Complex Object Versioning

Chabane Oussalah and Christelle Urtado

LGI2P / EMA - EERIE

Parc Scientifique Georges Besse - 30000 Nîmes - France

oussalah@eerie.fr, urtado@eerie.fr

Abstract. Engineering applications require semantically rich data to be modeled and managed over time. Complex objects, meaning objects linked to other objects by structural and existential dependences, allow such data to be represented. Complex object versions track the evolution of complex objects over time or during a design process. This paper describes a complex object version management model that combines two approaches for the management of complex object versions, depending on the granularity of the versioned objects: the macroscopic and the microscopic levels. These two levels give the various categories of users the possibility to choose the complex object version management technique that is best adapted to their needs. Unlike existing models, this model allows all types of complex objects, at both class level and instance level, to be managed. Furthermore, the microscopic level is highly parameterized and the macroscopic level offers an easy-to-use interface to users.

1 Introduction

Applications supporting a design or creation activity (such as computer-aided design frameworks, software engineering environments, multimedia composition environments, hypertext support environments, etc.) require that the database on top of which they are written possesses powerful techniques for the representation and management of data. Historically, these applications firstly expressed their need for the insertion into databases of structures allowing semantically rich data to be represented. This requirement is particularly well fulfilled by object-oriented databases integrating the complex object notion, that is, the notion of an object linked to exterior objects by relations, called dependence relations. These applications, because of their interactions with several types of users, are highly dynamic: the modification of either the structure and organization of the objects stored in the database or the objects themselves must be possible. This characteristic stresses the need for simultaneously keeping several states of data (instances) and of data structures (classes). This need for data evolution management techniques is widely considered to be fulfilled using the notion of version, whether it be by databases from research laboratories (Encore [12, 16], Orion [2], Version Server [4]) or from industry (ObjecStore [9], Versant [15]). The necessity to enhance database systems with complex object version management techniques emerges from these two requirements. The existing models propose two different approaches to solve this problem:

- approaches considering complex object versions as elementary object version graphs and providing automatic management techniques for such graphs (for example [4]),
- approaches considering complex object versions as versions of "black boxes" (or containers) which themselves contain elementary object versions and provide container version management techniques (for example [1]).

These two approaches happen to be complementary and solve different classes of problems.

The main contribution of this paper is the definition of a complex object version management model for object-oriented database applications, that adopts a mixed approach integrating an elementary object version graph management technique (the microscopic level) and a container version management mechanism (the macroscopic level). Depending on their needs and on those of the targeted application, the different types of users can either use one or the other approach. Technically, these two levels are based on the same notions (reified dependence relations, relation versions, propagation rules and strategies) and are particularly well integrated in our model.

The remainder of this paper is organized as follows. Section 2 describes the problematics related to the management of complex object versions both at the structural level and existential level and presents the two existing approaches for solving the identified problems. The following three sections introduce our model for the management of complex object versions. In section 3, the basis of our model, to be used both at the microscopic level and at the macroscopic level, is described. In sections 4 and 5 respectively, the microscopic and macroscopic levels for complex object version management are presented. Section 6 concludes this paper with perspectives for this work.

2 Problematics of Complex Object Version Management

Complex object version management extends elementary object version management (instance and class version management) for which we previously have proposed a model [13]. In this paper, the characteristics required for an elementary object version model [7] (elementary object version creation, version history, version state, storage, version id, etc.) are not detailed.

Complex object version management aims to solve problems which are typical of the complex object notion: the representation of complex object versions and the definition of operations to manage these complex object versions. In the following two subsections, both these structural and operational aspects of complex object version management will be described.

2.1 Structural Dependence: Coherence of a Set of Versions

The objects stored in a database are never independent from one another: they represent a coherent state of the modeled world. This structural dependence is

expressed by means of relations (such as inheritance, composition, association and equivalence). These objects, linked to other objects by relations which we will call dependence relations, are called complex objects. When several versions of the stored objects coexist in a same database, the database is responsible for presenting to users versions of objects that "go together" [1]. The database must therefore identify groups of object versions that represent a coherent state of the modeled world. Consequently, it is important that a complex object version management model provides structures allowing sets of coherent versions, meaning complex object version constituents, to be grouped together.

2.2 Existential Dependence: Creation or Destruction of a Complex Object Version

The elementary object versions constituting a given complex object version are, as we have just seen, dependent on one another. This dependence is not only structural, but also existential. In other words, the creation or destruction of a constituent object version can influence the creation or destruction of versions of other constituents of the same complex object. These operations can be carried out by users but this task can become too complex to be dealt with manually as soon as the objects become numerous. Consequently, it is important that operational mechanisms for the automatic creation and destruction of complex object versions be provided. As a conclusion, despite functionalities for elementary object version management, a complex object version management model must provide the user with:

- a representation model for sets of coherent elementary object versions,
- a technique for the creation and destruction of complex object versions.

2.3 Two Approaches for the Management of Complex Object Versions

Even though different terminologies such as schema versions, database versions or software configurations are used, a lot of models propose mechanisms for complex object version management. These are generally tailored to a given type of complex object and a given application. The existing models propose two distinct approaches for the representation and management of complex object versions. These approaches can be distinguished by the granularity of the versioned objects:

- *the macroscopic approach* versions high granularity objects (containers), that can, for example, be a configuration or a database, which themselves contain elementary object versions (class versions or instance versions),
- *the microscopic approach* versions only elementary objects (classes and/or instances) and manages graphs of such versions.

For each of these approaches, the user's needs are different. Let us suppose that the object considered by the user is a car. This car is described as a composite

instance: it is composed, among other components, of a body and an engine. The complex object car exists in two versions: a first version with a body painted in dark green and having a quoted value of 100,000 FF and a second version painted in a metallic color giving it a higher quoted value (105,000 FF). The user at the macroscopic level considers the complex object car as a box in which there is a version for the car (*My-car*), a version for the body (*My-carbody*) and a version for the engine (*My-carengine*), linked by composition relations (*the-body* and *the-engine*) as represented in the upper part of Figure 1. When a user works with a given version of the complex object car, the physical representation of the various versions of the car constituents is hidden from him. The user

Fig. 1. Two versions of the complex object car at the macroscopic level

at the microscopic level considers each version of the complex object car as a composition graph (cf. Figure 2). He/she can identify the derivation hierarchies of the various constituents of the complex object and, for example, see that a version of the engine is shared by several versions of the car.

2.4 Problematics Within the Framework of the Macroscopic Approach

The macroscopic approach consists in grouping together elementary entities (classes or instances) constituting the complex object in a high granularity object called *container*. This container is itself considered to be a versionable entity. The various versions of this container represent sets of coherent versions of its constituent entities. In the same way as we distinguish class versions from instance versions [13], we distinguish two types of containers:

Fig. 2. Two versions of the complex object car at the microscopic level

- *class containers* which group together classes and class versions. The models that propose class container versions are, like the models that propose class versions, not very numerous because only a few models are interested in storing both object structures (classes) and objects themselves (instances). They are used by Kim and Chou [5] in their schema version management model.
- *instance containers* which group together instances and instance versions. This category of container is used by the VBD model [1] for database versions and by the ObjectStore system [9] for configuration versions¹.

The two problems (structural and existential) that have been identified above can be stated more precisely within the framework of the macroscopic approach. The need for the complex object version management model to maintain the structural dependences between the constituents of complex objects, involves, in the case of the macroscopic approach, the following two sub-problems:

- *the physical description of a container.* In the VBD model [1], an association table allows to dynamically determine the instance version which belongs to a given database version. In Kim and Chou's model [5], each class version references its creator schema, through a static link (attribute), while a corresponding generic object references the schema that has suppressed a class. The system can therefore dynamically determine which class version belongs to a given schema.
- *references between elementary object versions.* In the VBD model, references are dynamic. If an object A is linked through a simple referencing relation to an object B, the instance version representing A in a database version is implicitly linked to the version representing B in this database version. In Kim and Chou's model, on the contrary, relations are static, encapsulated into classes, by means of attributes referencing superclasses and subclasses.

The existential dependence aspect of the complex object version management consists, in the case of the macroscopic approach, in defining the creation and

¹ For ObjectStore, a configuration is a group of instances that has to be treated as a versioning unit.

destruction operations for a container version. In the VBD model, the creation of database versions is carried out by means of a special transaction type. Therefore, the creation of a database version occurs at the user's initiative. On the opposite, in Kim and Chou's model, the addition of a superclass or of a subclass to a given class automatically triggers the creation of a version of this class in the concerned schema.

2.5 Problematics Within the Framework of the Microscopic Approach

In this approach, the versioned entities are the entities having the lowest granularity of the database. Complex objects are seen as graphs of these elementary entities the edges of which are dependence relations (inheritance, composition, association, etc.), and object versions are seen as graphs of these entities (classes or instances) and of versions of these entities (class versions or instance versions). These graphs interconnect sets of coherent elementary entity versions. Many models use this approach for different types of complex objects, meaning different types of dependence relations. Let us cite: Version Server [4], Encore [16] and Présage [13] for the composition relation, Encore [12] for the inheritance relation, Version Server [4] for the equivalence relation, and Orion [2] for the simple referencing relation².

The two problems (structural and existential) which have been identified above can be stated more precisely within the framework of the microscopic approach. In this approach, the problem of grouping sets of coherent elementary object versions consists in maintaining relations between elementary object versions (classes or instances) because complex object and complex object versions are seen as graphs. Most models support both static and dynamic referencing. Static referencing can involve the copying of the relation existing between objects or its versioning [4]. The creation or destruction of complex object versions is generally carried out using a version propagation mechanism that performs the automatic creation or destruction of a complex object version. This technique makes the user's task easier when object versions are too complex to be dealt with manually.

3 Basis of a Model Using a Mixed Approach

3.1 Objectives

Our point of view consists in providing the various types of users with a model for the management of complex object versions which is powerful, in that it applies to any type of application, expressive, in that it allows the user to define how complex object versions are managed, and easy to use, especially by non-expert users. With this perspective in mind, we choose to adopt an approach combining

² The simple referencing relation does not have any predefined semantics, in opposition with, for example, the composition relation.

the management of elementary object version graphs (microscopic level) and the management of container versions (macroscopic level). Therefore, users have at their disposal a range of possibilities which allow complex object versioning to be dealt with at two distinct granularity levels.

The macroscopic level allows non-expert users to consider complex object versions as "black boxes" which contain sets of coherent versions.

The microscopic level and its associated version propagation mechanism allows application designers to establish a real complex object version management policy that can adapt to the needs of an application. At the system level, these two approaches integrate very well because they are based on common concepts: the notions of dependence relation, relation version, propagation strategy and propagation rule.

Our complex object version model is based on an existing representation model for elementary object versions [13], extended to the representation of dependence relation versions, on top of which we have added three modules dedicated to complex object version management: a module describing propagation rules and strategies, which defines basic concepts, and two modules for complex object version management at the microscopic and macroscopic levels.

3.2 Dependence Relations, Propagation Rules and Strategies

In order to allow our model to be adapted to any type of complex object and even to the future addition of new types of relations, we choose to represent references between objects using objects: *dependence relations*. These relations allow semantics for inter-object references to be defined. In many models, the composition relation, for example, conveys strong semantics [6, 8]: a composition link can, for example, be exclusive or shared, dependent or independent, predominant or not. Furthermore, we have introduced the concepts of propagation rules and strategies that allow the operational capabilities of dependence relations to be described declaratively. *Propagation rules* define the actions that must be triggered on the objects which may be reached through the relation (that is, the extremities of the relation). A propagation rule describes, in a declarative manner, a version creation or destruction propagation mechanism associated with one or more dependence relation. *Propagation strategies* group together sets of rules that describe the version propagation capabilities of a dependence relation. The operations propagated are those of version creation and version destruction. A default version propagation strategy is associated with each dependence relation (class or instance). This strategy groups together the set of propagation rules which define the version creation and the version destruction propagation capabilities for this relation.

4 The Microscopic Approach

This approach, built upon the notions described in section 3, is based on the representation of complex objects and complex object versions by means of graphs

expressing the dependence between objects or object versions. From the operational point of view, the management of complex object versions is ensured by an operation propagation mechanism.

4.1 Complex Object Version Derivation

The derivation of a complex object version can be decomposed in terms of the following elementary transformations: addition, suppression or modification of an attribute of the object (which are operations usually defined for elementary object versions) or addition, suppression, or modification of a relation coming from or going to this object (which are operations inherent to the complex object notion). In our model, the modification of a dependence relation triggers the derivation of a relation version. The concepts of object and relation are therefore dealt with homogeneously.

4.2 A Parameterized Propagation Mechanism

We propose a version propagation model where the propagation itself is customizable. The way in which object versions which are related to a given object are propagated is not a structural property of the object: rather, it depends on the relation linking the objects. We therefore choose to attach the version propagation capabilities (expressed by the means of a propagation strategy) directly to the relations [11]. This strategy can be either reused by another relation (possibly through inheritance) or redefined. The system possesses predefined rules adapted to the most current inter-object relations. These rules allow the version creation and version destruction operations to be propagated in four directions and according to two modes. The propagation direction can be FORWARD, BACKWARD, BIDIRECTIONAL or NONE. FORWARD, for example, signifies that the propagation takes place from the source of the relation to its destination. The propagation mode can be RESTRICTED or EXTENDED. If it is RESTRICTED, the operation propagates from the extremity on which it is triggered to the relation object. If it is EXTENDED, the operation propagates from the extremity on which it is triggered to both the relation object and the other extremity of the relation.

The introduction of propagation rules allows a wide variety of version creation and version destruction propagation techniques to be used, from among which an application designer can choose the one that is best adapted to a given relation and a given application. Our version propagation model distinguishes version creation propagation from version destruction propagation. It also distinguishes class version propagation from instance version propagation.

The default propagation strategy associated with a relation for class version propagation can be redefined for a subclass of this relation. The default propagation strategy associated with a relation for instance version propagation is defined at class level and can be redefined in a subclass of the relation or even in a given relation instance. The redefinition of strategies is governed by the notion of admissible strategy [10, 14]. Propagation rules are active; they are written with the formalism of ECA rules [3].

4.3 An Example of Composite Object Version Management

Figure 3 proposes a description of the propagation rules and strategies that allow the example of the instance of car (represented in Figure 2) to be modeled. The creation of a version of the body with its new color triggers the application of rules R_2 from the strategy S_2 and R_3 from the strategy S_3 , the actions of which consist in the creation of a version of each composition relation (*the-body* and *the-engine*) and a version of *My-car* by increasing its quoted value. The destruction of the version *My-carbody*' triggers the destruction of *My-car*' and of the corresponding composition link *the-body*' (application of rule R_5 of strategy S_2) and of the composition link version *the-engine*' (application of rule R_6 inherited by S_3 from S_2).

Fig. 3. Propagation rules and strategies governing the composite object My-Car

5 The Macroscopic Approach

The macroscopic approach is a technique allowing the most frequently needed complex object version mechanisms to be expressed simply. In the design of our complex object version management model at the macroscopic level, we paid particular attention to the ease of use and the flexibility of the notion of container (which is not limited, like in most of the existing models, to a single dependence relation fixed since the design of the framework of the model). Furthermore, unlike most of the existing models, our model allows both class version containers and instance version containers to be represented.

5.1 Container Version Creation

A container is a high granularity object that groups together a set of elementary objects linked by dependence relations. The creation of a container is done by the user: it consists in enumerating all the elementary entities and relations of which it is constituted. A container can itself be constituted of another container: it is the notion of sub-container. This notion is similar to the notion of sub-configuration encountered, for example, in *ObjetStore* [9].

5.2 Derivation of a Container Version

This paragraph presents the container version derivation operation first from the user point of view and then from the system point of view.

User point of view:

The derivation of a container version respects the following principles:

- a unique version of a given object can be contained in a given container,
- containers are, in the same way as elementary object versions, organized in a derivation tree with the predecessor/successor notion,
- if an object version is not physically stored in a container version, it signifies that this object keeps the value it had in the predecessor container version,
- when a container version is created, it is by default identical to its predecessor container version.

The derivation relation between containers allows the changes that have been made between two successive container versions in the container version derivation hierarchy to be tracked. Deriving a container version consists in the addition, suppression or modification of one of its constituent objects. The modification of a constituent object signifies its derivation. Only this last operation is possible on instance containers (in order to respect the semantics of container instantiation). The addition, suppression or modification of a relation is not permitted in this approach because we consider that containers define complex objects as wholes which consist in elementary objects and relations. The internal organization of objects inside a container is a characteristic of the container itself which is defined, once for all, by the user. Dependence relations and dependence relation versions are fully managed by the system. Thus, the user's point of view over complex object versions is simplified. When deriving a container version, the user simply has to indicate which constituent objects (or object versions) have to be added, suppressed or derived.

System point of view:

Macroscopic object versioning is based on the concepts defined in section 3 which are already used by the microscopic approach. Our response to the structural dependence problem identified in section 2 is based on the existing notions of dependence relation and relation version. However, these mechanisms are only used at the system level and remain hidden from users, in order to simplify the

way in which macroscopic object versions are presented to them. The creation of a container version (*resp.* the destruction of a container version) is carried out by calling a version creation (*resp.* a version destruction) method which is distinct from the one defined for elementary object version management. A container is linked to its constituent objects by a system-defined relation (the *is-contained-in* relation), with which a propagation strategy is associated in the same way as in the microscopic level. This strategy is hidden from users. It is composed of a single creation rule which is automatically generated by the system, given the list of the objects to add, suppress or derive, and of a single destruction rule. These system-managed strategies and rules cannot be modified by users: the system assumes the responsibility of the coherence maintenance of containers. The execution of the container version creation rule or container version destruction rule follows the same principle as the one described at the microscopic level. However, the propagation rules executed here never trigger version creation recursively: they are written exclusively with terminal derivation operations that do not raise any event (except for the case of sub-containers).

5.3 An Example of Composite Object Version Management

As in the previous paragraph, we present through an example, first the user point of view then the system point of view.

User point of view:

Let us consider again the car example represented in Figure 1. The description of the instance container *Car-container* that contains the design of the car consists in enumerating all the contained instances (*My-car*, *My-carbody* and *My-carengine*) and all the contained relations (*the-body* and *the-engine*), which are, in this example, composition relations. To indicate what modifications must be made between the container *Car-container* and its first version, the user must specify that the creation of a version of the container *Car-container* is going to modify *My-carbody* and *My-car*. With this information, the system is able to automatically create a new container version for the car design in which the user will be able to modify the attribute values for the instances of *My-car* and of *My-carbody*. Figure 1 shows the new container version *Car-container'* after the modifications have been made by the user. The shaded parts of the figure indicate the modified elements.

System point of view:

The upper part of Figure 4 shows the container *Car-container* as it is represented by the system. The relation *is-contained-in* has been automatically introduced and allows the constituents of the container to be identified. This relation supports the version propagation strategies that allow the versions of *My-carbody*, *My-car*, *the-body* and *the-engine* to be created. These strategies are automatically generated by the system (*cf.* Figure 5). Their version creation propagation rule is generated as follows:

Fig. 4. System representation for the Car-container and its version Car-container'

- the rule propagates version creation in the FORWARD direction and with the EXTENDED mode for the relation *is-contained-in* referencing each object listed by the user. In the example, the objects concerned are *My-car* and *My-carbody*.
- the rule propagates version creation in the FORWARD direction and with the EXTENDED mode for the relation *is-contained-in* referencing each relation having an extremity among the objects listed by the user. In the example, the relations concerned are *the-body* and *the-engine*.
- the rule propagates version creation in the FORWARD direction and with the EXTENDED mode for the relation *is-contained-in* referencing each sub-container listed by the user. The rule is, in this case, responsible for the recursive propagation of version creation on these containers.
- the rule propagates version creation in the NONE direction for the relation *is-contained-in* referencing all other objects and relations. In the example, the object concerned is *My-carengine*.

Strategy S_1 governs version creation propagation for all instances of the *is-contained-in* relation. Strategy S_0 redefines the version propagation capabilities for the instance relation *is-contained-in-5*. This redefinition allows *My-carengine* not to be derived (rule R_0). The version *Car-container'* of the container obtained after derivation is represented in the lower part of Figure 4. The objects or relations that have been derived are shaded: for example, *My-carbody'* derives from *My-carbody*.

Fig. 5. Propagation rules and strategies generated by the system for the Car-container

6 Conclusion and Perspectives for this Work

In this article, we have proposed a mixed model for the management of evolution in object bases using versioning. This model manages complex object versions at two levels: the macroscopic and microscopic levels. It therefore distinguishes itself from the existing models that manage either elementary entity versions or versions of entities of high granularity.

In designing this model, particular attention was paid to its extensibility and, especially, to the management of complex object versions of any type. This is why both levels of our model are based on the concept of dependence relation between objects, relations for which the users can define new types, and the concepts of propagation rules and strategies which constitute a declarative support for the description of the operational capabilities of the dependence relations.

Perspectives for this work are numerous. The validation of our model on top of the database Versant [15] in its C++ version is in progress. We especially focus on the definition of an easy-to-use graphic interface for users of both levels of our model. We also plan to reinforce the links between class versions and instance versions by allowing the application designer to define versioning constraints on classes or class containers which are going to condition the versioning of instances or instance containers. Such constraints would allow quality criteria for end-user versions to be expressed.

References

1. Cellary, W., Jomier, G.: Consistency of versions in object-oriented databases. In Proceedings of the 16th conference on Very Large Data Bases, pages 432-441, Brisbane, Australia, 1990.
2. Chou, H. T., Kim, W.: Versions and change notification in an object-oriented database system. In Proceedings of the 25th ACM/IEEE Design Automaton Conference, pages 275-281, Anaheim, June 1988.
3. Dayal, U., Buchman, A. P., McCarthy, D. R.: Rules are objects too : a knowledge model for an active, object-oriented database system. In Proceedings of the 2nd workshop on object-oriented database systems, pages 129-143, West Germany, September 1988.
4. Katz, R. H.: Toward a unified framework for version modeling in engineering databases. ACM Computing surveys, 22(4):375-408, December 1990.
5. Kim, W., Chou, H. T.: Versions of schema for object-oriented databases. In Proceedings of the 14th conference on Very Large Data Bases, pages 148-160, Los Angeles, California, 1988.
6. Kim, W., Bertino E., Garza, J. F.: Composite objects revisited. In Proceedings of the ACM SIGMOD conference on Management of Data, Portland, Oregon, June 1989.
7. Loomis, M. E. S.: Object versioning. In Journal of Object-Oriented Programming, January 1992.
8. Magnan, M.: Reusability of components : exceptions in composite objects. PhD Thesis, University of Montpellier II, Montpellier, France, September 1994. (in french).
9. Object Design inc.: Version management. In ObjectStore release 3 for UNIX systems User Guide, chapter 8, pages 208-247, December 1993.
10. Oussalah, C., Urtado, C.: Adding semantics for version propagation in OODBs. In Proceedings of the workshop on Databases and Expert Systems Applications, IEEE Computer Society Press, Zurich, Switzerland, September 1996.
11. Rumbaugh, J., Controlling propagation of operations using attributes on relations. In Proceedings of the conference on Object-Oriented Programming Systems, Languages and Applications, pages 285-296, September 1988.
12. Skarra, A. H., Zdonik S. B.: The management of changing types in an object-oriented database. In Proceedings of the conference on Object-Oriented Programming Systems, Languages and Applications, pages 483-495, September 1986.
13. Talens, G., Oussalah, C., Colinas, M. F.: Versions of simple and composite objects. In Proceedings of the 19th conference on Very Large Data Bases, Dublin, Ireland, August 1993.
14. Urtado, C., Oussalah, C.: Semantic rules to propagate versions in object-oriented databases. In Proceedings of the 3rd workshop on Advances in Databases and Information Systems, Moscow, Russia, September 1996.
15. Versant OBMS: Versant concepts and usage manual. Release 4.0, Chapter 7, July 1995.
16. Zdonik, S. B.: Object management system concepts. In Proceedings of the conference on Office Information Systems. ACM SIGOA, pages 13-19, Toronto, Canada, June 1984.