Versions of Simple and Composite Objects

G. Talens and C. Oussalah
E.E.R.I.E / L.E.R.I
Parc Scientifique Georges Besse
30000 Nîmes
Phone : (33) 66 38 70 00
Fax : (33) 66 84 05 06
E-mail : oussalah@eerie.eerie.fr

M.F. Colinas
C.N.E.T.
38-40 rue général Leclerc
92131 Issy les Moulineaux
Tel : 45 29 45 51
Fax : 45 29 60 69
Mail : marie-francoise.colinas@issy.cnet.fr

Abstract

In this paper, we propose a model of versions which manages simple and composite objects.
In our model, we have identified two types of versions :
- class versions [10] [24]
- and instance versions [6] [8].
These different versions are linked to each other by different relationships.
A mechanism for the automatic propagation of versions [13] [24] for composite objects is proposed in order to propagate the creation of versions only in certain cases.

1 Introduction

In design, the objects have properties which are from time-dependent or otherwise parameterized data. In most cases, only the objects containing the most recent information are used but we can use the "old" information. One solution which is advocated is the use of versions. Generally, several versions of a same object must be kept. The versioning of objects helps not only to keep track of the evolution of the objects to be designed but also to store the data corresponding to a context.

Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993.

However, a crucial question comes to mind concerning the definition of a version. Does the modification of a property of an object always imply the creation of a new version of the object or are there some properties which can be updated within a version without a new version being created ? We often say that versioning is generally associated with a change in the state of an object but firstly, we have to define what the state [5] [8] [12] of an object is.
Moreover, the objects often have complex structures [10] [11], that is to say, they are composed of other objects. Whenever a version of one of the component objects is created, must this creation be propagated [13] [24] on the composite object ?
There are therefore many problems inherent in version management, hence the importance of having a model of versions which allows a rich semantics based on an object structure to be used.
In this paper, we propose an object version model the characteristics of which are discussed in the following points :
- we have identified two types of versions related to two categories of users :
  * class versions, in order to take into account the evolution of the classes, i.e. the properties and operations that a class contains can be modified or deleted or new properties can be added.
  * instance versions, in order to take into account the modifications of the properties inside the instances.
- the successive versions (class or instance versions) of a versionable class are linked in order to follow more easily the modifications made to one version when compared with its previous version. The history of the development is easier to follow, the information redundancy between successive versions is avoided and also the storage of versions is facilitated (delta method [16] [22] or binary coded table [9]).

- the composite versions are also taken into account and therefore, the propagation of versions is managed. In order to avoid the creation of infinite versions, we have identified two concepts which involve the propagation of versions :

* sensitive version : propagation is performed only if the user has designated this version as sensitive for propagation.

* sensitive composite attribute : if a component version has a "permanent" state and if the composite attribute which links it to the composite version is sensitive, the propagation of versions takes place.

This model is generic. It is more particularly adapted for hierarchical/multi-view modeling [14] which is used as the basis for our modeling. This model has been implemented in the Presage system [3] [17] which is a tool for network planning[1].

Section 2 of this paper presents the concepts of simple object versions. In section 3 the representation of composite object versions is explained. Section 4 describes the implementation of these versions in an object-oriented approach. In section 5, we give an example of an application, in order to illustrate the different concepts seen previously.

2 Representation of simple object versions

2.1 Taxonomy of versions

Our model of versions is based on an object-oriented approach[2], so we use the concept of class and instance.

Two types of users can use our model :

- the application builder : who is a domain specialist who supplies the specific knowledge required to accomplish a certain class of applications. His role is to define appropriate models for the class of applications, specify appropriate processing tools and establish problem resolution strategies adapted to his class of applications.
- the final user : he instantiates the model of applications built by the application builder in order to create his own application. His role is to give the initial data for a given problem and to use the tools necessary for the execution of this problem.

In our model, the class can evolve, i.e the properties and operations that it contains can be modified or deleted and new properties can be added. In order to take into account this evolution without questioning the existence of the class previously defined, we need class versions which will be used by the application builders. We also need instance versions so that we can take into account the evolution of instances, i.e the modification of properties contained in the instances. The instance versions will be used by the final users.

Therefore, two types of versions are necessary :

- versions of classes
- versions of instances.

The following models have versions of classes and versions of instances : Orion [5] [10], Encore [19] [24], Avance [2], Iris[1] and Charly [16]. Most of the other models have only versions of instances.

A class can be defined as being versionable or not. Defining a class as being versionable causes the creation of the class "generic class version". The class "generic class version" allows the version history of the versionable class to be managed and the current version of the versionable class to be known. The relationship between the versionable class and the class "generic class version" is the "is-version-of" relationship. The versions defined from the versionable class are linked to each other by the "inter-version" relationships which we will describe in paragraph 2.3. These different class versions are linked to the class "generic class version" by the "ISA" inheritance relationship.

The class "generic instance version" is created when the first instance version of the versionable class is created. "Generic instance version" allows the history of all the instance versions of the versionable class to be managed. The class "generic instance version" is linked to the versionable class by the "is-version-of" relationship.

We have distinguished the "generic-class-version" and the "generic-instance-version" in order to :

- separate the operations which can be used by the application builder (operations available for the class versions) and those used by the final user (operations available for the instance versions).
- facilitate access to the set of class versions or instance versions of a versionable class, because in the version history of "generic-class-version", we find all the class versions of the versionable class and in that of "generic-instance-version", we find all the instance versions of the versionable class.

Each instance version of the versionable class will be linked to the "generic instance version" by the "ISA" inheritance relationship. Each instance version is also linked to the versionable class or the class version from which it was created by the "ISA" inheritance relationship. This requires the use of a multiple inheritance model. The different instance versions are also linked to each other by "inter-version" relationships.

Our minimal model is composed of the versionable class and the "generic-class-version". With this model, it is

---

[1] Developed by the L.E.R.I. (Laboratoire d'Etude et Recherche en Informatique), C.N.E.T. (Centre National d'Etudes des Télécommunications) and ITECA (Informatique et TEChniques Avancées).

[2] We consider the reader to be familiar with the concepts and terminology of the object-oriented languages [20] [21].

possible to create instance versions from the versionable class.

The example in Figure 1 allows the concept of class versions and instance versions to be clarified. The class "class1" was built at the time t and this class became versionable at a time t+1. Making the class "class1" versionable involved the creation of the class "generic class version class1". This class is associated with "class1" by the "is-version-of" relationship. The class versions, ver0, ver1, ver2 and ver5 are derived from the class "class1". These different class versions are linked to each other by "inter-version" relationships. They are all linked to the class "generic class version class1" by the "ISA" inheritance relationship.
By creating the first instance version of the class "class1", we create the class "generic instance version class1". This class is linked to "class1" by the "is-version-of" relationship. This first instance version is created from the class version ver1. It is linked to the class "generic instance version class1" and to the class version ver1 by the "ISA" inheritance relationship. Bringing about the evolution of this instance version involves the creation of the instance version ver1.1 which is linked to the instance version ver1.0 by the "inter-version" relationship and to the "generic instance version class1" by the "ISA" inheritance relationship.

Note :
The instance "instance1" is not versionable because it was created before the class "class1" was versionable. When a class becomes versionable, all the existing instances are kept but they do not inherit versioned properties. These instances are therefore not versionable.

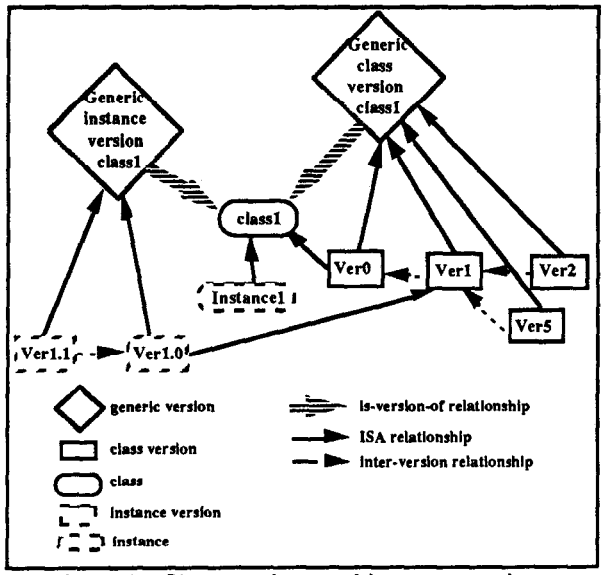


Figure 1 : Class versions and instance versions

2.2 States of versions

We must also take into account the state of the stability of the data contained in the versions [4] [5] [6] [8] because the versions which contain unstable data cannot be referenced. These have the status of temporary versions unlike permanent versions which contain stable data. In our model, we therefore define two states :
- permanent
- temporary.
These states are used either by the class versions or by the instance versions.

We use the term permanent version for a version which :
- is stable, therefore updating is forbidden,
- can be deleted.
A temporary version can be derived from a permanent version.
A temporary version can be promoted to a permanent version. This promotion can be explicit (made by the user) or implicit (made by the model). We distinguish two types of promotion:
- the promotion is explicit, that is to say, the user decides that he wants to transform a temporary version into a permanent version.
- the promotion is implicit, that is to say, the user wants to derive a version from a temporary version. In this case, the model transforms the temporary version into a permanent version and the user can then create its derivation.

We use the term temporary version for a version for which:
- updating is possible,
- deletion is possible.
A temporary version cannot be derived from a temporary version and a permanent version cannot be derived from a temporary version. The temporary versions are leaves.

When a version is created, its state is temporary by default. Its state can become permanent, either because the user has decided this (after verification he thinks that his version is stable therefore permanent) or because he wanted to derive a version from a temporary version (this transformation is performed by the model).
A permanent version can become a temporary version (transformation performed by the user) if this version does not have versions which are its derivatives and, in the case of a class version, if no instance version has been created from it.

The other models have :
* two states [6] [16] :
- in-progress (or working) which corresponds to an unstable version
- frozen (or released) which corresponds to a stable version
* three states [1] [12] :
- transient (deletion and updating are possible)

- working (only deletion is possible)
- released (deletion and updating are impossible)

* or four states [8] :
- in-progress
- effective
- released
- archived

In some models [24], the date is the criterion for the stability of the versions.

2.3 Version evolution

Class versions are derived from the versionable class or existing class versions. During the derivation of a class version, we do not copy the information from one class version to another because we have dynamic inheritance and not static inheritance.

The different versions derived from the versionable class are linked to each other by the "is-derived-from-with-*" relationship where * can represent different relationships :
- **except** (which represents the inheritance relationship and expresses the notion of exception concerning the properties) specifies the attributes to be removed from the new version.
- **plus** (which represents the inheritance relationship and expresses the notion of specialization concerning the properties) specifies the attributes to be added to the new version.
- **mod** (which represents the inheritance relationship and expresses the notion of masking concerning the properties) specifies the attributes the values of which will be modified in the new version.
- **refer** (which represents the inheritance relationship and expresses a notion of priority). This relationship allows the conflicts generated by the merging of versions to be avoided. When a version has several predecessor versions, this relationship allows a version to be designated, for which all the properties and values will be inherited in the case of conflict.

These different relationships allow the differences between two successive versions to be identified. Therefore, the users can follow the evolution of the different versions better.

In Figure 2, we take another look at the example in Figure 1. The class version ver1 is derived from the class version ver0. We suppose that the class version ver0 has the properties P1, P2 and P3. In the version ver1, we want to modify the property P2 of the version ver0 and for this, the "is-derived-from-with-mod" relationship is positioned. We also want to add the property P4 to the version ver1, using the "is-derived-from-with-plus" relationship. Therefore, ver1 has, in addition to P4, all the properties contained in ver0, with a new value for P2.

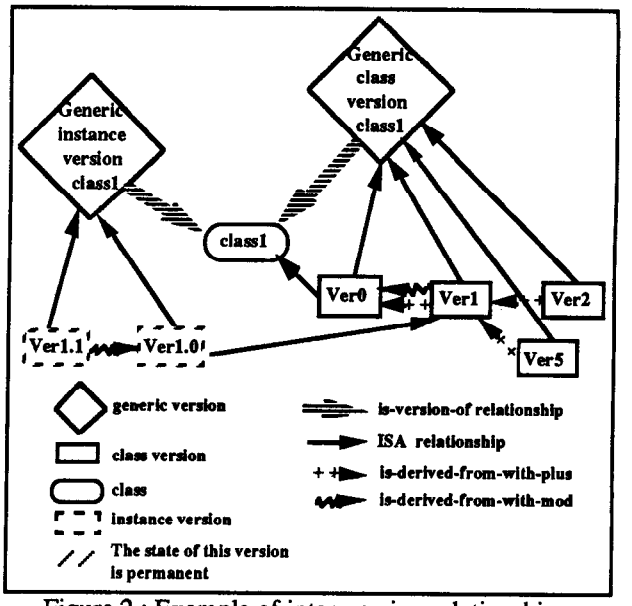


Figure 2 : Example of inter-version relationships

The inter-version relationships which exist between the instance versions are the "is-derived-from-with-mod" and "is-derived-from-with-refer" relationships, in order to respect the rules which state that an instance belongs to its class. These relationships are the same relationships as those defined previously between the class versions.

Different operations can be executed on these versions [15]:
- derivation,
- modification
- and deletion.

In most of the models, when a new version is created, it is related only to its predecessor by the predecessor/successor relationship [1] [13] [16] [24] or by the is-derived-from relationship [7] [12]. Each version is related to a version set object or a generic object by the member-of-version-set relationship [24] or by the version-of relationship [1] [16].

We will now describe how classes which have a complex structure are taken into account and managed in our model.

3 Representation of composite object versions

Most of the objects have complex structures, that is to say, they are composed of other objects which can themselves be composed of other objects. These objects constitute a hierarchy. They are composed of objects of the lowest level which may themselves be composed of component objects. The highest object in the hierarchy is called the composite object [11]. In a similar way, we have

composite versions (class versions or instance versions) which are composed of component versions (respectively class versions or instance versions). One of the main objectives is to set up relationships between the versions of objects of each node of this hierarchy.

The composite versions can be created in two ways [1] :
- explicitly
- or implicitly.

Explicit generation of versions means that the versions have been created by the user himself.
Implicit generation of versions means that these versions have been created because of modifications made to their component versions. The modifications made to component versions have brought about version propagation and consequently, the creation of composite versions.

3.1 Explicit creation

When the creation of a version is requested by a user, the creation of this version is explicit. The explicit creation of composite versions can take place in three ways :
- bottom-up,
- top-down
- or mixed.

Whatever the method of creation used (bottom-up, top-down or mixed), consistency between the composite versions and the component versions is verified.

In order to carry out bottom-up creation, we must first create the versions of the component objects of the lowest level in the hierarchy and step by step, we climb the composition hierarchy to the version of the composite object at the highest level in the composition hierarchy.
For top-down creation, the opposite action is taken.
We do not have to respect any order for the creation of versions of a composite object, i.e a version of a component object can be created, after the creation of a version of a composite object and a version of a component object has taken place. Furthermore, all the versions which constitute the composition do not have to be created, that is, an "old" version can be a component or a composite of a version that we have just created. This method of creation is called mixed creation, that is, both top-down and bottom-up creation.

3.2 Implicit creation : Propagation

Version propagation is an important mechanism for the control of the evolution of objects. It is more commonly used for the control of the evolution of a composite object. Version propagation is the process which automatically incorporates new versions of the composite objects each time a version of one of its component objects is created [8].
The propagation of the creation of versions is not always desirable from the lowest level of the hierarchy up to the highest level [24]. Version propagation always has an objective but it is not always necessary for it to take place because it can cause a considerable proliferation of versions. On the other hand, if propagation is not performed, information is lost. One needs to know when propagation is necessary and when it is not necessary.
A solution is to mark the components which bring about version propagation. In order to do this, the property of the composite object containing the component is defined as being sensitive or significant [13] [24]. Therefore, each time the component object is modified, deleted or created a version of the composite object is modified, deleted or created.

In our model, the creation of versions can be propagated :
- either by the designation of a sensitive composite attribute,
- or by the designation of a sensitive version.

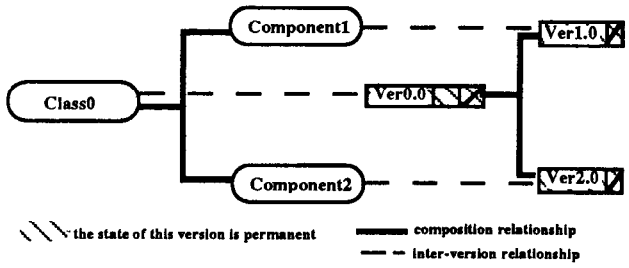
During propagation, the composition relationships between component versions and composite versions which are not affected by the propagation are maintained. These relationships can then be modified by the user as long as the composite version is temporary. In fact, during propagation, only the mother and/or daughter versions of the version which has caused the propagation, are affected by the propagation. The sister versions are not affected by the propagation because they are not linked directly to the version which has caused the propagation.

3.2.1 Sensitive composite attribute

In our model, we can designate a sensitive composite attribute as Zdonik proposed. However, the creation of a component version associated with the sensitive composite attribute does not cause version propagation (class versions or instance versions) to occur because our versions have states (temporary or permanent). Version propagation is performed only when a component version is designated as being permanent by the user.

The class "class0" is composed of two components "component1" and "component2" (see Figure 3). The attribute "Lcomponent1" of "class0" has the value "component1". This attribute has been designated as a sensitive composite attribute by the user. The attribute "Lcomponent2" has the value "component2". This attribute is not a sensitive composite attribute. Therefore, the creation of versions derived from "component2" will not cause the creation of versions from the class "class0". On the contrary, the creation of versions derived from

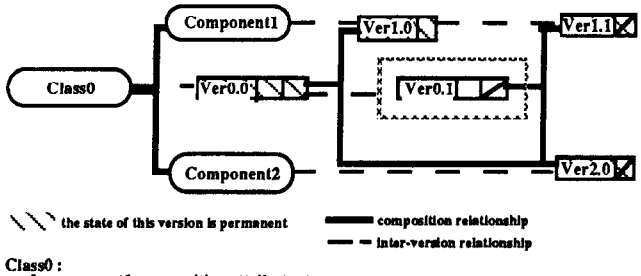
"component1" will cause the creation of versions from "class0".



Class0 :
Lcomponent1 --> sensitive attribute: true
Lcomponent2 --> sensitive attribute : nil

Figure 3 : Sensitive composite attribute

The version ver0.0 of "class0" is composed of the version ver1.0 of "component1" and the version ver2.0 of "component2". The creation of the version ver1.1 from the version ver1.0 does not bring about version propagation, although the attribute "Lcomponent1" is designated as a sensitive composite attribute. On the other hand, the fact that ver1.1 becomes permanent (see Figure 4) causes version propagation as the attribute "Lcomponent1" is a sensitive composite attribute. Therefore, the version ver0.1 of "class0" is created (see Figure 4). The version ver0.1 of "class0" is composed of version ver1.1 of "component1" and the version ver2.0 of "component2". The composition relationship between ver2.0 and the version ver0.0 is not affected by the propagation. It is therefore maintained in the version ver0.1.



Class0 :
Lcomponent1 --> sensitive attribute: true
Lcomponent2 --> sensitive attribute : nil

Figure 4 : Version propagation caused by a sensitive composite attribute

3.2.2 Sensitive version

Another way to propagate the creation of versions (class versions or instance versions) is to designate a permanent version as being sensitive for propagation. When the user thinks a version is stable, he designates this version as permanent. He can also designate it as sensitive and therefore, composite and/or component versions of this version will be created. Bottom-up, top-down or mixed propagation will be activated.

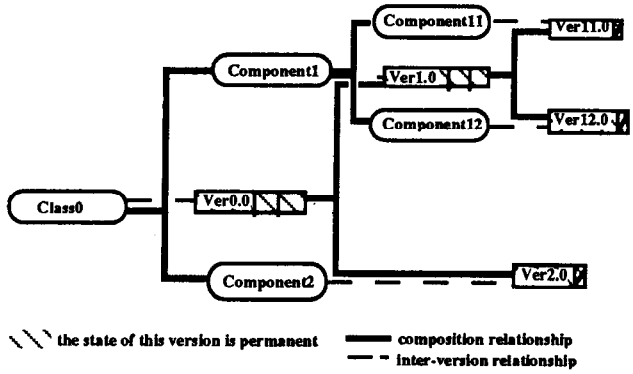


Figure 5 : Composite versions

In this example (see Figure 5), the class "class0" is composed of the classes "component2" and "component1". The latter is itself composed of two classes "component11" and "component12". All these classes are versionable. The composite attributes "Lcomponent1" and "Lcomponent2" of the object "class0" are not sensitive composite attributes. The composite attributes "Lcomponent11" and "Lcomponent12" of "component1" are also not sensitive.
The version ver0.0 of the class "class0" is composed of the version ver2.0 of "component2" and of the version ver1.0 of "component1". The latter is composed of the version ver11.0 of "component11.0" and the version ver12.0 of "component12".
The evolution of the version ver1.0 of "component1" in the version ver1.1 does not bring about version propagation. On the other hand, the fact that ver1.1 becomes a sensitive version causes version propagation (see Figure 6). Firstly, bottom-up propagation of versions is performed and the version ver0.1 of "class0" is created. It is composed of the version ver2.0 of "component2" which is not affected by the propagation and the version ver1.1 of "component1". Secondly, top-down propagation is executed. The version ver11.1 of "component11" and the version ver12.1 of "component12" are created and become component versions of the version ver1.1. This version has caused mixed propagation to occur.

Having described the different concepts of version modeling and management, we will now explain the implementation of these concepts in an object-oriented environment.

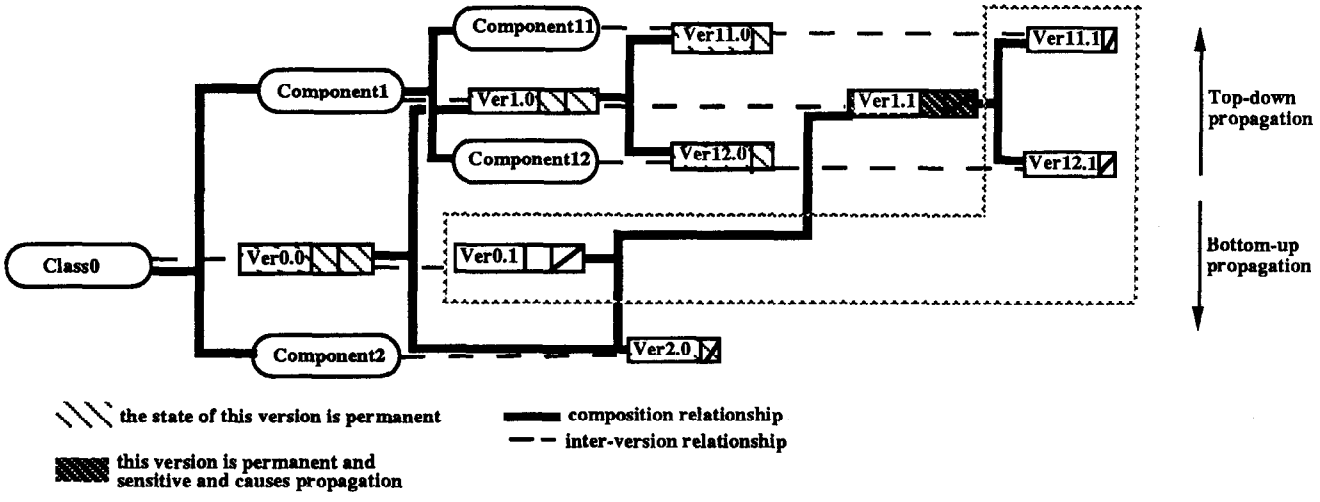


Figure 6 : Propagation caused by a version designated as sensitive by the user

4 Implementation

We have chosen the object-oriented approach as a modeling support because we think that it is more adapted to support a hierarchical structure containing complex objects. The object-oriented approach also allows an important property of objects to be used : inheritance.
Our objects have a frame structure. The inheritance relationship between two classes is the "AKO" [19] inheritance relationship.

The set of concepts associated with the class "generic class version" are, in the case of an object approach, represented by the following attributes :
- the *next-number* allows us to know the number of the next version
- the *default-version* allows the current version to be known
- the *state* allows us to know if a version is permanent or temporary
- the *sensitive* allows us to propagate or not the creation of versions
- the *derivative* contains the names of the versions which have been derived from the version under consideration
- the *derivation* allows the derivation hierarchy of this version to be known
- the *composed-of* attribute is used only by composite versions. It contains the versions of which the version under consideration is composed
- the *instances-linked-to* allows the names of instance versions which have been instantiated from this class version to be known.

The class "generic instance version" from which each instance version is created, is a sub-class of the class "generic class version", for which the *instances-linked-to* attribute has been masked.

The *next-number*, *default-version* and *derivation* attributes are contained only in the sub-classes of the "generic class version" and "generic instance version" classes. The other attributes are in all the class and instance versions because these attributes are specific to each version.

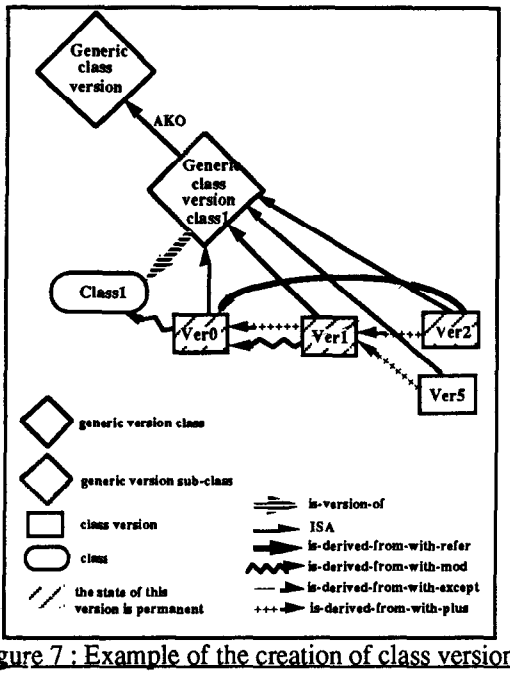


Figure 7 : Example of the creation of class versions

We will present in detail the representation of the class versions Ver0, Ver1 and Ver2 which appears in Figure 7. In this figure, we have the class "class1" which is versionable. It has the properties P1, P2 and P3. Since the

class "class1" is versionable, a class "generic class version class1" is associated with it. This class is a sub-class of the "generic class version" class. "Generic class version class1" is linked to "class1" by the "is-version-of" relationship (see Figure 8). The version ver0 which is permanent (hachured in Figure 7) is derived from the versionable class "class1" and the version ver1 is derived from the version ver0. The versions ver2 and ver5 are derived from the version ver1.

Generic-class-version-class1
AKO : Generic-class-version
Is-version-of: Class1
Next-number : 6
Default-version : Class1.Ver5
Derivation :
((Class1.Ver0 (Class1.Ver1 (Class1.Ver2 Class1.Ver5))))

Class1.Ver0
ISA (Class1 Generic-class-version-class1)
State : permanent
Derivative: Class1.Ver1
Is-derived-from-with-mod : (class1 P1)
P1 : V1.2

Class1.Ver1
ISA(Class1.Ver0 Generic-class-version-class1)
State : permanent
Derivative : (Class1.Ver2 Class1.Ver5)
Is-derived-from-with-plus : (Ver0 P4)
P4 : V4.1
Is-derived-from-with-mod : (Ver0 P2)
P2 : V2.2

Class1.Ver2
ISA (Class1.Ver0 Class1.Ver1 Generic-class-version-class1)
State : permanent
Is-derived-from-with-refere : Class1.Ver0
Is-derived-from-with-plus: (ver1 p5)
p5 : V5.1

Figure 8 : Example of class versions

The version ver0 is created from the class "class1" with modifications to the property P1. The version Ver1 is created from the version Ver0 with the addition of the property P4, the value of which is V4.1, and with the updating of the property P2, which will have the new value V2.2 (see Figure 8).
The version Ver2 is created from the version Ver1. An "is-derived-from-with-refer" relationship has been established with the version Ver0 in order to have all the properties and values of the version Ver0. With this link, Ver2 has the property P2 of value V2.1. Without this link, Ver2 will have the property P2 but its value will be V2.2. The "is-derived-from-with-plus" relationship has been set up in order to add to the version Ver2 the property P5, the value of which is V5.1.

Creating an instance version from the version ver1 involves the creation of the instance version ver1.0 (see Figure 9). This instance version is linked to the class "generic instance version class1" and to the class version ver1 by the "ISA" relationship. The evolution of this instance version means that the instance version ver1.1 is created.

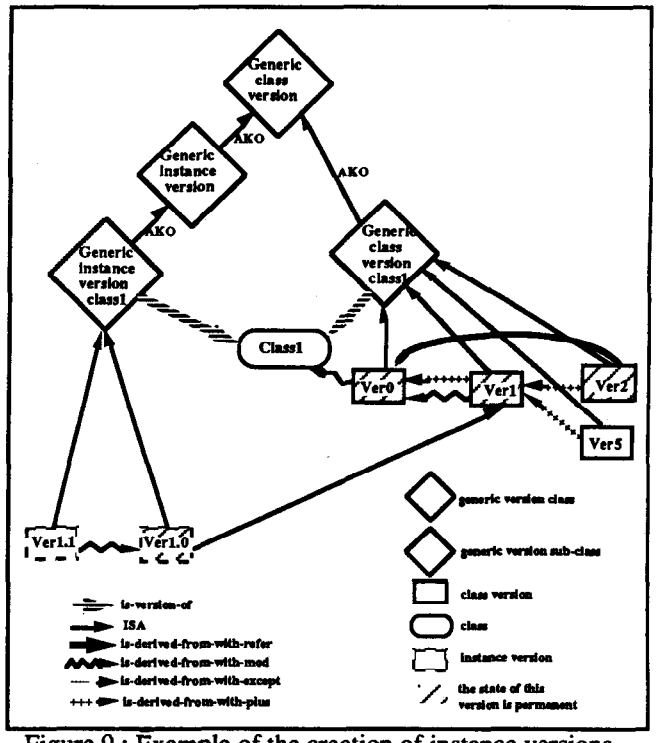


Figure 9 : Example of the creation of instance versions

5 Example of application

We will now give an example of application in order to explain the different concepts of versions that we have defined. Our example is based on the telecommunications networks which are modeled thanks to hierarchical/multi-view modeling [13]. We will briefly describe the concepts of this modeling. Finally, we will present in detail the notion of version associated with an abstraction level of this modeling.

5.1 Hierarchical/multi-view modeling

An advantage of this modeling is the structuring of the model. This modeling represents both the element hierarchy of which the model is composed, and the way this has been created, that is to say, its history.
This modeling enables us to have several representations of a same real system. These representations correspond to different views of a real system and each of these views may have any number of hierarchical levels.
Hierarchical/multi-view modeling allows the models to be structured [13] according to :
- views which reflect the different aspects of the model during the successive stages of a given application
- and abstraction levels which include the data structure of the model.

The building of an abstraction hierarchy has the advantage of reducing a complex system into a series of easily processed sub-problems. These levels make it possible to have a progressive approach to the difficulties.

5.2 Example of versions associated with an abstraction level

In order to explain the different inter-version relationships, we will take as an example an abstraction level which is called the "system-level" and we will make it evolve (see Figure 12) and therefore, create class versions and instance versions from this abstraction level.
The class "System-level", with which we are concerned, is composed of the node "STM1" and of the edge "Ring-STM1". This class becomes versionable and therefore, the class "generic-class-version-system-level" is created. Class versions can now be created from the class "system-level". The class version "system-level-Ver0" is created from this class with the addition of the edges "Ring-STM4" and "system" and of the node "STM4". It is the "is-derived-from-with-plus" relationship which allows this addition of edges and nodes to be taken into account (see Figure 10).

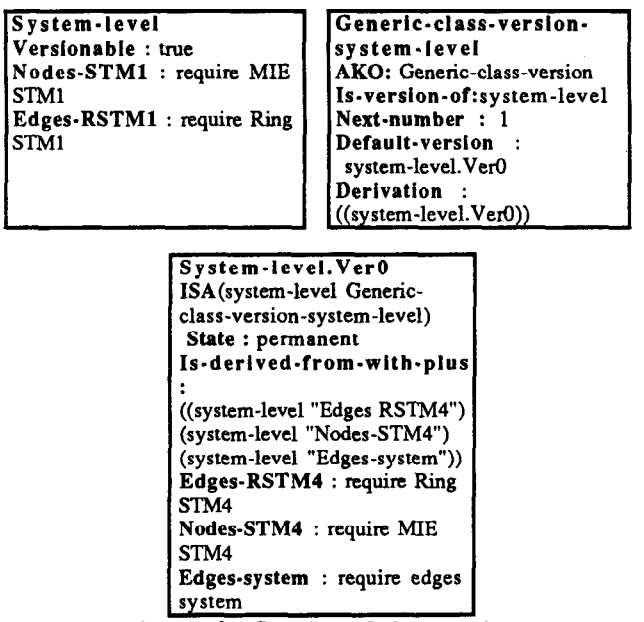


Figure 10 : Creation of class versions

From this class version, instance versions are created; the first is called "system-level-ver0.0". This instance version is the first instance version of the versionable class "system-level" and thus, the class "generic-instance version-system-level" is created (see Figure 11). The version "system-level-ver0.0" represents communication routes of a urban network. With this representation, we quickly notice that the demand for the two rings on the right of Figure 12 is saturated. In order to solve this problem, we need to replace the two rings of type STM1 (on the right of Figure 12) with a ring of type STM4 which allows four times more information to flow than a ring of type STM1.
In order to carry this out, we derive the version "system-level-ver0.1" from the version "system-level-ver0.0" (see Figure 11). In this new version, we eliminate the two rings of type STM1 in the attribute "Edges-RSTM1", as well as the five nodes of type STM1 in the attribute "Nodes-STM1" thanks to the "mod" relationship. We obtain a new version which we test in order to verify that the problem of saturation has been resolved and that no other problems have appeared.

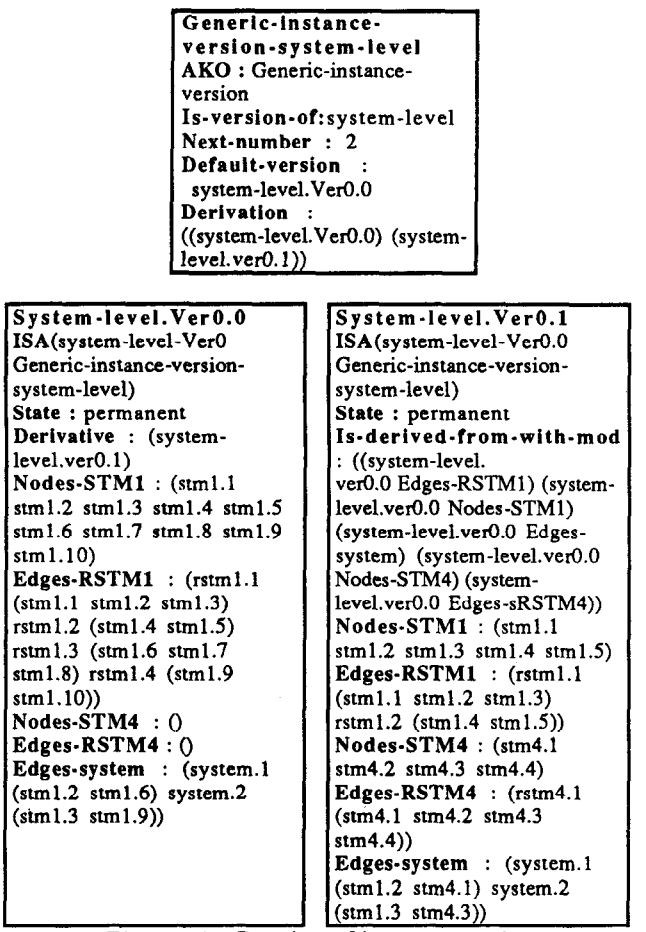


Figure 11 : Creation of instance versions

After some time, new communication requirements appear. In order to take them into account, we create the instance version "system-level-ver0.2" from the instance version "system-level-ver0.1". We add new edges of type "system" to this new version in order to accept the new demands and we test this new version to verify that it corresponds to the initial needs and does not involve other problems. It is the "is-derived-from-with-mod" relationship which allows the attribute "Edges-system" to be modified.

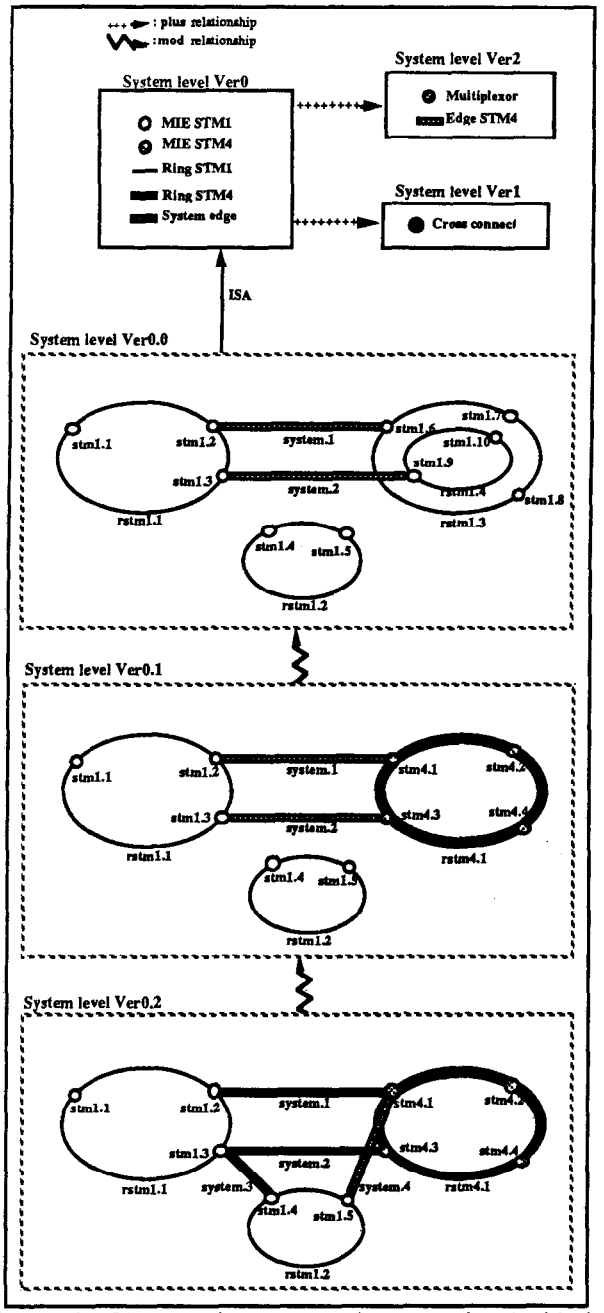


Figure 12 : Class and instance versions associated with the "system-level"

At a time t, we decide to derive a class version from the abstraction level "system-level-ver0. We create this new version because we think that the former version will soon be saturated. We therefore provide a new version to avoid the possible problems that we will not be able to solve with the only equipment which is available in the version "system-level-ver0". We add a new node of type "cross-connect" to the new version in order to gain in the flexibility of points. This is done thanks to the "plus" relationship.

We can also make other tests from the abstraction level "system-level-ver0" by adding a node of type "multiplexor" and an edge "STM4" to it. In order to do this, we derive a new version called "system-level-ver2" from "system-level-ver0" and with using the "plus" relationship, we add a new node of type "multiplexor" and a new edge of type "STM4" to this new version. This new version is then tested.

Most evolution (the creation of class versions or instance versions) is due to problems of saturation, which may also be financial or security problems of an existing network.

6 Conclusion

In this paper, we have identified the concepts of taxonomy of versions, states of versions, version evolution, composite versions and version propagation.

A version may be :

- a class version
- or an instance version.

The successive class or instance versions from a same object are linked to each other so that the evolution from one version to another can be followed more easily. The clarification of the relationships between the different versions allows the different users to manage the evolution of their model better. For each version, they know the modifications which have been made since the previous version.

The objects we use have a complex structure because they are composed of other objects. In order to take this into account, we have introduced the concept of composite version. This means that we have to deal with the problems concerning the propagation of versions which are components of other versions. In order to handle this, we need the concept of sensitive version which allows the creation of composite versions to be propagated each time a version becomes sensitive. We also need the concept of sensitive composite attribute which allows composite versions to be created each time that both a version of the component associated with this sensitive composite attribute is created and it is designated as permanent by the user. The propagation of versions will be performed for the composite versions.

The model for modeling and managing versions has been developed with the Yafool [24] object-oriented language. It has been implemented in the Presage system [3] [17].

References

[1] D. Beech and B. Mahbod, "Generalized Version Control in an Object-Oriented database", *IEEE Conference on Data Engineering*, Los Angeles, CA 1988.

[2] A. Bjornerstedt, C. Hulten, "Version control in an object-oriented architecture", *Object-oriented Concepts, Databases, and Applications*, Edited by W. Kim, F.H. Lochovky, By ACM 1989.

[3] A. Caminada, C. Oussalah et al., "Modeling concepts for telecommunications network planning", *Artificial Intelligence, Expert Systems and Symbolic Computing*, , in Houstis E.N., North-Holland, 1992.

[4] H.T. Chou and W. Kim, "A unifying framework for versions in a CAD environment", in *Proc. Int. Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986.

[5] H.T. Chou and W. Kim, "Versions and change notification in an object-oriented database system", in *Proc. 25th ACM/IEEE Design Automat. Conf.*, June 1988.

[6] K.R. Dittrich and R.A. Lorie, "Version support for engineering database systems", *Transactions on Software Engineering*, Vol. 14, N° 4, April 1988.

[7] R.H. Katz et al., "Design version management", *IEEE DESIGN&TEST*, pp. 12-22, February 1987.

[8] R.H. Katz, "Toward a unified framework for version modelling in engineering databases", *A C M Computing Surveys*, Vol. 22, N° 4, pp 375-408, 1990.

[9] W. Kim, D. Batory, "A model and storage technique for versions of VLSI CAD Objects", *Conference on foundations of Data Organization*, Kyoto, May 1985.

[10] W. Kim, H.T. Chou, "Versions of schema for object-oriented databases", *Proceedings of the 14th VLDB Conference*, Los Angeles, California, 1988.

[11] W. Kim, E. Bertino, J.F. Garza, "Composite objects revisited", in *Proc.ACM SIGMOD Intl. Conf. on Management of Data Portland*, OR Vol. 18, No2, June 1989.

[12] W. Kim, "Introduction to Object-oriented databases", MIT Press, Cambridge Massachusetts, 1990.

[13] G.S. Landis, "Design Evolution and History in an Object-Oriented CAD/CAM Database", *IEEE COMPCON*, San Francisco, CA 1986.

[14] C. Oussalah, "Modèles hiérarchisés / Multi-vues pour le support de raisonnement dans les domaines techniques", Thèse de Docteur d'Université, Université d'Aix-Marseille, 1988.

[15] C. Oussalah, G. Talens and N. Giambiasi, "Version management for the modelling of complex systems", *International Conference on Economics / Management and Information Technology 92*, August 31-September 4, 1992 Tokyo.

[16] C. Palisser, "Le modèle de versions du systèmes Charly", *6èmes journées Bases de Données Avancées*, INRIA, Montpellier, Septembre 1990.

[17] Rapport interne C.N.E.T. - L.E.R.I., PRESAGE : Un atelier pour la planification de réseaux, Avril 1991.

[18] R. B. Roberts and I. P. Goldstein : The FRL Manual; AI Laboratory; *MIT, AI Memo 409*, Cambridge, Massachusetts, 1977.

[19] A.H. Skarra, S.B. Zdonik, "The management of changing types in an object-oriented database", *OOPSLA Conference*, Portland, September 1986.

[20] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", *Proceedings of the 1st OOPSLA*, Portland, Oregon, pp. 38-45, 1986.

[21] M. Stefik and D. G. Bobrow, "Object-oriented Programming : Themes and Variations, *The AI Magazine*, Vol. 6, n°. 4, pp. 40-62, 1986.

[22] W.F. Tichy, "Design, Implementation and Evaluation of a Revision Control System, *Proceedings of the 6th Conference on Software Engineering*, pp. 58-67, Japan 1982.

[23] YAFOOL, "Manuel d'Utilisation ", SEMA Group, Mai 1991.

[24] S. B. Zdonik, "Version management in an object-oriented database", International Workshop, Trondheim, Ed Reidar Conradi et al., *Lecture Notes in Computer Science*, No244, June 1986.