Armazenamento da Informação-SBi
Banco: Dados orientados: Objetos
Versões: Banco: Dados
CNPq 1.03.03.00-2

Versions and configurations in object-oriented database systems: a uniform treatment

Lia Goldstein Golendziner
Clesio Saraiva dos Santos *

Abstract

Object-oriented database models usually allow versions only at the most specialized type/class in an inheritance hierarchy. The possibility of having versions at different levels of abstraction provides a richer model and allows a more natural representation of the reality. The presence of objects and its corresponding sets of versions at different levels of a type/class hierarchy introduces the need for handling version mappings. Integrity constraints can be associated to these mappings, restricting the set of possible combinations of versions appearing at different levels of the hierarchy. Sets of versions associated with each level of an object hierarchy often represent a very large set of possible configurations for that object, which is difficult to be handled directly by the user.
In this context, adequate mechanisms are very important to define and build object configurations by means of selections applied to the set of all possible configurations, defined by the combinations of versions. This paper proposes an approach in which versions and configurations may appear at different levels of an inheritance hierarchy, and a uniform treatment is given to these two concepts.

Keywords: Versions, object-oriented data models, configurations, dynamic references

1 INTRODUCTION

In the context of object-oriented database systems, versions allow the simultaneous representation of many object states. A version represents an identifiable state of an object, considered by the user as semantically significant, and must be handled by the data model as any other object in the system.

Research related to versions were motivated by the requirements of some application areas, mainly Engineering applications (CAD-Computer Aided Design), Software Engineering (CASE-Computer Aided Software Engineering), Manufacturing (CAM-Computer Aided Manufacturing), Office Automation, Hyperdocuments and Historic Databases.

The work in engineering applications focused mainly at the problems related to object representation [2, 6, 16, 17, 21, 23, 28, 32], and are based in semantic data models like the Entity-Relationship Model, that is the most frequently used. The papers from

* clesio@inf.ufrgs.br, Instituto de Informática, UFRGS, Porto Alegre, RS, Brasil

Haskin&Lorie [20] and Lorie&Plouffe [30] do not approach directly the version concept, but are important due to the introduction of the concept of complex object and their related mechanisms: queries involving complex objects, design transactions and checkin and checkout operations. In [24], Katz argues that many proposals presented in the area of engineering applications are similar, and he proposed a basic terminology together with a collection of mechanisms that must be present in any approach to represent this kind of information.

In CASE applications, research approaches specially the aspect of systems configuration (ex: [4, 31]). Few works give emphasis on database utilization [3, 13, 22].

With historical databases, the emphasis is put on the storage of the information about entities, organized with respect to time. Some works were developed aiming at the extension of database models with temporal concepts and mechanisms, starting with the Relational Model [9, 34, 37]. More recently, object-oriented database models were also extended [18, 19, 39].

Presently there is a trend to extend object-oriented database models and systems with version concepts and mechanisms, aiming at the definition of a framework, that may be refined to support many classes of application [1, 8, 26, 29, 35, 36]. Some works approach the use of versions to support database schema evolution [8, 11, 33, 38, 40].

Object-oriented database models usually allow versions only at the most specialized type/class in an inheritance hierarchy [1, 3, 26]. The possibility of having versions at different levels of abstraction provides a richer model and allows a more natural representation of the reality. On the other hand, when versions can be associated to database objects, the user needs to choose from a possibly large set of options, the specific combination of versions that will compose the object in each situation. Each combination of specific component versions of an object is called a configuration.

Configurations are very important in the context of object-oriented databases that supports versions, since they are, in fact, the objects handled by the applications. They correspond to the instances of objects in versionless databases. In this context, adequate mechanisms are very important to define and build object configurations by means of selections applied to the set of all possible configurations defined by the combinations of versions.

This work focuses on version management at the application level, to support the representation of sequence- or time-dependent information, as defined by the users. The user must be allowed to edit the sequence (or graph) of versions, defining when and where a version must be included or removed.

Versions and configurations are commonly treated as different concepts in most of the models that support version management. This separation of concepts has some drawbacks, such as the impossibility of freely combining versions and configurations to construct higher level versions. This kind of distinction is particularly inconvenient in the case of object-oriented databases, where the uniform treatment of everything as object increases significantly the possibility of combination of objects to form other higher level objects.

The approach proposed in this paper does not make substantial distinction between versions and configurations, which may be freely combined. Special attention is dedicated to object-oriented database systems. The need to incorporate new concepts to the data model is discussed. Emphasis is given to the versioning of objects participating in inheritance hierarchies, as well as relationships between objects and versions located at different levels of the hierarchy.

A version model is proposed, in which the versioning of objects at all levels of an inheritance hierarchy is allowed, not restricting the versioning to the leaf level. It is shown how these extensions to the object-oriented paradigm allow a more natural modeling of many real world situations, specially when the objects are constructed in a top-down process.

The next sections are organized as follows: Section 2 presents the main concepts related to versions in object-oriented databases, which are used in this paper. The aspects related to object and version hierarchy are discussed in Section 3. The main effects of object versioning at different levels of an inheritance hierarchy are presented. The advantages of this approach in modeling applications are discussed and compared to the traditional approach where versions are concentrated at the leaves of the hierarchy. Configurations are discussed in Section 4. Section 5 presents the operations defined for manipulation of objects and versions in the inheritance hierarchy and Section 6 presents the facilities required for configuration specification. The main conclusions of this work are presented in Section 7.

2 CONCEPTUAL BASE

2.1 Version and versioned object concepts

A version is a description of an object at certain time or from a certain point of view, which is considered relevant for a defined application. In an object-oriented model, a version is a first class object, having an Object Identifier (OID). A version can then be directly manipulated or queried.

Versions of a real world entity must be kept together and constitute a *versioned object*. A versioned object is also a first class object and maintains information about its associated versions. A versioned object can have properties, which should be common to all its versions. Each version belongs to exactly one versioned object.

Considering that applications can not always determine if an object will present versions or not, objects can dynamically change from non versioned to versioned.

Objects (versioned or not) having the same properties and behavior can be grouped into classes. Since the feature of being versioned belongs to an object and not to a class, a class can have versioned and non versioned objects as instances. An automobile being designed can be considered as a versioned object, having several versions associated, which represent the different stages throughout its design. Figure 1 illustrates this example. The notation used is based on that introduced in [25]. Being objects, both versions and versioned objects can be referenced by other objects through its respective OID, as well as be used as parameters of operations.

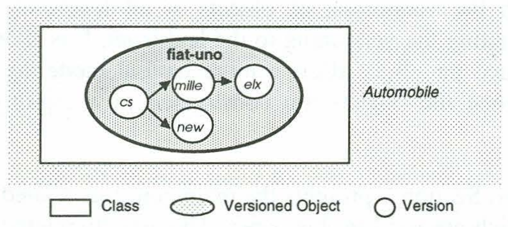

Figure 1 - Versioned object and its versions

Each versioned object has one version considered as its *current version* (sometimes called *default version*). The current version is automatically maintained by the system as the most recently one. The designer can specify a different version to be the current one, but in this case, the current version will remain fixed, not being affected with the creation of new versions. The current version is used whenever an operation is applied to a versioned object, without specifying one of its versions.

2.2 Version Properties

Versions of a versioned object are related through a derivation relationship, which form a directed acyclic graph. For the version *mille* (figure 1), version *cs* is called its predecessor and version *elx*, its successor. A version can have several successors and predecessors.

When a version is created as a successor of another one, a copy is made of its predecessor. When a version is created as a successor of more than one version, only the first version indicated is copied, and a relationship is established with the others indicated. The idea is that the new version should be a merge of its predecessors, but it is user's responsibility to extract the necessary information from the several versions. Merge operation is a difficult task, not implemented by any system at the present moment.

Versions have a status, reflecting its robustness, that can be *working*, *stable* or *consolidated* (similar to the classification in [3, 26]). Operations on versions are restricted, according to their status. A *working* version is essentially a temporary version that has to undergo modifications to reach a more stable status. A *stable* version has reached more stability and can be shared. Stable versions can not be modified, but can be removed. A *consolidated* version is a final version that can neither be modified nor removed.

New versions are created with working status. When a version is derived from another one, its predecessors are automatically promoted to stable, thus avoiding modifications on versions that were important from a historical point of view. The user can explicitly promote versions from working to stable, or from stable to consolidated.

2.3 Static and Dynamic References

When an object having versions is used as component of another one, references to it can be made in one of two ways: reference to a specific version -called *static reference-* or reference to the versioned object -called *dynamic reference* [1, 26] (or generic in [3]). A static reference behaves as a simple reference to an object, and the composite object is said to be *statically bound* [26] to the version. A dynamic reference means that a specific version will be chosen at run time, and the composite object is said to be *dynamically bound* to the versioned object.

Composite objects can be built recursively, resulting in an aggregation hierarchy. Figure 2 shows two versions of a composite object from the class *Automobile*. The version *cs* has a static reference to the first version of the object fiat-motor, i.e., the value of the attribute motor is the OID of the first version of the object **fiat-motor**. The version *new* contains a dynamic reference to the versioned object **fiat-motor**, i.e., the value of the attribute motor is the OID of the versioned object.

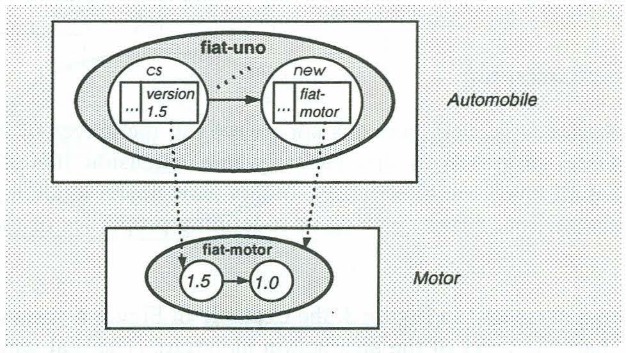

Figure 2 - Static and dynamic references

The replacement of a versioned object reference by a reference to a specific version is called *dynamic reference resolution*. Mechanisms are needed for this resolution, that occurs in two situations: 1) when the referenced object is accessed and 2) when a configuration is built for the composite object. In the first situation, the current version is used. In the configuration process, different options are provided for selecting a version associated to the versioned object. For example, selection can be based on pre-defined criteria, such as the first, the most recent, or in expressions containing attribute values of the versions.

3 OBJECT AND VERSION HIERARCHIES

3.1 Inheritance

Inheritance is one of the basic concepts in object-oriented databases [5] and one of the reusability mechanisms. *Refinement and extension* [7] are the two ways in which inheritance can occur.

Inheritance by refinement is the most traditional approach, corresponding to the *is-a* relationship between objects. In this case, there is a migration of properties through the levels of the hierarchy, from top to bottom. The leaves may be seen as complete instances of the objects, including all the non-conflicting properties of their ascendants, as well as properties that result from conflict resolution. This type of inheritance is present in many object-oriented database systems, such as O_2 [15], ORION [27], GemStone [10].

Extension inheritance is related to the idea of prototypes and appears in data models such as PEGASUS [7, 36] (extension of EXTRA [12]). In this case, each property refers to a specific level of the hierarchy, modeling some relevant aspect of the real world object. The union of all levels models the complete object corresponding to the considered hierarchy.

3.2 Object and version mapping

When refinement inheritance is used, versions appear only at the leaves of the hierarchies [3, 8, 26]. In the model proposed in this paper, where extension inheritance is used, versions are allowed at all levels simultaneously. In this way, object modeling can be done at various levels of abstraction, either defining or redefining properties of the objects, one level at a time.

Considering the schema presented in Figure 3, the example in Figure 4 shows the modeling of versions at more than one level of the abstraction hierarchy. The real world entity **fiat-uno** is represented at two levels of abstraction: *Vehicle*, with the properties *motor* and *fuel*, and *Automobile*, with the properties *drivetrain* and *accessories*. In each of these levels, there are versions associated to the corresponding versioned objects. Each version must have at least one corresponding ascendant, to which it is bound at creation time. In some situations, one version may have more than one ascendant.

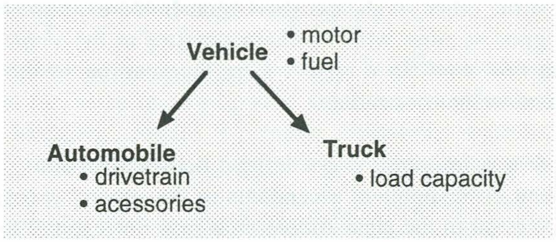

Figure 3 - Example schema

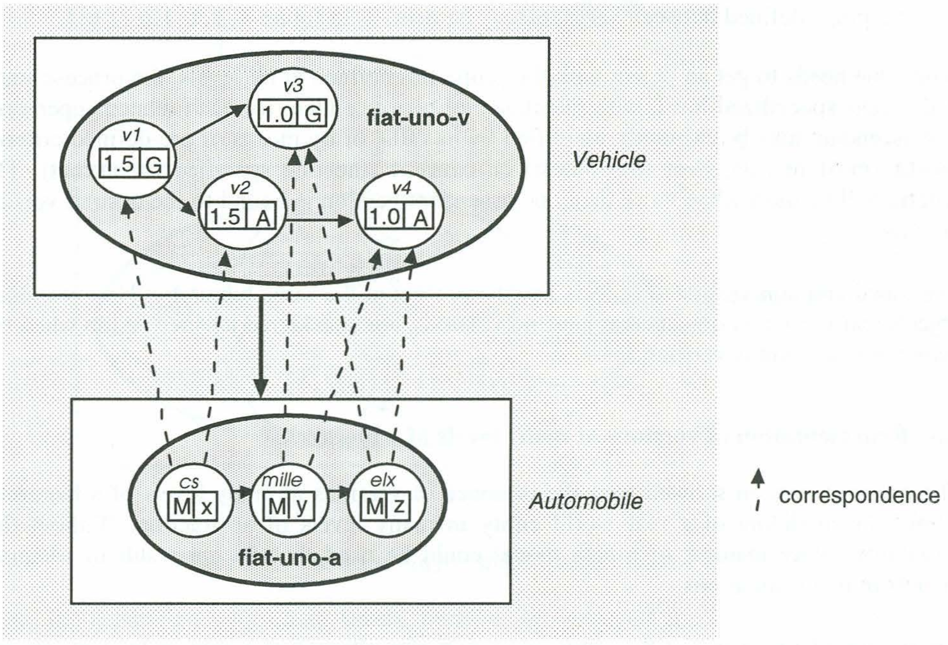

Figure 4 - Versions represented at more than one level of the inheritance hierarchy and their correspondences

In the example, the same characteristics defined for version **cs** (of **fiat-uno-a**, at the *Automobile* level) may be bound to different versions of **fiat-uno-v** (at the *Vehicle* level), representing the two options for the model **fiat-uno cs**, one using gas and the other using alcohol. On the other hand, the same version in a superclass may correspond to more than one version in a subclass. This situation occurs in the example of Figure 4, where one version of **fiat-uno-v** (ex: *v4*) corresponds to two versions of **fiat-uno-a** (*mille*, *elx*).

As illustrated, the design of a new automobile may be carried on starting at the top level and having the details of the other levels specified later. In the example, a new automobile (or truck) may be designed starting with its characteristics at the *Vehicle* level, and thus creating the versions at this level. In a further step, the versions at the *Automobile* level may be created and bound to their ascendants.

In this way, *correspondences (mappings)* are defined between versions of an object at one level and versions of their correspondent ascendants in the superclass(es). In Figure 4, the mapping is *n:m*. Each version in the class *Automobile* may correspond to n versions in the superclass *Vehicle*, and vice versa. The mapping defines an integrity constraint, which is specified with the definition of the inheritance relationship between a class and its superclass, in the database schema. It is system's responsibility to enforce the constraint. The mappings defined between versions may be **n:m**, as in figure 4, **1:1**, **1:n** or **n:1**.

When one needs to get an object with the properties defined at all levels, the process starts at the most specialized level, with the choice of one ascendant for each related superclass. The ascendant may be explicitly identified by its OID, or by means of pre-defined criteria: **recent** (most recent), **first** (the oldest) or **current** (the one specified as current). The criteria will be used when more than one ascendant version is bound to the desired version or object.

Versioned and non versioned objects may be present at the same hierarchy. Non versioned objects and versioned objects that have no versions, are considered as one version when the mapping constraint is verified.

3.3 Representation of versions at many levels of a hierarchy

The previous section showed how the presence of versions at many levels of a hierarchy allows the modeling of a real world entity in many levels of abstraction. Without this possibility, other features of a data model could be used, but do not result in adequate models in many situations.

Considering the example of the previous section, one alternative would be to start creating one version of a *Vehicle* object, which would be later on refined, by the addition of *Automobile* properties. The solution would be to migrate the *Vehicle* object to the *Automobile* subclass. The problem with this solution is that object migration is not allowed, in general. The reasons commonly pointed out are the need to redefine the OID of the migrating object (because the class is part of the OID [27]), and the possible existence of versions derived from the migrating version/object.

Another possibility is the creation of versions directly in the class *Automobile*, but only with the *Vehicle* properties (the other properties receiving null values, which are redefined later). In this case, a restriction must be imposed: the actual values for the undefined properties must be set before deriving a new version from this one (when a new version is derived, its ancestors may not be changed anymore).

When versions are allowed at only one level, it is difficult to find out differences and similarities between the versions, concerning the characteristics defined at different levels of the inheritance hierarchy. Figure 5 presents a possible representation of the situation modeled in Figure 4, but with versions appearing at one level only.

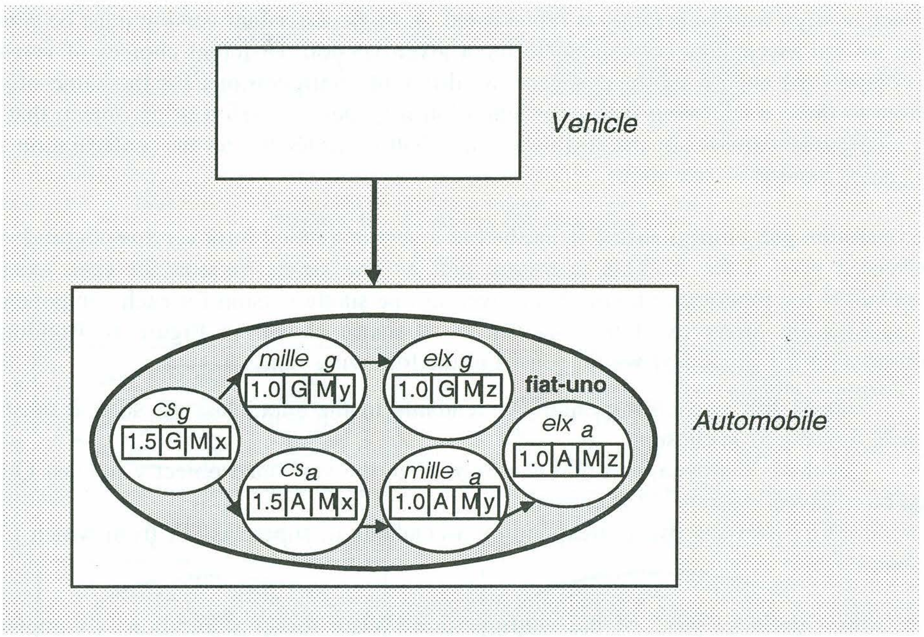

Figure 5 - Versions only at the most specialized class

In the representation with many levels, versions are grouped according to the values of their properties. In Figure 4, for example, one may obtain all versions of **fiat-uno** at the *Automobile* level, which have the *1.0 motor* and *gas fuel*, by getting the descendants of the version *v3* (*Automobile* level). Versions at one level may be considered as alternatives, for which versions are created at the lower levels of the hierarchy.

4 CONFIGURATIONS

4.1 The configuration concept

In a composed object, where components can be versioned objects, a configuration binds exactly one version of each of its components to the composed object. Since objects are hierarchically composed, a configuration must recursively include definitions for all the objects in the aggregation hierarchy. Dynamic references must be resolved and the system must choose a version according to some pre-defined criteria.

In the proposed model, since a real world entity can be modeled at various levels of abstraction in the inheritance hierarchy, the configuration must also define a single version for each level in which the entity is represented. A single ascendant version must be chosen when several ascendants correspond to the a given version. Different choices of versions for components and/or ascendants generate different configurations for the same object, leading to the consideration that a configuration is a special version of an object, that we call *configured version*. In the following, we shall consider the terms configuration and configured version as synonyms.

The operation **get_configuration** is applied to a version (called *base version*) to produce a configured version for it. This operation defines one single version for each existing ascendant in the inheritance hierarchy, as well as one single version for each component in the aggregation hierarchy. Lets consider the scenario shown in Figure 6. Building a configuration for version b1 would consist of the following steps:

1) b1 has two corresponding ascendants in the superclass A, so one must be chosen. Assume a2 is chosen.

2) there is a dynamic reference from a2 to the versioned object y in class Q that must be resolved. Assume q2 is chosen.

3) q2 also has two corresponding ascendants in superclass P, from which p2 is chosen.

Each choice defines "part" of the configuration. When the configuration is completely defined at one level, a configured version is created as a successor of its base version. Figure 7 shows the configured versions created. Version c1 is the starting point; versions c2, c3 and c4 were created as the result of choices made in step 1, 2 and 3, respectively. In this example, only one choice had to be made for each version that should be configured.

However, several choices could be necessary for one version, if there were more than one dynamic reference and/or multiple ascendants in several superclasses. Thus, the configuration process is recursive, consisting of the following steps for each version:

1) dynamic reference resolution, choosing one of the versions associated to the referenced versioned object;

2) ascendant version definition, for each of the superclasses.

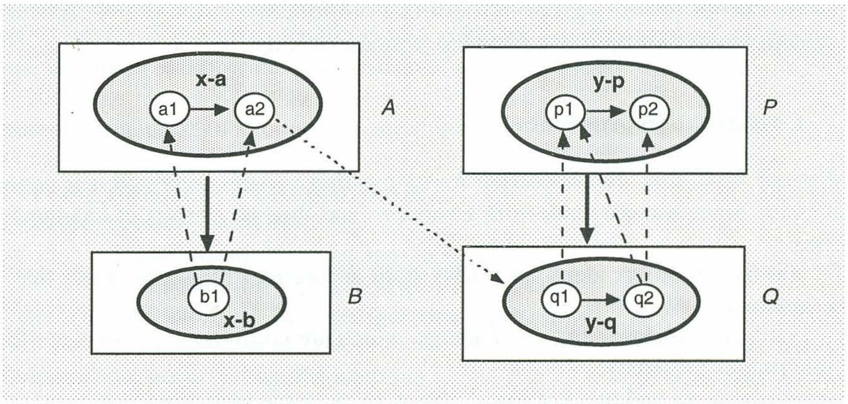

Figure 6 - Scenario for configuration

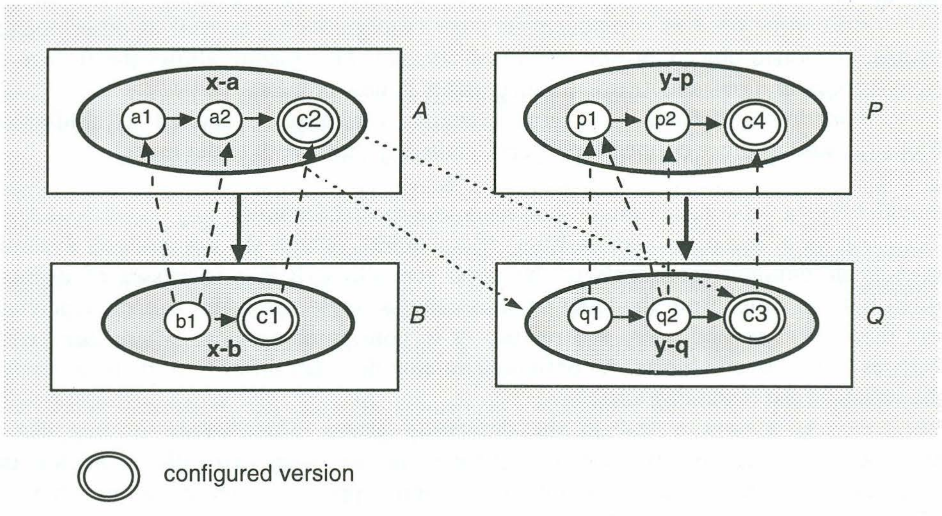

Figure 7 - Configured versions and their relationships

A configured version can only reference other configured versions, so that the system can insure that a configuration is completely specified - no dynamic references or multiple ascendants exist anymore. Configured versions can be shared by other objects, either regular versions or configured ones. Versions c2, c3 and c4 can be considered as "partial configurations", which can be referenced by other configurations. Providing partial configurations is advantageous when objects are composed of a great number of components, since the choices made for these components can be reused in new configurations.

4.2 Configuration properties

A configured version is always created for an existing version, to which it is connected as its successor. Since a configuration is a version (and thus an object), it has the following properties:

• it has an object identifier (OID), built according to the same rules that apply to regular versions;

• it has ascendants and descendants, which are configured versions at other levels of the inheritance hierarchy;

• it is associated to a versioned object, in the same way as its base version;

• it has a status, that can be *working*, *stable* or *consolidated.* In this way, configured versions can be tested, being approved or removed by the designer. If a configured version is approved, its removal can be avoided by promoting it to status *consolidated*;

• it can be used as component of other configurations, as well as other objects (versions, versioned objects or non-versioned objects). This feature allows the storage of approved configurations and its use as components of new objects;

• operations defined for regular versions can also be applied to configured versions (as well as those defined for objects), bringing uniformity to the model.

Although being a version, a configuration has some special features:

• it is a complete specification of an object, which do not present dynamic references or multiple ascendants in the same superclass. It is a successor of its base version, but can differ from the latter by presenting a single ascendant in each superclass where there were several ones, and references to configured versions where there were references to versioned objects. A configured version does not differ from its base version in the values of the remaining attributes;

• it is always a leaf in the derivation graph. This feature insures that a configuration can always be removed (unless the designer explicitly promotes the configuration), avoiding a proliferation of non approved configurations. Several configurations can be created for a base version;

• modifications in a configured version can only occur for existing references, that is, references to configured versions can only be substituted by references to other configured versions. Otherwise, a configured version would no longer be a configuration, for example, if a reference to a configured version could be changed to a reference to a versioned object.

5 OPERATIONS ON THE HIERARCHY OF OBJECTS, VERSIONS AND CONFIGURATIONS

The operations defined for objects and versions (also applied to configured versions) can be classified in three groups: operations for creation, operations for navigation in the inheritance hierarchy and operations for retrieval of versions/objects.

The operators for creation of versions are the following:

```
        create_versioned_object (class): OID;
        derive_version ( set(OID)
                [, ascendant: [class1:] set(OID)...]
                [, descendant: [class2:] set(OID)...]): OID;
        get_configuration (OID [, configuration expression]): OID;
```

Version creation can occur in one of four ways:

1) creation of a versioned object and afterwards creation of versions for it. The possibility of creating a versioned object without versions allows references to the versioned object, so that top-down designs can be carried on. Derivation of versions can then go on, using the versioned object OID as parameter. When a version is created, it must be necessarily connected to an ascendant version/object. Optionally, descendants can be informed;

2) derivation of a version from an existing one (or more than one). The new version is a copy of its predecessor. If more than one version is used as parameter, only the first one is copied, but a derivation relationship is kept with all of them;

3) a version can be derived for an object that was non-versioned up to this moment. In this case, the non versioned object becomes the first version of a new versioned object, and a new version is derived from it.

4) a configured version is created with the operator **get_configuration**. A configuration expression can be provided, based on object's properties. If this expression is not supplied, the configuration process considers the current version in case of dynamic references, and the most recent ascendant, in case of multiple ascendants.

The derive operation can also be applied to configured versions. The result of this operation is a copy of the configured version, connected as a successor of the same base version of the copied version.

Operations for navigation in the inheritance hierarchy allow the retrieval of ascendants and descendants of an object, in given classes. The operations are:

```
        get_ascendant (OID, class [criterion/"*"]): set (OID ascendant object );
        get_descendant (OID, class [criterion/"*"]): set (OID descendant object);
```

If more than one ascendant version exists for the desired version, all the versions can be returned (option *) or only one, according to the specified criterion. The criterion could either indicate a manual selection, when the OID of the ascendant version is given, or an automatic selection, when one of the pre-defined options is given. The pre-defined criteria are: **recent**, **first** or **current** (recent is used as default). The **get_descendant** operation is similar to the **get_ascendant**, but applied to descendant objects in the subclass identified.

Retrieval of objects can be made through the following operations:

get_object (OID): list(attribute values);
get_complete_object (OID [, **ascendant**: class1 [: criterion]
[, class2 [: criterion]]...]): list(OID);

The operator **get_object** retrieves attribute values of an object defined at the specified class. This operation returns only the attributes defined at one level of the inheritance hierarchy. To retrieve all the attributes of a real world entity modeled in the database, navigation must occur in the hierarchy, so that all objects representing the given entity be retrieved. The operation **get_complete_object** was defined with this purpose and it returns the ascendants of an object in the inheritance hierarchy, one for each superclass. If only some ascendants are desired, the desired classes must be identified.

When there is more than one ascendant for a version, the criterion is used to select only one, in the same way that happens in the operations **get_ascendant** and **get_descendant**.

Besides these operations, operations for navigation on the derivation graph are provided (**get_first_version**, **get_last_version**, **get_successor**, **get_predecessor**, **get_ versioned_object**).

It must be noted that all these operations apply also to versioned objects, giving uniformity and orthogonality to the proposed model.

6 FACILITIES FOR CONFIGURATION SPECIFICATION

The **get_configuration** operation can be applied to any object that has components or ascendants that must be resolved. Resolution can follow the pre-defined criteria or can be based on a configuration expression supplied by the user. The pre-defined criteria are the following: a) a dynamic reference is replaced by a reference to the current version of the versioned object; b) the ascendant version is the current one, if this version is among the correspondent versions associated to the version being configured. Otherwise, the most recent version of the set of correspondent versions is chosen.

Version selection through the pre-defined criteria is automatic, though restricted. To build configured versions with more flexibility, allowing the user the selection of versions through different criteria, additional facilities are necessary. It is desirable to select versions based on its properties, having the facilities of a query language. However, facilities must be added to a query language to allow references to versions, as well as to express conditions involving the relationships among versions in the derivation and inheritance hierarchies.

The facilities that must be added can be classified in three types: facilities for referencing objects, facilities for navigation in the inheritance hierarchy and facilities for navigation in the derivation hierarchy

Considering a SQL-like language (for example as in [14, 36]), the mentioned facilities are exemplified in the following:

1) Facilities for referencing objects
Along with the existing facilities (dot notation for referencing components of aggregates, use of cursor for referencing set elements), facilities are necessary to reference versions of objects.

```
Example:
select x.motor, x.fuel
from Vehicle v, versions(v) x
where x.motor > 1.0 and
          x.fuel ¬= 'A';
```

The term **versions**(v) allows visiting versions of a versioned object *v*, which is an instance of class *Vehicle*. In Figure 4 this expression identifies version v1.

2) Facilities for navigation in the inheritance hierarchy

With extension inheritance, navigation in the inheritance hierarchy, in the direction descendant-ascendant, is done automatically for non versioned objects, through value inheritance and delegation [Bil 90]. That is, whenever an attribute or method that was not defined for one object at a specific level is referenced, the ascendants of the object are recursively visited until such attribute or operation is found.

However, since the correspondence among versions of an object at different levels can be n:m, there is a need to visit the different versions associated to a given one. The terms **is_ascendant_of** and **is_descendant_of** allow the navigation in the inheritance hierarchy.

```
Example:
select va.*
from Vehicle v, versions(v) vv, Automobile a, versions(a) va,
where vv.motor=1.0 and
          va is_descendant_of vv;
```

In Figure 4, this expression selects versions *mille* and *elx*, which are descendants of versions that have property motor equal to 1.0 (*v3* and *v4*).

3) Facilities for navigation in the derivation hierarchy

Sometimes it is necessary to identify versions according to its relative or absolute position in the derivation hierarchy. The terms **is_first**, **is_last**, **is_successor_of**, **is_predecessor_of** allows, respectively, the identification of the first or most recent version in a derivation graph, or the successor or predecessor version of a given one.

Considering the correspondences among versions at different levels of the hierarchy, there is a need to choose one ascendant or descendant version, but constraining the set of visited versions to that corresponding to the given one. The criteria **less_recent** and **more_recent** apply to a subset of versions in one level that correspond to a given version in another level of the inheritance hierarchy.

```
Example:
select vv.*
from Automobile a, versions(a) va, Vehicle v, versions(v) vv
where va.acessories='y' and
        vv is_ascendant_of va and
        vv is_less_recent;
```

This expression selects version *v3*, which is the least recent version among the ascendants of the version *mille* (with property accessories='y'), assuming that it was created before version *v4*.

7 CONCLUSIONS

Presently there are many kinds of applications for which the concept of version is considered essential. Common examples of those applications are Computer Aided Design and Manufacturing (CAD, CASE, CAM), Office Automation and Hyperdocuments.

In this paper the concept of version was discussed in the context of object-oriented database models and systems. The possibility of having versions at any level of an inheritance hierarchy was discussed, eliminating the need to concentrate all aspects related to the versioning of objects at the leaf level of the hierarchy. It was shown how the liberation of this constraint allows a more natural modeling of real world situations, specially those in which the objects are constructed in a top-down process. In this case, objects at higher levels of the hierarchy may be versioned before the creation of lower level objects, without the need of null values or similar constructions.

On the other hand, versions appearing at different levels of a hierarchy introduce the idea of mappings between versions. Those mappings allow a more natural and concise representation of the alternatives for object configuration. Configurations are obtained by the choice of the most adequate version at each level, without the need to explicitly represent the entire set of combinations permitted, as in the models where versions are allowed only at the leaf level of the hierarchies. In this context, where the (often very large) set of all possible combinations of versions is implicitly represented in the inheritance hierarchies, the notion of object configuration is very important, which denotes a completely resolved object instance, having only static references to its components. Configurations correspond to the concept of instances in a database without the version concept.

The approach proposed in this paper treats configurations as versions, improving the orthogonality of the model and introducing a very useful possibility of combining versions and configurations to build other versions, as well as deriving new versions from configurations. This approach allows the use of configurations representing the final result of a design as a component in other composed objects of higher level. Modularity can be increased and reusability of database objects is provided. This uniform treatment also results in a more concise model, since the set of operations defined for handling versions is the same used for configurations.

References

[1] Agrawal, R.; Buroff, S.; Gehani, N.; Shasha, D. Object Versioning in Ode. In: *Proc. Data Engineering*, 1991, Kobe, Japan. p. 446-455.

[2] Batory, D.S.; Kim, W. Modeling concepts for VLSI CAD objects. **ACM Transactions on Database Systems**, v.10, n.3, p.322-346, Sept. 1985.

[3] Beech, D.; Mahbod, B. Generalized Version Control in an Object-Oriented Database. In: *Proc. Data Engineering*, 1988, Los Angeles-EUA. p.14-22.

[4] Belkhatir, N.; Estublier, J. Experience with a database of programs. **Sigplan Notices**, v.22, n.1, p.84-91, Jan 1987.

[5] Bertino, Elisa; Martino, Lorenzo. **Object-Oriented Database Systems: Concepts and Architectures**. Addison-Wesley Publishers Ltd., 1993.

[6] Berkel, T. Et Al. Modelling CAD-objects by abstraction. In: *Proc. Int. Conf. on Data And Knowledge Bases*, 1988, Jerusalem, Israel. p. 227-240.

[7] Biliris, A. Modeling design object relationships in PEGASUS. In: *Proc. Data Engineering*, 1990. Los Angeles-USA. p. 228-236.

[8] Björnerstedt, A. ; Hultén, C. Version Control in an Object-Oriented Architecture. In: Kim, W.; Lochovsky, F.H. (eds.). **Object-Oriented Concepts, Databases, and Applications**. ACM Press, chap. 18, p. 451-485, 1989.

[9] Blanken, H. Implementing version support for complex objects. **Data & Knowledge Engineering** , v.6, p. 1-25, 1991.

[10] Breitl, R. The Gemstone data management system. In: Kim, W.; Lochovsky, F.H. (eds.). **Object-Oriented Concepts, Databases, and Applications**. ACM Press, p. 283-308, 1989.

[11] Byeon, K.J.; McLeod, D. Towards the unification of views and versions for object databases. In: *Proc. Int. Symp. on Object Technologies for Advance Software*, Nov. 1993, Konazawa-Japan.

[12] Carey, Michael J.; Dewitt, David J.; Vandenberg, S. A data model and query language for EXODUS. In: *Proc. ACM SIGMOD Conference*, 1988, Chicago.

[13] Cellary, W.; Vossen, G.; Jomier, G. **Multiversion object constellations: a new approach to support a designer's database work**. Giessen, Universität Giessen, Nov. 1991. (Bericht Nr. 9105)

[14] Delobel, Claude; Lecluse, Christophe; Richard, Philippe. **LOOQ: a query language for object-oriented databases, informal presentation**. Rapport Technique Altaïr 22-88, 8 Oct. 1988

[15] Deux Et al. The story of O_2. **IEEE Transactions on Knowledge and Data Engineering**, v.2, n.1, p.91-108, Mar. 1990.

[16] Dittrich, K.R.; Gotthard, W.; Lockemann, P.C. DAMOKLES-a database system for software engineering environments. In: *Proc. Int. Workshop on Advanced Programming Environments*, June 1986, Trondheim-Norway. p. 353-371.

[17] Dittrich, K.; Lorie, R. Version support for engineering database systems. **IEEE Transactions on Software Engineering**, v.14, n.4, p. 429-437, Apr. 1988.

[18] Edelweiss, N.; Oliveira, J.P.M. De; Pernici, B. An Object-Oriented Temporal Model. In: *Proc. CAISE'93*, Paris, 1993. p.397-415.

[19] Greenspan, S.J.; Borgida, A.; Mylopoulos, J. A Requirements modeling language and its logic. In: M.L. Brodie; J. Mylopoulos (eds.) **On Knowledge Base Systems**. Springer-Verlag, 1986. p.471-502.

[20] Haskin, R.L.; Lorie, R.A. On extending the functions of a relational database system. In: *Proc.ACM SIGMOD Conference*, 1982, Orlando. p.207-212.

[21] Hudson, S.E.; King, R. Object-oriented database support for software environments. In: *Proc. ACM SIGMOD Conference*, 1987, San Francisco, CA. p.491-503.

[22] Hudson, S.E.; King, R. The Cactis project: database support for software environments. **IEEE Transactions on Software Engineering**, v.14, n.6, p. 709-719, June 1988.

[23] Katz, R.; Chang, E.; Bhateja, R. Version modelling concepts for computer aided databases. In: *Proc. ACM SIGMOD Conference*, 1986, Washington. p.379-386.

[24] Katz, R.H. Toward a unified framework for version modeling in engineering databases. **ACM Computing Surveys**, v.22, n.4 , p. 375-408, Dec. 1990.

[25] Kim, W.; Banerjee, J.; Chou, H.T.; Garza, J.F. ; Woelk, D. Composite object support in an object-oriented database system. In: *Proc. Object-Oriented Programming Systems and Languages (OOPSLA)*, 1987. p. 118-125.

[26] Kim, W.; Bertino, E.; Garza, J.F. Composite objects revisited. In: *Proc. ACM SIGMOD Conference*, 1989, Oregon. p.337-347.

[27] Kim, W. Et Al. Features of the ORION Object-Oriented Database System. In: Kim, W.; Lochovsky, F.H. (eds.). **Object-Oriented Concepts, Databases, and Applications**. ACM Press, chap. 11, p. 251-282, 1989.

[28] Klahold, P.; Schlageter, G.; Wilkes, W. A general model for version management in databases. In: *Proc. VLDB*, 1986, Kyoto, Japan. p. 319-327.

[29] Lamb, Charles W.; Landis, Gordon; Orenstein, Jack A.; Weinreb, Daniel L. ObjectStore. **Communications of the ACM**, v.34, n.10, p.51-63, Oct. 1991.

[30] Lorie, R.L.; Plouffe, W. Complex objects and their use in design transactions. In: *Proc. ACM SIGMOD Conference*, 1983, San Jose, Calif. p.115-122.

[31] Mahler, A.; Lampen, A. An integrated toolset for engineering software configurations. In: *Proc. ACM-SIGSOFT-SIGPLAN Symposium on Practical Software Development Environments*, Boston, MA, 1988.

[32] McLeod, D.; Narayanaswamy, K.; Bapa Rao, K. An approach to information management for CAD/VLSI applications. In: *Proc. ACM Conference on Databases for Engineering Applications*, May 1983, San Jose, CA. p.39-50.

[33] Monk, S.R.; Sommerville, I. A model for versioning of classes in object- oriented databases. In: *Proc. BNCOD*, July 1992, Aberdeen, Scotland.

[34] Rowe, L.A.; Stonebraker, M.R. The POSTGRES data model. In: *Proc. VLDB*, Sept. 1987, Brighton, Engl. p. 83-96.

[35] Sciore, E. Using annotations to support multiple kinds of versioning in an object-oriented database system. **ACM Transactions on Database Systems**, v.16, n.3, p.417-438, Sept. 1991.

[36] Sciore, E. Versioning and configuration management in an object-oriented data model. **VLDB Journal**, v.3, p. 77-106, Jan. 1994.

[37] Snodgrass, R.; Ahn, I. A taxonomy of time in databases. In: *Proc. ACM SIGMOD Conference*, May 1985, Austin.

[38] Talens, G.; Oussalah, C.; Colinas, M.F. Versions of simple and composite objects. In: *Proc. VLDB*, Sept. 1993, Dublin, Ireland. p. 62-72.

[39] Wuu, G.T.D. ; Dayal, U. A uniform model for temporal and versioned object-oriented databases. In: **Temporal Databases: Theory, design and implementation**. Benjamin/Cummings, 1993. p. 230-247.

[40] Zdonik, S. Object-Oriented Type Evolution. In: Bancilhon, F; Buneman, P. (Eds). **Advances in Database Programming Languages**. Addison-Wesley, p.277-288, 1990.

Este artigo recebeu a Menção Honrosa no Prêmio COMPAQ de Estímulo a Pesquisa e Desenvolvimento em Informática, categoria Pesquisa Científica. Foi originalmente publicado nos Proceedings of the Seventh International Conference on Management of Data, Pune, India, dezembro de 1995.