Armazenamento da Informação SBU
Banco: Dados orientados:
Objetos
modelagem: Dados

Modeling an engineering design application using extended object-oriented concepts

Versões: Banco: Dados
CNPq 1.03.03.00-6

Lia Goldstein Golendziner

Clesio Saraiva dos Santos*

Flávio Rech Wagner*

Abstract

This paper presents an approach to extend object-oriented data models, in which versions of an object are allowed to appear at different levels of an inheritance hierarchy, in contrast to the known approaches where they are admitted only at one level. This approach allows the design and instantiation of objects to become very natural, starting with the design of an object in a class and refining it, adding properties to the subclasses. Versions of objects can be defined in a subclass, having ascendant versions/objects associated to the superclasses. The paper also discusses how the extended model can be used to model engineering applications, fulfilling their requirements. The application is the STAR framework, which implements an innovative and flexible data model that allows the user to define an object schema for each design object. Design alternatives and views can be created during the design process and are represented in the object schema. Versioning appears in the STAR model not only for the real design data, but also for alternatives and views in the object schema. This requirement is not naturally modeled by the existing version models in object-oriented databases.

Keywords object-oriented databases, engineering databases, data modeling, versions

1 Introduction

Research related to versions was 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.

Efforts in engineering applications focused mainly at the problems related to object representation [2,5, 11,12,14,20,21], and are based in semantic data models such as the Entity-Relationship Model (the most frequently used one). Katz [16] argued that many proposals presented in the area of engineering applications were similar, and proposed a basic terminology together with a collection of mechanisms that must be present in any approach to represent this kind of application.

* (clesio , flavio)@ inf.ufrgs.br - Instituto de Informática-UFRGS, Caixa Postal 15064
CEP 91501-970 Porto Alegre-RS

Currently 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 applications [1,7,19,24]. 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 treated by the data model as any other object in the system.

On the other hand, few results have been reported on the use of these object-oriented models by engineering applications [15,22,23,28].

The STAR framework [27], which is being developed at UFRGS (Federal University of Rio Grande do Sul - Brazil), is an Electronic Design Automation Framework which implements an innovative and flexible data model, that allows the user to define, for each object type, a schema of design alternatives and views to be created during the design process [25]. During the development of this project it was observed that the STAR data model has some requirements not adequately satisfied by the existing object-oriented models. These requirements include representation of versions at more than one abstraction level in an inheritance hierarchy and support for definition and manipulation of configurations.

Some object-oriented database models allow versions, but only at the most specialized type/class in an inheritance hierarchy [1,3,18]. The possibility of having versions simultaneously 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 may be required to choose, from a possibly large set of options, the specific combination of versions that will be associated to the components of a complex object in each situation. Each combination of specific component versions of an object is called a configuration.

The version model introduced in [13] and used in this paper handles configurations as versions, introducing a very useful possibility of combining versions and configurations to construct other versions, as well as deriving new versions from configurations. Configurations are out of the scope of this paper and the adopted approach is described in [13].

This work focuses on version modeling at the application level, to support the requirements from engineering applications. A version model is presented, in which the versioning of objects at all levels of an inheritance hierarchy is allowed, not restricting versioning to only one level. It is shown how this extension to the object-oriented paradigm allows a more natural modeling of many real world situations, specially when the objects are constructed in a top-down process. A concrete application, the STAR framework, is presented, and its requirements in terms of versioning of objects are outlined.

The remainder of the paper is organized as follows. Section 2 presents the version model, highlighting its basic concepts and the possibility of having object and version hierarchies. Section 3 presents the data model of the STAR framework, and illustrates it through an application, that represents a design of a microprocessor named RISCO. Section 4 describes the mapping of the STAR data model to the version model presented in section 2, and the main conclusions of this work are presented in section 5.

2 The version model

2.1 Version and versioned object

A version is an instance of an object at a given point in 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 cannot 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 as well as non-versioned objects as instances.

An automobile under design can be considered as a versioned object, having several associated versions, which represent the different stages or alternatives considered throughout the design. Figure 1 illustrates this example, where the several alternatives of an automobile are shown. The notation used is based on that introduced in [17].

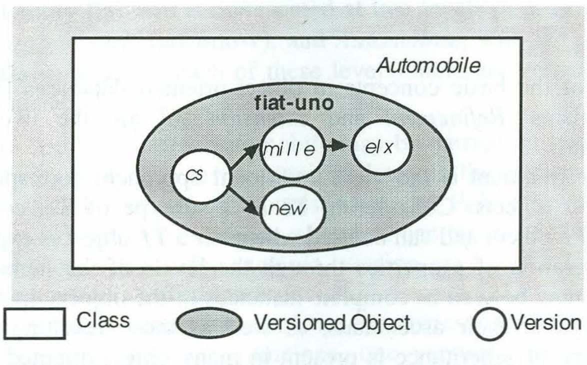

Figure 1 Versioned object and its versions

Each versioned object has one version considered as its *current version.* The current version is automatically maintained by the system as the most recently created one. If the designer wants a different version to be considered as the current one, he can specify it, but in this case the current version will remain fixed and will not change automatically with the creation of new versions (as in [18]). The current version is used whenever an operation is applied to a versioned object and does not specify one of its versions.

2.2 Version properties

Versions of a versioned object are related through a derivation relationship, which forms 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.

Versions have a status, that can be *working*, *stable*, or *consolidated* (similar to the classification in [3,18]), reflecting their robustness. 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 cannot 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 the *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 Object and version hierarchies

This section presents the proposal of object versioning at different levels of an inheritance hierarchy. The advantages of this approach in modeling applications are discussed and compared to the traditional approach where versions are restricted to one level of the hierarchy.

Inheritance

Inheritance is one of the basic concepts in object-oriented databases [4] and one of the reusability mechanisms. *Refinement* and *extension* [6] 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. Considering *T2* as a subtype of *T1*, each *T2*-object is a "special case" of a *T1*-object and can be used whenever a *T1*-object is expected [24]. 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 those resulting from the conflict resolutions. This type of inheritance is present in many object-oriented database systems, such as O_2 [10], ORION [19] and GemStone [8].

Extension inheritance is related to the idea of prototypes and appears in data models such as PEGASUS [6,24] (extension of EXTRA [9]). Each *T2*-object has an associated *T1*-object (called *prototype* in [6] and herein called *ascendant*). In this case, each property refers to a specific level of the hierarchy, modeling some relevant aspect of the real world object. References to attributes that are not defined in T2 are *delegated* to its associated T1-object.

Object and version mapping

When refinement inheritance is used, objects and their corresponding versions appear at only one level of the hierarchy [3,7,18]. In the version model presented in this paper, where extension inheritance is used, versions of each object are allowed at all levels. In this way, object modeling and instantiation can be done at various levels of abstraction, and the definition or redefinition of object properties can be done at any level.

Lets assume the schema presented in Figure 2, which represents an inheritance hierarchy: *Vehicle* is a superclass and *Automobile* and *Truck* are subclasses. The example in Figure 3 shows the modeling of versions at more than one level of this inheritance hierarchy.

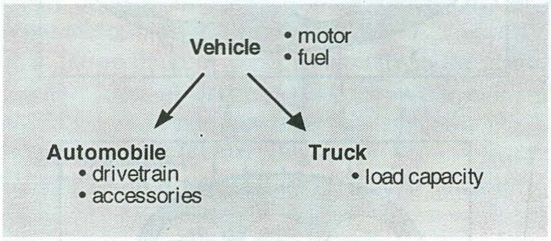

Figure 2 Example schema

The real world entity **fiat-uno** is represented at two levels of abstraction: *Vehicle*, with the properties *motor* and *fuel* (**fiat-uno-v**), and *Automobile*, with the properties *drivetrain* and *accessories* (**fiat-uno-a**). At each of these levels, there are versions associated to the corresponding versioned objects.

In this way, 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 may be designed starting with its characteristics at the *Vehicle* level, thus creating the versions at this level (for example, versions *v1* and *v2* can be created). In a further step, the versions at the *Automobile* level (for example, version *cs*) may be created and linked to their ascendants (version *cs* is linked to *v1* and *v2*).

Each version at a subclass must have at least one corresponding ascendant, to which it is linked at creation time. In some situations, one version may have more than one ascendant. This flexibility is important, allowing the designer to concentrate at one level at a time during the design process, as well as to determine all the possible combinations of one version with versions at higher levels of the inheritance hierarchy.

In the example, version *cs* of **fiat-uno-a**, at the *Automobile* level, can be combined with two different versions of **fiat-uno-v** (versions *v1* and *v2*), at the *Vehicle* level. These combinations represent two possible configurations for a **fiat-uno**: version *cs* having *v1* as ascendant (using gas as fuel), and version *cs* having *v2* as ascendant (using alcohol as fuel).

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 3, where one version at the *Vehicle* level (ex: *v3*) corresponds to two versions at the *Automobile* level (ex: *mille* and *elx*).

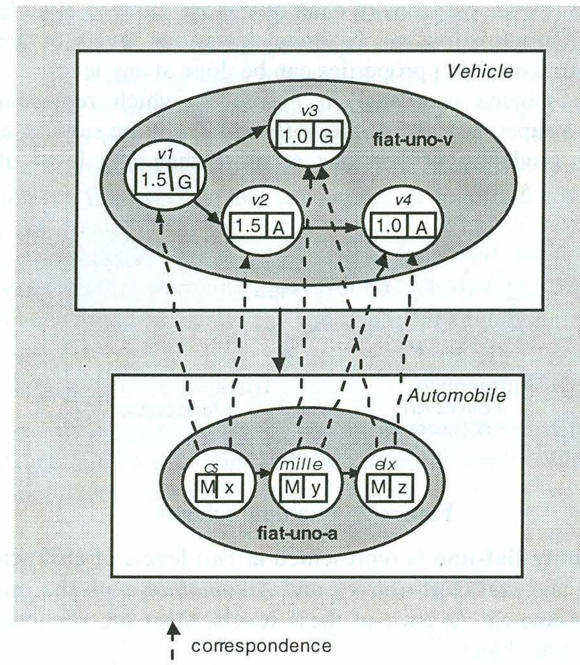

Figure 3 Versions represented at more than one level of the inheritance hierarchy and their correspondences

In this way, *correspondences (mappings)* are defined between versions of an object at one level and versions of their corresponding ascendants in the superclass. In Figure 3, 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 among versions may be *n:m*, as in Figure 3, *1:1, 1:n*, or *n:1*.

When one needs to get an object including properties from 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 criterion will be used when more than one ascendant version is linked to the desired version or object.

Versioned and non-versioned objects may exist in the same inheritance hierarchy. Non-versioned objects and versioned objects that have no versions are considered as a version, when the mapping constraint must be verified.

Without the possibility of representing versions at various levels of the inheritance hierarchy, other features of a data model could be used, but they do not result in adequate models in many situations. These alternative features are analyzed below.

Considering the **fiat-uno** example, 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 in general is not allowed. The reason commonly pointed out is the need to redefine the OID of the migrating object (because the class is part of the OID [19]), as well as the OID of the possible existing 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 will be 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 (because the model establishes that, when a new version is derived, its ancestors may not be changed anymore).

In addition, 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 4 presents a possible representation of the situation modeled in Figure 3, but with versions appearing at only one level.

Operations defined for objects and versions

The operations defined for objects and versions can be classified into three groups: operations for creation, operations for navigation in the inheritance hierarchy, and operations for retrieval of versions/objects.

The operators concerning creation of versions are the following:

create_versioned_object (class): OID;
derive_version (set(OID)
    [, **ascendant**: [class1:] set(OID)...]
    [, **descendant**: [class2:] set(OID)...]): OID;

Version creation can occur in one of the following 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 proceed, using the versioned object OID as a parameter;

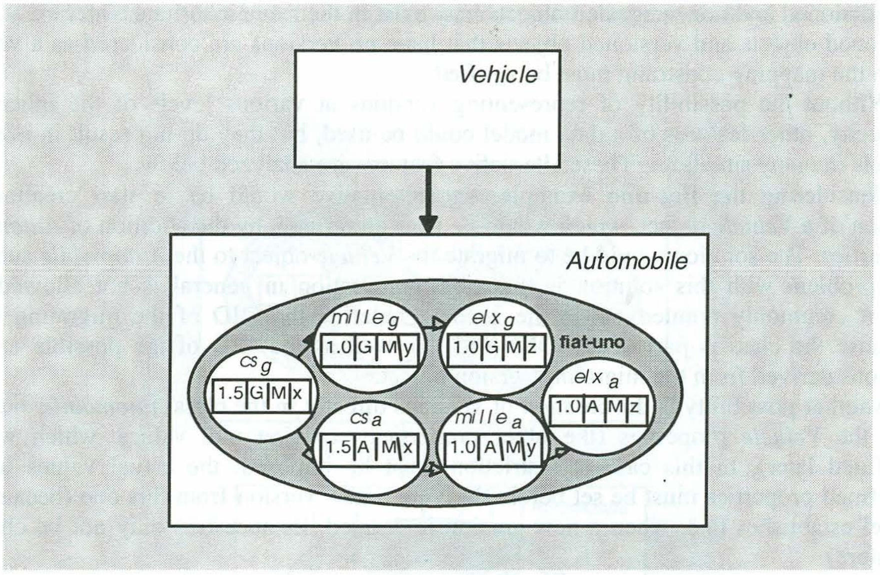

Figure 4 Versions only at the most specialized class

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.

When a version is created, it must be necessarily connected to an ascendant version/object. Optionally, descendants can be informed.

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, either 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 following pre-defined criteria is given: **recent**, **first**, or **current** (recent is used as default). The **get_descendant** operation is similar to the **get_ascendant** one, but it is applied to descendant objects in the identified subclass.

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 the values of those attributes defined in the class to which the object belongs. This operation returns the attributes that are defined at a single level of the inheritance hierarchy. To retrieve all the attributes of a real world entity modeled in the database, navigation must be performed along the hierarchy, so that all objects representing the given entity are retrieved. The operation **get_complete_object** was defined with this purpose and 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, exactly as in the operations **get_ascendant** and **get_descendant**.

Besides these operations, operations for navigation in the version derivation graph are also provided (**get_first_version**, **get_last_version**, **get_successor**, **get_predecessor**, **get_versioned_object**).

It must be noted that all the operations presented in this section apply also to versioned objects, in which case the current version will be considered.

3 The data model of the STAR framework

In the following sections the concepts of the STAR framework that are relevant to this work will be described. An example of an application of STAR is given to illustrate these concepts. The mapping from the STAR concepts to the version model is presented in Section 4.

3.1 Object types

A real world entity being designed in the STAR framework is represented by a *Design* object (Figure 5). Each *Design* object gathers an arbitrary number of *ViewGroups* and *Views*[1]. *ViewGroups* may in turn gather, according to application-defined criteria, any number of other *ViewGroups* and *Views*, building a tree-like hierarchical *object schema*. Three types of *Views* are supported: *HDL Views*, for behavioral descriptions, *MHD Views* (Modular Hierarchical Description), for structural descriptions, and *Layout Views*, for geometric descriptions. In all *View* types, objects can be described as a composition of sub-objects that are instances of other objects.

ViewGroups can be used, for instance, to build a hierarchy of design decisions, where alternatives for a given design state are appended to the *ViewGroup* that corresponds to this state and represented by another *ViewGroup* or *View*. The advantages and generality of this data model are stressed elsewhere [25,27].

[1] The concept of *View* is not that of the database field, but corresponds to the description of a design object at a given abstraction level (electrical, gate-level, etc.).

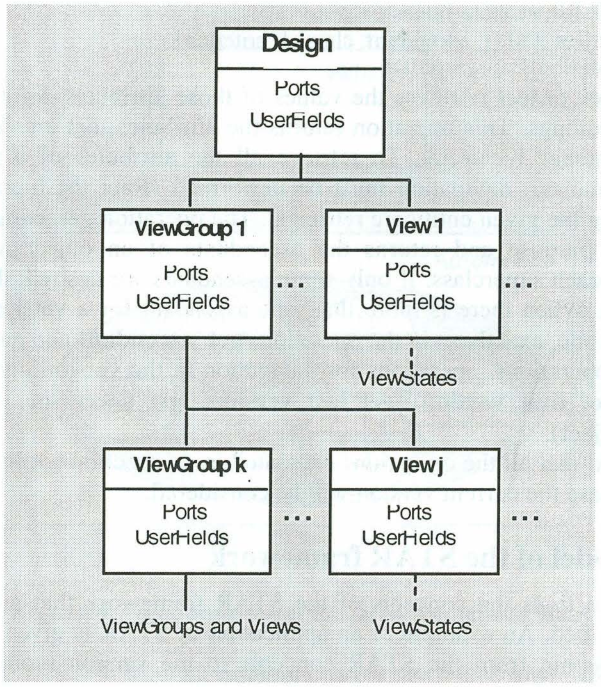

Figure 5 The STAR data model

The hierarchy formed by *Design*, *ViewGroup* and *View* nodes defines properties of the entity being designed and is an inheritance hierarchy. Each node has properties that may be inherited by its descendant nodes (extension inheritance). Not only the existence of an attribute is inherited by the descendant nodes, but also its value, when defined. The role of *Design*, *ViewGroups* and *Views* is to organize the various representations of a design object, ensuring consistency for common properties, through the inheritance mechanism. Thus, each node contains properties that are shared by the representations it gathers.

ViewStates contain the real design data that correspond to the various design representations, such as layouts, HDL (Hardware Description Language) descriptions and structural decompositions. *ViewStates* correspond to revisions created for each *View*.

Properties of each node of the object schema can be of one of three types: *Port*, *UserField,* and *Parameter*. *Ports* represent interface signals and can also present properties. *UserFields* are user-defined attributes and have a name and a domain. *Parameters* allow the designer to build parameterized, generic objects.

An example of an application in the STAR framework is the design of the microprocessor RISCO, a 32-bit microprocessor developed at UFRGS. Figure 6 shows part of the object schema corresponding to the design of this microprocessor. A more complete description can be found in [26].

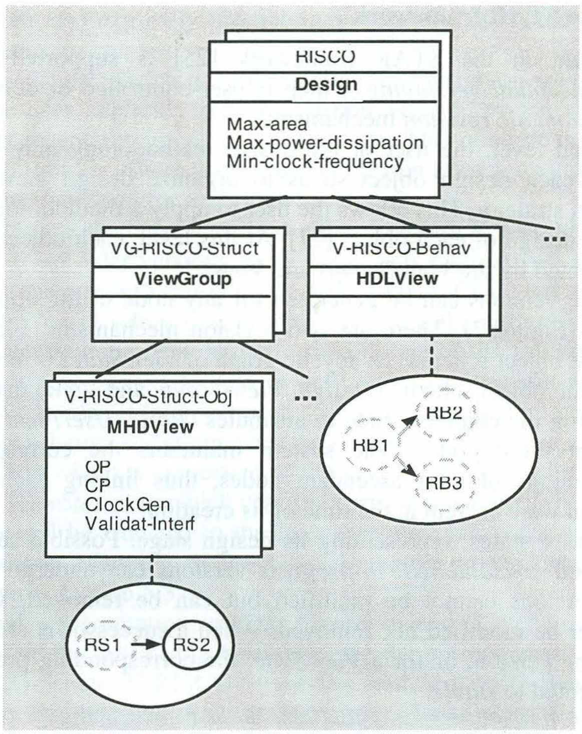

Figure 6 Object schema for the microprocessor RISCO

The methodology for the microprocessor behavioral design specifies the RISCO *Design* object and its initial object schema, including *Views* that contain behavioral related information. V-RISCO-Behav, of type HDL, is one of these *Views*, corresponding to an initial behavioral specification of the microprocessor written in some hardware description language. As a result of the design process, *ViewStates* (RB1, RB2 and RB3) are created for this *View*. The RISCO object has three initial attributes, corresponding to requirements set by the design manager: maximum area (**Max-area**), maximum power dissipation (**Max-power-dissipation**), and minimum clock frequency (**Min-clock-frequency**). These attributes, with their corresponding values, are defined at the root of the object schema.

A structural representation is manually generated from the behavioral one. The circuit is partitioned into its four main structural blocks: the operational block **OP**, the control block **CP**, the clock generator **ClockGen**, and the validation interface **ValidatInterf**. A *ViewGroup* VG-RISCO-Struct is added to the object schema, gathering all *Views* of the RISCO object that correspond to the structural design. One of these *Views* is V-RISCO-Struct-Obj, of type MHD, and contains references to four other *Designs*: OP, CP, ClockGen, and ValidatInterf. Associated to this *View*, two *ViewStates* were created: RS1 and RS2.

3.2 Versions in the STAR framework

Version management in the STAR framework [25] is supported by two different mechanisms: a) *conceptual versioning*, which is user-controlled or defined by the design methodology; b) *automatic revision* mechanism.

At the conceptual level, the user or the design methodology may define a particular object schema for each design object so as to organize design views and alternatives according to a given strategy. This allows the user to apply a methodology control which is highly tuned to the design of each object [27]. At this level, each object has a few number of versions, represented by the *ViewGroups* and *Views*.

At a lower level, versions can be generated for any node of the object schema, during the design activity (Figure 7). There are two revision mechanisms. Firstly, to each *View* (i.e., each leaf of the object schema) an acyclic graph of *ViewStates* is associated. Secondly, the other nodes of the object schema (*Design*, *ViewGroup*, and *View*) may have a sequence of versions, reflecting the changes made to attributes (*Ports, UserFields,* and *Parameters*) that are defined at these nodes. The system maintains the correspondence between *ViewStates* and versions of their ascendant nodes, thus linking each *ViewState* to the ascendant nodes that were current at the time of its creation.

Each version has a status, representing its design stage. Possible status values are *in-progress*, *stable*, and *consolidated*. *In-progress* versions can undergo modifications and removal. *Stable* versions cannot be modified but can be removed, while *consolidated* versions can neither be modified nor removed. When a successor is created for a *Design*, *ViewGroup,* or *View* version, or for a *ViewState*, the corresponding predecessor(s) is(are) automatically promoted to *stable*.

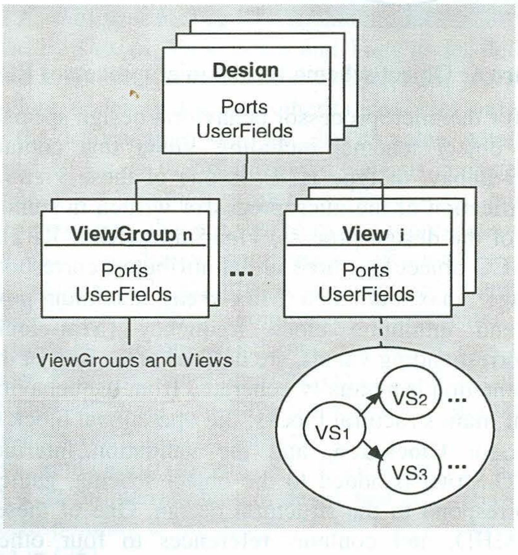

Figure 7 Versions in the STAR data model

The designer can also explicitly promote any version that should neither be modified nor removed anymore.

The concept of *current version* is used to define the version on which operations will be applied. Either the user explicitly sets the current version, or the system automatically maintains the most recent version as the current one. This concept is exactly the same as that described in section 2.1. There is one current version at each node of the object schema and also one current *ViewState* for each *View*.

The user may change the current version either to return to a previous design phase, or to create revisions from older versions, or to create new alternatives. A **select** operation is provided to change the current version to the selected one. Changing the current version at a given level of the object schema may imply changing the current versions of the other levels too. The select operation can be performed in one of the following ways: 1) partial selection - selects a new current version at only one level, without changing the current versions at the other levels; 2) total selection - selects a new current version at an arbitrary level *n* of the object schema, and propagates the selection to all upper levels, by selecting at each level the version which is associated to the chosen one at level *n*. Total selection can also be applied to *ViewStates*, in which case the current versions of the ascending nodes of the object schema will be changed to the ones that were current at the time the selected *ViewState* was created. Mapping of the **select** operation to the operations described in section 2.3 is shown in section 5.3.

In the graphical representation used in Figures 6 and 7, versions of nodes *Design*, *ViewGroup* and *View* that are in the front plan are the current ones. In Figures 6, nodes RISCO, VG-RISCO-Struct, V-RISCO-Struct-Obj and V-RISCO-Behav have associated versions, since it is assumed that their attribute values have been changed. The graphs of *ViewStates* are represented in their entirety, whereby the *ViewStates* that are not descendants of the current version of the corresponding *View* have a dashed contour. In the example, *ViewState* RS1 is not a descendant of the current version of *View* V-RISCO-Struct-Obj, but of its previous version. In the same way, *ViewStates* RB1, RB2 and RB3 were created when the first version of V-RISCO-Behav was the current one and, thus, are not descendants of the current version.

For each node of the object schema, values can be assigned to its attributes when the node is created or afterwards, when its descendants are created. For example, values can be assigned to attributes of the node *Design* RISCO when the node is created, or when a descendant *ViewState* is generated.

4 Mapping the STAR data model to the version model

4.1 Version mapping

When mapping the STAR data model to the version model, the two versioning modes of STAR are represented in a similar way.

Considering the *conceptual versioning*, each *ViewGroup* or *View* is represented by a versioned object, which is an instance of a defined class. ViewGroups and Views must be versioned objects, since each reference to any *ViewGroup* or *View* is always a generic reference to the node, leaving to the system the decision of selecting its current version. In the same way, the *Design* node is also represented by a versioned object.

To model the RISCO microprocessor, the classes Microprocessor, MP-structure, MP-behavior, and MP-V-structure were defined, building an inheritance hierarchy (Figure 8). Each of these classes has a versioned object as its single instance, representing the corresponding node of the schema object in STAR.

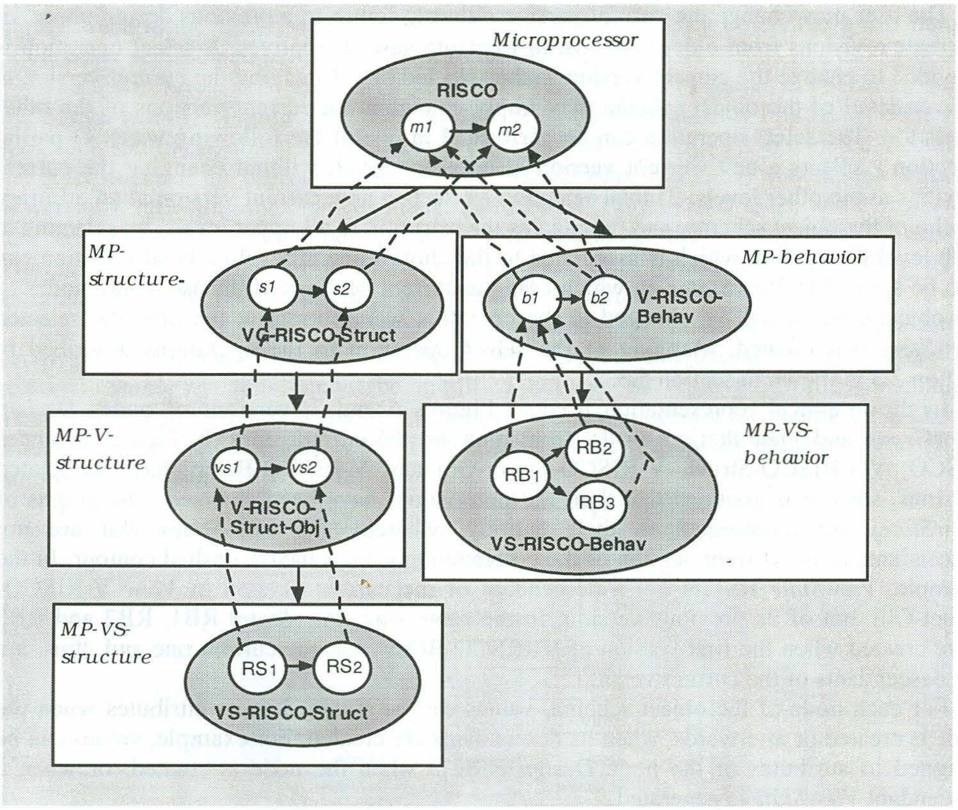

Figure 8 Representation of the RISCO schema object in the version model presented

Concerning *automatic versioning*, the two *revision* mechanisms are represented in two distinct ways. The graph of *ViewStates* associated to a *View* is represented by a versioned object which has the object representing the *View* as its ascendant object. In the example, two new classes MP-VS-structure and MP-VS-behavior were defined for these versioned objects. The versioned objects VS-RISCO-Behav and VS-RISCO-Struct represent the ViewState graphs and have, respectively, the objects V-RISCO-Behav and V-RISCO-Struct-Obj as ascendants, which correspond to the *Views* with the same name.

The second revision mechanism (versioning of nodes of the schema object) is implemented by versions that are created for the object representing the node. For example, versions of the RISCO Design node are represented by versions m1 and m2.

The concepts of current version and version status in the STAR framework are represented by similar concepts in the version model.

4.2 Correspondences among versions

When mapping the STAR data model, correspondences between versions in a given class and versions in its superclasses are always n:m. Several versions of an object in a class can correspond to the same version of an object in a superclass (for example, RB1, RB2 and RB3 in Figure 8 correspond to the same version b1). On the other hand, several versions of an object in a superclass can correspond to the same version of an object in a subclass (for example, m1 and m2 correspond to the version b2).

Since in the STAR model each node of a schema object can be linked to only one ascendant node, each class will have a single superclass (single inheritance).

In STAR, whenever a version of a node is created, it is connected to the current version of the ascendant node. In the version model, while creating a version, the user must indicate the ascendant object, and if this object has versions, the current one is considered. To get the correspondences as shown in Figure 8, the objects must have been created in the order shown in Table 1 (only the left subtree of the hierarchy is shown). Table 1 also shows the corresponding operations in the version model.

4.3 Partial and total selection

The mechanisms for version selection in STAR are supported in the version model in the following way:

1- Partial selection: the selection of a version or ViewState is implemented by changing the current version of a versioned object, through the **change_current** operation.

2- Total selection: this operation implies the modification of the current version at one node and at all its ascendant ones. In the version model, this operation is performed through a sequence of operations **change_current** and **get_ascendant**. **Change_current** modifies the current version at one node, and **get_ascendant** returns the corresponding ascendant. This ascendant version must be made the current one at its node. This sequence must be repeated until the root of the inheritance hierarchy is reached.

5 Conclusions

This paper presented a version model, in which one of the main characteristics is enabling definition of versions at various levels of inheritance hierarchies. It was shown how this feature 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, the objects at the higher levels of the hierarchy may be versioned before the creation of the lower level objects, without the use of null values or similar constructions.

Table 1 Mapping of operations of the STAR data model to the version model

Operations	
In the STAR data model	**In the version model presented**
creation of *Design* RISCO	• creation of class Microprocessor • creation of a versioned object (RISCO), instance of Microprocessor • creation of version m1
creation of *ViewGroup* VG-RISCO-Struct	• creation of class MP-structure • creation of a versioned object (VG-RISCO-Struct), instance of MP-structure, having object RISCO as ascendant • creation of version s1, having version m1 as ascendant (current version of ascendant)
creation of *View* V-RISCO-Struct-Obj	• creation of class MP-V-structure • creation of a versioned object (V-RISCO-Struct-Obj), instance of MP-V-structure, having object VG-RISCO-Struct as ascendant • creation of version vs1, having version s1 as ascendant (current version of ascendant)
creation of *ViewState* RS1	• creation of class MP-VS-structure • creation of a versioned object (VS-RISCO-Struct), instance of MP-VS-structure, having object V-RISCO-Struct-Obj as ascendant • creation of version RS1, having version vs1 as ascendant (current version of ascendant)
version creation for *Design* RISCO	• derivation of version m2, as successor of m1
version creation for *ViewGroup* VG-RISCO-Struct	• derivation of version s2, as successor of s1, having version m2 as ascendant
version creation for *View* V-RISCO-Struct-Obj	• derivation of version vs2, as successor of vs1, having version s2 as ascendant
creation of *ViewState* RS2	• derivation of version RS2, as successor of RS1, having version vs2 as ascendant

On the other hand, versions of the same object appearing at different levels of a hierarchy introduce the idea of version mapping. These mappings allow a more concise representation of the alternatives for object configuration [13]. Configurations are obtained by the choice of an adequate version at each level, without the need to explicitly represent the (often very large) entire set of permitted combinations, as in those models where versions of each object are allowed only at one level of the inheritance hierarchy.

It was shown how the presented version model supports the requirements from the STAR data model. The STAR framework is an engineering design application, which has non-conventional requirements in terms of representation of objects and versioning. The STAR data model, developed to support digital systems design, includes some features that are not naturally modeled by object-oriented models in which versions of each object are allowed at only one level of inheritance hierarchies. STAR versions can be created for any node of an object schema, requiring the modeling of correspondences among versions at different levels.

In this context, mappings among versions were used to model the relationship among the various nodes of an object schema in STAR

References

[1] Agrawal, R.; Buroff, S.; Gehani, N.; Shasha, D. Object versioning in Ode. In: *Int. Conf. on Data Engineering*, 7., 1991. 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: *Int. Conf. on Data Engineering*, 1988. p.14-22.

[4] Bertino, E.; Martino, L. *Object-Oriented Database Systems: Concepts and Architectures*. Addison-Wesley, 1993.

[5] Berkel, T. et al. Modelling CAD-objects by abstraction. In: *Int. Conf. on Data and Knowledge Bases*, 3., 1988. p. 227-240.

[6] Biliris, A. Modeling design object relationships in PEGASUS. In: *Int. Conf. on Data Engineering*, 6., 1990. p. 228-236.

[7] 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, p. 451-485, 1989.

[8] 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.

[9] Carey, M.J.; Dewitt, D.J.; Vandenberg, S. A data model and query language for EXODUS. In: *ACM SIGMOD Conf.*, 1988.

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

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

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

[13] Golendziner, L.G.; Santos, C.S. Versions and configurations in object-oriented database systems: a uniform treatment. In: *Int. Conf. on Management of Data (COMAD)*, 7., 1995. p.18-37.

[14] Hudson, S.E.; King, R. Object-oriented database support for software environments. In: *ACM SIGMOD Conf.*, 1987. p.491-503.

[15] Jones, M.C.; Rundensteiner, E.A. Extending view technology for complex integration tasks. In: *Int. IFIP Working Conf. on Electronic Design Automation Frameworks*, 4., 1994. p.76-85.

[16] 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.

[17] Kim, W.; Banerjee, J.; Chou, H.T.; Garza, J.F. ; Woelk, D. Composite object support in an object-oriented database system. In: *OOPSLA*, 1987. p. 118-125.

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

[19] Kim, W. et al. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, v.2, n.1, p.109-124, Mar. 1990.

[20] Klahold, P.; Schlageter, G.; Wilkes, W. A general model for version management in databases. In: *VLDB*, 12., 1986. p. 319-327.

[21] McLeod, D.; Narayanaswamy, K.; Bapa Rao, K. An approach to information management for CAD/VLSI applications. In: *ACM Conf. on Databases for Engineering Applications*, 1983. p.39-50.

[22] Miller, J.; Gröning, K.; Schulz, G.; White, C. The object-oriented integration methodology of the CADlab work station design environment. In: *ACM/IEEE Design Automation Conference*, 29., 1989.

[23] Ramakrishnan, R.; Ram, D. Janaki. Modeling design versions. In: *VLDB*, 22., 1996. p. 556-566.

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

[25] Wagner, F.R.; Golendziner, L.G.; Lacombe, J.; Lima, A.V. Design version management in the STAR framework. In: Newman, M.; Rhyne, T. (eds.). *3rd IFIP Workshop on Electronic Design Automation Frameworks, 1992*. p 85-97.

[26] Wagner, F.R. *Modeling the Design Methodology for the RISCO Microprocessor*. Research Report 174, 1992.

[27] Wagner, F.R.; Golendziner, L.G.; Fornari, M.R. A tightly coupled approach to design and data management. In: *EURO-DAC*, 1994. p.194-199.

[28] Wolf, Wayne. An object-oriented, procedural database for VLSI chip planning. In: *ACM/IEEE Design Automation Conference*, 23., 1986. p 744- 751.

Artigo originalmente publicado em: INTERNATIONAL CONFERENCE on DATABASE SYSTEMS for ADVANCED APPLICATIONS (DASFAA), 5., Melbourne, Australia, Apr. 1-4, 1997. Proceedings. Singapore: World Scientific, 1997. p.343-352.