, and the set of physical class identifier as $\mathcal{P}vCid$. A $lvClass$ has a unique physical version but this version may be shared by different logical versions of the same $mvClass$. Below, we define three functions on a $lvClass$ and its physical version:

The $\boldsymbol{ident_\phi}$ function from $\mathcal{P}vClass$ to $\mathcal{P}vCid$ returns the identifier of a physical class. This function is applied to the mapping between the set of $\mathcal{P}vClass$ and the set of $\mathcal{P}vCid$:
$\forall\ pclass \in \mathcal{P}vClass$, $ident_\phi(pvclass) = pvcid$ where $pvcid \in \mathcal{P}vCid$.

The $\boldsymbol{physicalVersion_{lvc}}$ function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{P}vCid$ maps a $lvClass$ to its physical version:
$\forall\ lvcid_{mvcid,dbvid} = (mvcid,\ dbvid) \in \mathcal{M}vCid \times \mathcal{D}bvId$,
$physicalVersion_{lvc}\ (lvcid_{mvcid,dbvid}) = pvcid$ where $pvcid \in \mathcal{P}vCid$
and $\forall\ pvcid_j \in \mathcal{P}vCid$, if $physicalVersion_{lvc}\ (lvcid) = pvcid_j$ then $pvcid = pvcid_j$.

The $\boldsymbol{assignment_\phi}$ function from $\mathcal{P}vCid$ to $\mathcal{O}id$ assigns an object identifier to a physical version of a $lvObject$:
$\forall\ pvcid \in \mathcal{P}vCid$, $assignment_\phi(pvcid) = oid$ where $oid \in \mathcal{O}id$
with $\mathcal{O}id = \cup\ \{\ assignment_\phi(pvcid) \mid pvcid \in \mathcal{P}vCid\ \}$.

Example 4.6 The invocation of $physicalVersion_{lvc}$ on the logical version of the multiversion $Circle$ class in database version DBV_3, maps this logical version to its physical version, identified by $pvcid_5$. Moreover, the invocation of $assignment_\phi$ on this physical version assigns a new identifier (oid_{13}), associated with the physical version of the multiversion object $circle_1$:

$physicalVersion_{lvc}(lvcid_{mvcid_2,dbvid_3}) = pvcid_5$;
$assignment_\phi(pvcid_5) = oid_{13}$. □

4.2 Well-formed logical class hierarchy

Inheritance Relationship. An *inheritance relationship* is a partial order on the logical versions of multiversion classes in the same database version. Given two logical versions $lvcid_{mvcid_1,dbvid}$ and $lvcid_{mvcid_2,dbvid} \in \mathcal{M}vCid \times \mathcal{D}bvId$, if $lvcid_{mvcid_1,dbvid} \prec_{dbvid} lvcid_{mvcid_2,dbvid}$
then $lvcid_{mvcid_2,dbvid}$ is a $superclass_{dbvid}$ of $lvcid_{mvcid_1,dbvid}$
and $lvcid_{mvcid_1,dbvid}$ is a $subclass_{dbvid}$ of $lvcid_{mvcid_2,dbvid}$.

Function $superclass_{dbvid}$ from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{M}vCid \times \mathcal{D}bvId$ computes all the direct superclasses of the logical version of a multiversion class in the same database version using: (1) its physical version, and (2) the function $superclass_\phi$ defined on the physical version of a $lvClass$:
$\forall\ lvcid_{mvcid,dbvid} \in \mathcal{M}vCid \times \mathcal{D}bvId$,
$superclass_{dbvid}(lvcid_{mvcid,dbvid}) = \cup_{mvcid_i \in Super}\ logicalVersion_{mvc}(mvcid_i,\ dbV(lvcid_{mvcid,dbvid}))$
where $Super = superclass_\phi(physicalVersion_c(lvcid_{mvcid,dbvid}))$
and the $superclass_\phi$ function from $\mathcal{P}vCid$ to $\mathcal{M}vCid$ is defined as :
$\forall\ pvcid \in \mathcal{P}vCid\ superclass_\phi(pvcid) = \{mvcid \mid mvcid \in \mathcal{M}vCid\}$.

Example 4.7 The invocation of $superclass_{dbvid}$ on the logical version of the multiversion class *Ellipse* ($mvcid_3$), in the database version DBV_3, is performed in two following steps. The first one, determines the set of multiversion classes whose logical versions are the direct $superclass_{dbvid_3}$ of the logical version of *Ellipse*, identified by $lvcid_{mvcid_3,dbvid_3}$,i.e, the set composed of the multiversion class $Circle$. The second step computes the logical version of this multiversion class in the database version $dbvid_3$, using the $logicalVersion_{mvc}$ invocation:

$superclass_{dbvid_3}(lvcid_{mvcid_3,dbvid_3}) = \cup_{mvcid_i \in Super}\ logicalVersion_{mvc}(mvcid_i, dbV(lvcid_{mvcid_3,dbvid_3}))$
after inference : $superclass_{dbvid_3}(lvcid_{mvcid_3,dbvid_3}) = \{lvcid_{mvcid_2,dbvid_3}\ \}$
and the final result is $\{lvcid_{mvcid_2,dbvid_3}\ \}$. $\Box$

Well-formed class hierarchy. An inheritance hierarchy on *logical versions* of multiversion classes in a database version, identified by $dbvid$, is a 3-tuple:

$$< \mathcal{L}vClass, representation_{lvc}, \prec_{dbvid}> \text{ where}$$

- $\mathcal{L}vClass$ is the set of logical versions of multiversion classes;
- $representation_{lvc}$ is a logical mapping function from $\mathcal{L}vClass$ to $\mathcal{S}tructuralType$ computed using the representation on its physical version, since only its physical version has a structural type description:
  $\forall\ lvcid_{mvcid,dbvid} \in \mathcal{L}vClass$,
  $representation_{lvc}(lvcid_{mvcid,dbvid}) = representation_{\phi}(physicalVersion_{lvc}(lvcid_{mvcid,dbvid}))$
  where $representation_{\phi}$ is a mapping function from $\mathcal{P}vCid$ to $\mathcal{S}tructuralType$ defined as:
  $\forall\ pvcid \in \mathcal{P}vCid,\ representation_{\phi}(pvcid) = \tau$ where $\tau \in \mathcal{S}tructuralType$;
- $\prec_{dbvid}$ is an inheritance relationship on logical versions of multiversion classes in the database version identified by $dbvid$.

A logical class hierarchy is *well-formed* when the following assumption is true:
let $lvcid_{mvcid_a,dbvid}, lvcid_{mvcid_b,dbvid} \in LvClass$; $lvcid_{mvcid_a,dbvid} \prec_{dbvid} lvcid_{mvcid_b,dbvid} \Rightarrow$
$representation_{lvc}(lvcid_{mvcid_a,dbvid}) \preceq representation_{lvc}(lvcid_{mvcid_b,dbvid})$, i.e.,
if $lvcid_{mvcid_a,dbvid}$ is a subclass of $lvcid_{mvcid_b,dbvid}$ then the type of the physical version of the $subclass_{dbvid}$ has to be the refinement of the type of the physical version of its $superclass_{dbvid}$.

4.3 Root of Persistence

There are two main assumptions of the persistence policy: a root of persistence is global, i.e., known by all the database versions, and a root is not directly versioned, i.e., its domain may be versioned but a root of persistence can not change its definition in a database version.

Root of persistence. A Root of persistence attributes persistence to objects: an object explicitly attached to a root in a database version becomes persistent, and it is managed by the versionning system. Persistence is also attributed to an object if this object is referenced by another persistent object. Moreover, a root may be an instance of a class or a type.

The function $instanceOf_{dvbid}$ from $\mathcal{R}oot \times \mathcal{DBV}id$ to $\mathcal{D}omain \times \mathcal{DBV}id$ gives the domain of a root in a database version identified by $dbvid$:
$\forall\ r \in \mathcal{R}oot \times \mathcal{DBV}id,\ instanceOf_{dbvid}(r, dbvid) = logicalVersion_{\tau}(\tau, dbvid)$ where $\tau \in \mathcal{D}omain$.

The function $logicalVersion_\tau$ from $\mathcal{D}omain \times \mathcal{DBV}id$ returns the logical version of a domain in a database version:

τ is an atomic type. $logicalVersion_\tau(\tau, dbvid) = logicalVersion_{atomicType}(\tau, dbvid)$ with $logicalVersion_{atomicType}(\tau, dbvid) = \tau$;

τ is a tuple. $\tau = [\ldots, a_i\colon \tau_i, \ldots]$, $logicalVersion_\tau(\tau, dbvid) = logicalVersion_{tuple}(\tau, dbvid)$ with $logicalVersion_{tuple}(\tau, dbvid) = [\ldots, a_i{:}logicalVersion_\tau(\tau_i, dbvid), \ldots]$;

τ is a set. $\tau = \{\tau_i\}$, $logicalVersion_\tau(\tau, dbvid) = logicalVersion_{set}(\tau, dbvid)$ with $logicalVersion_{set}(\tau, dbvid) = \{\ logicalVersion_\tau(\tau_j, dbvid)\}$;

τ is a list. $\tau = <\tau_i>$, $logicalVersion_\tau(\tau, dbvid) = logicalVersion_{list}(\tau, dbvid)$ with $logicalVersion_{list}(\tau, dbvid) = <\ logicalVersion_\tau(\tau_j)\ >$;

τ is a $mvClass$**.** $\tau\ in\ \mathcal{M}vCid$, $logicalVersion_\tau(\tau, dbvid) = logicalVersion_{mvc}(\tau, dbvid)$.

Example 4.8 The circle drawn by Joe, named JoeCircle, is an instance of the *Circle* class. The definition of this root is made in the database version DBV_1 but only the link between the root and the multiversion *Circle* class is stored by the system. The domain of this root in the database version DBV_3 is inferred by the system using the $logicalVersion_\tau$ invocation:
$instanceOf_{dvbid}(JoeCircle, dbvid_3) = Lvcid_{mvcid_1, dbvid_3}$.

The logo of Olympic games is a set of instances of the multiversion *Circle* class. Its definition is made in the database version DBV_3. The invocation of $instanceOf_{dvbid}$ on this root infers its logical version on the DBV_3:
$instanceOf_{dvbid}(OlympicGames, dbvid_3) = \{lvcid_{mvcid_5, dbvid_3}\}$;

The declarations of these roots in the database version DBV_3 are:
name JoeCircle = Circle;
name OlympicGames = set(Circle); □

The $extent_{dvbid}$ function from $\mathcal{R}oot \times \mathcal{DBV}id$ to $\mathcal{V}alueMvObject = \mathcal{V}alue \cup \mathcal{M}vObject \times \mathcal{DBV}id$ associates with each root of domain τ a logical version of a value or a $mvObject$ in a database version identified by $dbvid$:
$\forall\ r \in \mathcal{R}oot \times \mathcal{DBV}id$, $extent_{dbvid}(r, dbvid) = logicalVersion_{val\_mvo}(x, dbvid)$ where $x \in \mathcal{V}alueMvClass$;

The $logicalVersion_{val\_mvo}$ function from $\mathcal{V}alueMvObject = \mathcal{V}alue \cup \mathcal{M}vObject \times \mathcal{DBV}id$ infers the logical version of a value or a $mvClass$ in a database version:

x is an atomic value. $logicalVersion_{val\_mvo}(x, dbvid) = logicalVersion_{atomicValue}(x, dbvid)$ with $logicalVersion_{atomicValue}(x, dbvid) = x$;

x is a tuple value. $x = [\ldots, a_i\colon x_i, \ldots]$, $logicalVersion_{val\_mvo}(x, dbvid) = logicalVersion_{tupleValue}(x, dbvid)$ with $logicalVersion_{tupleValue}(x, dbvid) = [\ldots, a_i{:}logicalVersion_{val\_mvo}(x_i, dbvid), \ldots]$;

x is a set value. $x = \{x_i\}$, $logicalVersion_{val\_mvo}(x, dbvid) = logicalVersion_{setValue}(x, dbvid)$ with $logicalVersion_{setValue}(x, dbvid) = \{\ logicalVersion_{val\_mvo}(x_j, dbvid)\}$;

x **is a list value.** $x = < x_i >$, $logicalVersion_{val\_mvo}(x, dbvid) = logicalVersion_{listValue}(x, dbvid)$ with $logicalVersion_{listValue}(x, dbvid) = < logicalVersion_{val\_mvo}(x_j) >$;

x **is a** $mvObject$**.** x in $\mathcal{M}vCid$, $logicalVersion_{valmvo}(x, dbvid) = logicalVersion_{mvo}(x, dbvid)$.

Example 4.9 The object attached to the circle drawn by Joe, named $JoeCircle$, in the database version dbV_3 is the logical version of $circle_1$ in this databse version, i.e., $lvoid_{mvoid_1, dbvid_3}$. The objects attached to root $OlympicGames$ constitute a set of logical versions of $circle$ instance in the database version DBV_3:

$extent_{dvbid}(JoeCircle, dbvid_3) = Lvoid_{mvoid_1, dbvid_3}$;
$extent_{dvbid}(OlympicGames, dbvid_3) = \{lvcid_{mvoid_10, dbvid_3}, lvcid_{mvoid_11, dbvid_3}, lvcid_{mvoid_12, dbvid_3}, lvcid_{mvoid_13, dbvid_3}, lvcid_{mvoid_13, dbvid_3}, \}$; $\square$

4.4 Definition of a logical Version of a Schema

The *logical version of a schema* of a database version is a 3-tuple:

$$SV =< \mathcal{L}vClass, representation_{lvc}, \prec_{dbvid}, \mathcal{R}oot > \text{ where:}$$

- $\mathcal{L}vClass$ is the set of logical versions of multiversion classes;
- $representation_{lvc}$ is a logical mapping function from $\mathcal{L}vClass$ to $\mathcal{S}tructuralType$;
- $\prec_{dbvid}$ is a partial order on $\mathcal{L}vClass$ in a database version $Dbvid$;
- $< \mathcal{L}vClass, representation_{lvc}, \prec_{dbvid}>$ is a well-formed class hierarchy in the database version;
- $\mathcal{R}oot$ is the set of roots of persistence;

4.5 Definition of a logical Version of a Schema instance

The *logical version of a schema instance* of a database version is a 3-tuple:

$$IV =< logicalAssignment, Assignment_{lvc}, value_{lvo}, extent_{dbvid} > \text{ where}$$

- $LogicalAssignment$ is a function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{M}vOid \times \mathcal{D}bvId$ which computes an identifier of a logical version of a $mvObject$;
- $Assignment_{lvc}$ is a function from $\mathcal{M}vCid \times \mathcal{D}bvId$ to $\mathcal{M}vOid \times \mathcal{D}bvId$ which computes an object identifier of a $lvObject$, using the assignment function on $mvClass$ and the database version of the logical class;
- $value_{lvo}$ is a function from $\mathcal{M}vOid \times \mathcal{D}bvId$ to $\mathcal{V}alue$ which computes the mapping of a given logical version of a multiversion object using its physical version;
- $extent_{dbvid}$ is a function from $\mathcal{R}oot \times \mathcal{D}BVid$ to ValueMvClass = $\mathcal{V}alue \cup \mathcal{M}vClass \times \mathcal{D}BVid$ which associates with each root of domain τ a logical version of a value or a $mvObject$ in a database version.

Figure 6 resumes the relevant functions proposed in this section. When a function is inferred from another function, it is drawn as a dashed line. Entities at the logical level (seen by users) are drawn as gray ovals, whereas entities at the physical level are drawn as white ovals.

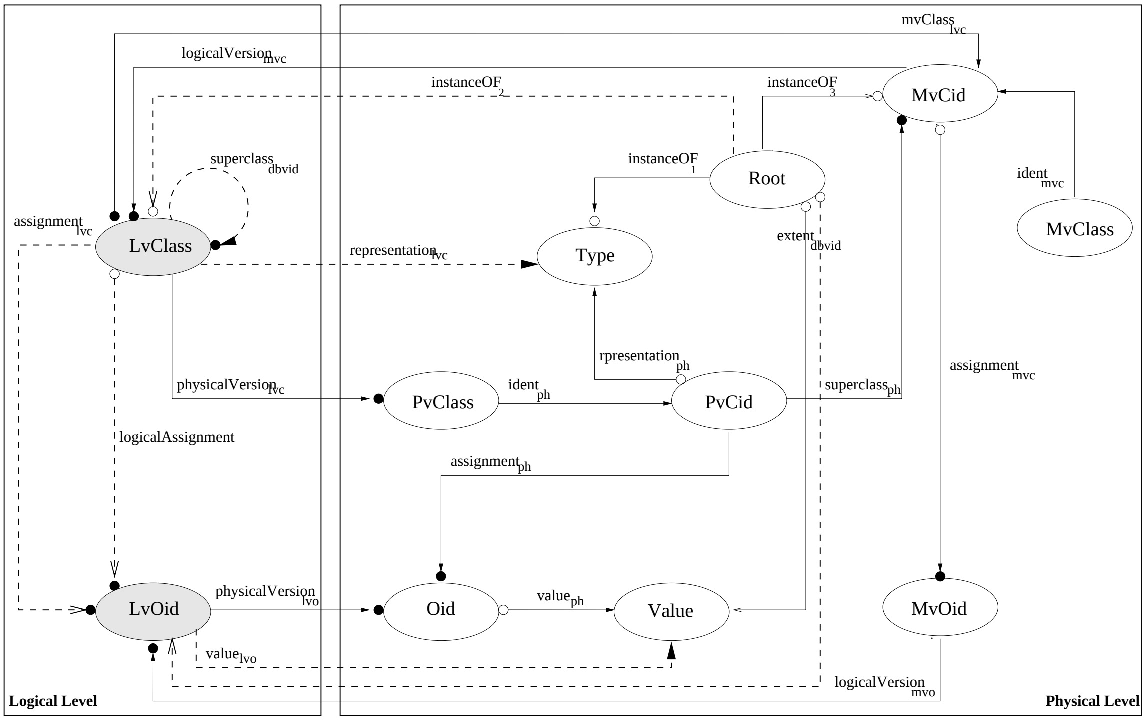


Figure 6: *The graphical representation of a database version*

5 Conclusion

In this paper, we present a formal model of multiversion databases considering only the structural aspects: type, inheritance relationship, and root of persistence. We base on the database version approach. We show how the notion of the database version, as a unit of consistency allows to manage both schema versions and versions of schema instances. By this notion, appropriate object and class versions are bound together to represents consecutive states and state variants of the modeled universe, that may vary by their value, type, and behavior. We are not aware of any other work in the fields of object-oriented systems with versions that identify an appropriate class version from an object version. In the model proposed in this paper, it is identified immediately, because switching concerns the entire database version comprising all the required object and class versions, instead of a version of an object.

In the formal model, we dissociate three kinds of an object and a class: *multiversion*, *logical version* and *physical version*. We define a logical version of schema and a version of the schema instance in term of a set of entities like set of logical versions of classes and roots of persistence, and functions defined on these sets. These functions constitute the kernel of the database version approach. They show when and how the versioning mechanism is considered. The main advantage of this model is its extensibility and adaptability to take into account other object models. Moreover, this model offers a rigorous framework for the development of new multiversion functionalities such that evolution support, view mechanism, query process and integrity constraints.

The next step in this research will concern methods formalization, and implementation of the extension to the database version mechanism, in order to support the version of schema with inheritance relationship and methods. A version manager according to the model described in this paper has been implemented in the Franco-Polish School of New Information and Communication Technologies of Poznan in collaboration with O2 Technology, where object-oriented database O_2 is developed.

References

[ABDE96] E. Amiel, M.J. Bellosta, E. Dujardin, and E.Simon. Type-Safe Relaxing of schema Consistency Rules for Flexible Modelling in OODBMS. *Vldb Journal- special issue*, 1996.

[ADL91] R. Agrawal, L.G. DeMichiel, and B. G. Lindsay. Static type checking of multimethods. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1991.

[AHPZ95] E. Andonoff, G. Hubert, A. Le Parc, and G. Zurfluh. Modelling inheritance, composition and relationship links between objects, object versions and class versions. In *CAiSE/International Conference on Advanced Information System Engineering*, 1995.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundation of Databases.* Adison Wesley Publishing Company, 1995.

[AJFL] L. Al-Jadir, G. Falquet, and M. Leonard. Context Versions in an Object-Oriented Model. In *Unknown yet.*

[ALP91] J. Andany, M. Léonard, and C. Palisser. Management of Schema Evolution in Databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 161–170, 1991.

[AS95] S. Abiteboul and C.S. Santos. IQL(2): A model with Ubiquitous Objects. In *Fifth International Workshop on Database Programming Languages*, 1995.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 311–322, 1987.

[Car84] L. Cardelli. A semantic of multiple inheritance. In *Proc. of the Symposium on Semantics of Data Types*, 1984.

[CGJ94] W. Cellary, Vossen G, and G. Jomier. Multiversion Object Constellation; a new Approach to Support a Designer's Database Work. *Engineering with Computers*, 10:230–244, 1994.

[CJ90] W. Cellary and G. Jomier. Consistency of Versions in Object-Oriented Databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 447–462, 1990.

[CJK91] W. Cellary, J. Jomier, and T. Koszlajda. Formal Model of an Object-Oriented Database with Versioned Objects and Schema. In *Proc. of the Intl. Conf. on Data and Expert Systems Applications (DEXA)*, pages 239–244, 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[Dan90] S. Danforth. Multi-targeted virtual functions for OODB. In *Bases de Données Avancées*, 1990.

[Deu91] 0. Deux. The O2 system. *Communications of the ACM*, 34(10):34–48, 1991.

[Kat90] R.H. Katz. Toward a Unified Framework for Versioning Modeling in Engineering Databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–386, 1990.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice-Hall, 1994.

[Laj93] J. Lajoie. The new language extensons. Technical Report 5(6), C++ report, 1993.

[LH90] B. Lerner and A. Habermann. Beyond Schema Evolution to Database Reorganisation. In *SIGPLAN Notices, 25(10)*, pages 67–76, 1990.

[McK92] R. McKenzie. *An Algebraic Model of Class, Inheritance, and Message Passing.* PhD thesis, Computer Science Dept, University of Texas at Austin, 1992.

[MS93] S. Monk and I. Sommerville. Schema Evolution in OODBs Using Class Versioning. In *SIGMOD RECORD, Vol. 22, No. 3*, pages 16–22, 1993.

[RC94] editor R.G.G Cattell. *The Object Database Standard: ODMG-93.* Morgan Kauffman, 1994.

[SG94] G. Jomier S. Gancarsky. Managing Entity Version within their Context : a Formal Approach. In *Proc. of the Intl. Conf. on Data and Expert Systems Applications (DEXA)*, pages 400–409, 1994.

[SZ86] A.H. Skarra and S.B. Zdonik. The Managemen of Changing Types in an Object-Oriented Database. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 483–495, 1986.

[TK89] L. Tan and T. Katayama. Meta Operations for Type Management in Object-Oriented Databases - A Lazy Mechanism for Schema Evolution. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 58–75, 1989.

[TOC93] G. Talens, C. Oussalah, and M.F. Colinas. Versions of Simple and Composite Objects. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 62–72, 1993.