Consistency of Versions in Object-Oriented Databases

Wojciech Cellary * and **Geneviève Jomier** **

* Technical University of Poznań, 60965 Poznań, Poland
** Gip Altaïr, BP 105, 78153 Le Chesnay-Cedex, France

Abstract

This paper presents an approach to maintaining consistency of object versions in multiversion database systems. In this approach a multiversion database is considered to be a set of logically independent and identifiable ***database versions.*** Each database version is composed of a version of each object stored in the system. However, identical object versions may be shared by many database versions. Database versions are identified by ***version stamps.*** Version stamps are also used to associate object versions with database versions. Because of the particular construction and semantics of version stamps, object version management is very efficient. Moreover, it is orthogonal to other problems of version management, such as object addressing, concurrency control, access authorization, etc. The paper explains how the requests of object reading, updating, creating and deleting are realized.

1 Introduction

In recent years, development of database technology has addressed non-traditional domains, such as computer aided design (CAD), manufacturing, management, software engineering (CASE) and office automation. Database management systems (DBMS) devoted to these domains need to support new functions. One of the most important is ***version management***, which appears necessary in new object-oriented database systems [9,18,20,22,23,24,29,31,35]. These systems are required to manage simultaneously several versions of the same object. For instance, in computer aided management applications, consecutive real world states appearing one after the other have to be stored in a database. In CASE and CAD applications, a database has to store different alternatives of the same object. Such databases are called *multiversion.*

Various aspects of version management have been considered in the literature: version identification and manipulation, change notification and propagation, version primitives, functions, histories, structures of version graphs, etc. These aspects have been considered separately for CAD databases [3, 8,13,15,16,17,19,21,27], information systems [10,25] and engineering databases [11,12,34], taking application specificity into account. There is also considerable work concerning temporal aspects of databases using versions [1,26,28,30]. All these aspects are important, however, as soon as the database becomes large, with a great number of objects, and many among them with several versions, the key problem of version management is the problem of *consistency.* Intuitively that means that the DBMS must be able to present to the user ***the versions of different objects that go together.*** If this problem is not solved efficiently, it is impossible to query and update the database consistently.

In monoversion databases the problem of consistency is stated as follows. A ***monoversion object*** is defined as a pair (object identifier, object value). A ***monoversion database*** is defined as a set of objects, and a ***monoversion database state*** as the set of values of all the objects contained in the database. A monoversion database is considered to be consistent if it accurately represents a state of the real world that it models. The real world modeled may physically exist or not, as happens in the case of design databases, where a designer stores a representa-

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

tion of a real world state that exists only in his/her mind. Formally, database consistency is defined by consistency constraints imposed on object values. To maintain database consistency, *atomic transactions* are used, which transform one consistent state of the database into another [14].

In multiversion databases the consistency problem is more complex. Now, a *multiversion object* is defined as a pair (object identifier, set of object versions). An *object version* is defined as a pair (version identifier, version value). A *multiversion database* is defined as a set of multiversion objects. A *multiversion database state* is defined as the set of the values of all the object versions contained in the database.

Introduction of object versions has a fundamental consequence: generally, a multiversion database is inconsistent, i.e. considered as a whole it does not reflect any state of the real world. Assume a multiversion database containing two objects, A in two versions a_1 and a_2, and B in one version b_1. Even if b_1 is consistent with respect to both a_1 and a_2, the multiversion database state $\{a_1, a_2, b_1\}$ is inconsistent. As a consequence, in multiversion databases the definition of a transaction as a process that transforms one consistent state of the database into another is no longer valid, because the initial state of a multiversion database is inconsistent. Thus, a fundamental problem of multiversion databases is to recognize which versions of different objects are consistent together. This problem is important, because in a database composed of m objects, each one in n versions, there are up to n^m different subsets of the database containing one version of each object. Even if not all of them are consistent, and even if m and n are small, the user will quickly be lost without the help of the system.

The problem of version consistency has been pointed out in some papers referenced above, and tools to maintain *partial consistency* [3], i.e. consistency of parts of the database, have been proposed. They may be seen as links established between consistent versions of different objects. In [35], these links are given in the form of *slices*, where a slice is a set of object versions that have been produced by a single transaction. In [15,16] they are given in the form of *version histories*, which maintain is-a-descendant-of and is-an-ancestor-of relationships among many versions of the same object, and *configuration objects*, which in CAD are parts of a design hierarchically constructed. In [29] *consistency surfaces* are proposed for tracking the state of particular versions of objects and the degree to which they are consistent with versions of other objects. In [3] the idea of *layers* and *contexts* introduced in Pie [5] is taken up again; a layer is a group of sets of related changes and a context is a sequence of layers. In [3] the problem of configuring a system in software and design database domains is considered. A syntactic characterization of a correct configuration tied to a transaction model is presented. In this model each object is stamped with the signature of the transaction that created it. Then, correct configurations are generated by the use of a version graph for each object and transaction dependence graphs.

The common point of all these approaches is that, by different means, they establish explicit links between consistent object versions. Storage, use and maintenance of these links impose a heavy burden on database management. It grows rapidly with the number of objects and the number of versions of each object [22]. So these approaches seem to be impracticable, except in some limited or particular cases.

In this paper, a totally different solution of the consistency problem, called the *database version approach*, is presented. Its concepts are described in Section 2. Section 3 explains how object versions are managed in the system. Section 4 is devoted to operating on objects. Section 5 deals with concurrency control. In Section 6 version management of composite objects is presented and compared with other approaches. Section 7 concludes the paper.

2 Database Version Approach

The database version approach is not based on the notion of partial consistency of the multiversion database. To solve the problem of multiversion database consistency, we use the same notion of consistency as used in monoversion databases.

A monoversion database stores one representation of the real world state, strictly taken, the last one introduced by the user, which replaces the previous one. If a user of a monoversion database modifies one object, in fact, he replaces the entire representation of one real world state by the representation of another.

Similarly, if a user of a multiversion database creates a new version of an object, in fact, he creates a new representation of an entire real world state. In the future, he will need to retrieve this representation. This is possible if the multiversion database stores the set of representations of the real world states, introduced by the users.

In our approach a representation of a real world state is called *database version*. A multiversion database is defined as a set of logically independent and identified database versions (Figure 1). Formally, a database version is defined as a pair composed of the database version identifier and the set of versions of all the objects contained in the multiversion database, one version per object. The state of a database version is defined as the set of values of all the object versions that it contains.

The concept of database versions allows the use of transactions defined as an extension of the classical definition [14]: a transaction is defined as a process that takes a set of database versions each one from a consistent state to another consistent state. Before or after transaction execution a database version may be empty.

In the simplest case a transaction concerns one database version. It may be non-versioning or versioning. A *non-versioning transaction* queries or updates a database version, causing its evolution independently of the evolution of the other database versions. It corresponds exactly to the notion of transaction in monoversion databases. A *versioning transaction* creates a new database version. It is addressed to a database version, the *parent* database version, and it creates a *child* database version, which is a logical copy of the parent. Thus, the set of database versions is organized as a tree, called *derivation tree*. Once created, the new database version will evolve autonomously, according to the non-versioning transactions addressed to it.

A user operates on a multiversion database in the following way. First of all he chooses (a) database version(s). One way to do that is to specify a database version identifier used by the DBMS. However, it is more convenient to use other identifiers, which reflect the semantics of the database, and which are translated into the system identifiers. For instance, in a temporal database each system identifier of a database version may be associated to a date, in a CASE application to a software configuration [32].

When the database version is chosen, the user may perform non-versioning transactions addressed to it, as if he worked on a monoversion database. The system will automatically identify object versions belonging to the database version chosen. The user, however, is responsible for writing transactions properly, i.e. in a way that a transaction transforms an initial consistent state of the database version into another consistent state. By running a versioning transaction, the user may create a new (child) database version and then work on it. Finally, a user may work simultaneously on several database versions, embedding operations addressed to different database versions in a transaction. In this case, he may, for example, move the value of an object version from one database version to another, browse through the multiversion database, read all the different versions of an object, and so on. The only requirement is that the transaction must transform a consistent state of each database version accessed into another consistent state.

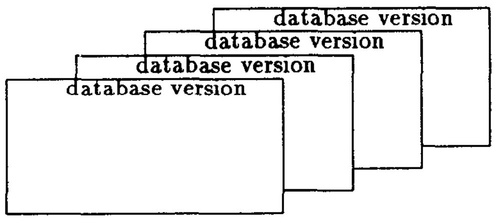


Figure 1. Multiversion database as a set of database versions.

To summarize, there are two levels of operation on a multiversion database. At the upper level the user creates or deletes a specified database version. At the lower level he reads, writes, creates or deletes a specified object in a specified database version.

3 Object Version Identification

Since a child database version usually differs only partially from its parent, versions of the same object contained in different database versions may have identical values. To avoid redundancy, this value has to be physically shared by several database versions. This may be done by associating, for an object, several identifiers of database versions with one value that they share. However, the following problem arises: when a new database version is created, its identifier must be associated with one value of each object stored in the multiversion database. In a large database the association process would be inadmissibly long. To solve this problem, in the database version approach, database version identifiers are constructed in a special way. They are called *database version stamps* or simply stamps.

As the multiversion database is organized as a tree of database versions, the stamp of a database version is constructed in such a way that it makes it possible to identify all the database version's ancestors. If a database version is the n-th child of its parent, whose stamp is p, then the child stamp is

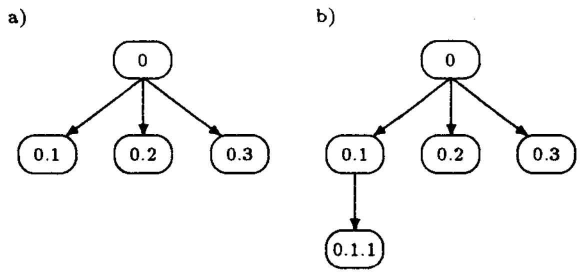


Figure 2. Database version derivation trees.

p.n. The root database version is stamped 0.

The following example shows how stamps are used to identify object versions. Consider a multiversion database, composed of four database versions. Its derivation tree is shown in Figure 2.a. An object A is stored in the multiversion database. From the logical point of view, one version of A appears in each database version. Thus each object version of A may be seen as a row of the relation *Object_Versions_of_A (Value, DB Version Stamp)* presented in Table 1.

Value	DB Version Stamp
a_0	0
a_1	0.1
a_2	0.2
a_2	0.3

Table 1. Relation *Object_Versions_of_A*.

However to avoid the replication of values of versions of A which are identical in several database versions, like a_2 in database versions 0.2 and 0.3, this relation is implemented as shown in Table 2. Each row of this table, named *Oid_A*, may represent several object versions of A. For instance the last row of Table 2 implements object versions (0.2, a_2) and (0.3, a_2) of A. This may also be read "the value of object A in database version 0.3 is a_2".

Oid_A

Value	DB Version Stamps
a_0	0
a_1	0.1
a_2	0.2 , 0.3

Table 2. The multiversion object A.

Suppose now that database version 0.1 has a child, database version 0.1.1 (Figure 2b) and that the value a_1 is shared by database versions 0.1 and 0.1.1. In this case the association of database version stamps with values presented in Table 2 does not need to be modified. It is sufficient to establish a rule saying that:

> ***For an object, if no value is explicitly associated with database version stamp s, then database version s shares the value with its parent database version.***

Suppose that the value of A in database version 0.1.1 is required. Since, in Table 2, no value is stamped with 0.1.1, using the rule above, one can deduce that the desired value is shared with the parent database version, stamped 0.1, so a_1 is found.

This mechanism works recursively for an arbitrary number of ancestor database versions sharing a value. Thus a versioning transaction avoids the explicit association of the stamp of the database version that it creates with a value belonging to each object. As a result, we distinguish between ***unshared value of object version***, belonging to only one database version, and ***shared value of object version***, belonging to several database versions, whose stamps are associated explicitly or implicitly.

Consider now object B stored in the same multiversion database (Table 3).

Oid_B

Value	DB Version Stamps
b_0	0
b_1	0.2
b_2	0.1.1

Table 3. The multiversion object B.

From stamp association $\{a_0, b_0\}$, $\{a_1, b_0\}$, $\{a_2, b_1\}$, $\{a_2, b_0\}$ and $\{a_1, b_2\}$ are consistent because each pair is contained in a database version. On the contrary, $\{a_1, b_1\}$ and $\{a_2, b_2\}$ are not known to be consistent.

In the example above, Table 2 represents a multiversion object A identified by its object identifier, *Oid_A*. Each value a_i may be arbitrarily complex, in particular, it may be totally or partially composed of references to other objects, i.e. their *oid*.

To implement this versioning strategy, the only requirements are that the identifier of an object identifies the data structure implementing the set of its versions, and that each value of object version is associated with its list of database version stamps.

4 Operating on Objects

In this section we explain how the requests for object reading, updating, creating and deleting are performed. These requests are addressed to a database version stamped s.

Reading

To perform a read request, the value of an object version belonging to database version s must be identified and read. This is presented in Section 3.

Updating

To update an object in database version s, its value in database version s must first be identified. Then, it must be determined if this value is shared or not. A value is unshared if only one stamp is explicitly associated with it, and if all the children of this stamp are explicitly associated with other values of versions of the same object. Otherwise, a value of an object version is shared.

If a value v is not shared, it can be updated without any modification of stamp. If it is shared, a new row, *new*, with value v and stamp s must be created in the table implementing the multiversion object. Then v is updated in *new*. The value v of the old row, *old*, remains unchanged, stamped by the version stamps of all the database versions that shared it, except s. Because of implicit sharing, the set of stamps associated with v in *old* contains after the update:

1. All the stamps explicitly associated with v before the update, except s, if s was explicitly associated with v, because the database version s could share v implicitly.
2. The stamps of all the children of the database version s, which are not explicitly associated with other values of versions of the object. Before the update these children shared implicitly v with the database version s.

The performing of an update request is illustrated by the following example. Consider the derivation tree presented in Figure 3. Object A, stored in the database, has two object version values: a_0 belonging only to the database version stamped 0, and a_1, which is shared by the remaining database versions, as shown in Table 4a. Table 4b shows object A after it has been updated in database version 0.1.1. Value a_1 is preserved in database versions 0.1, 0.1.1.1 and 0.1.1.2, which share it explicitly, and database

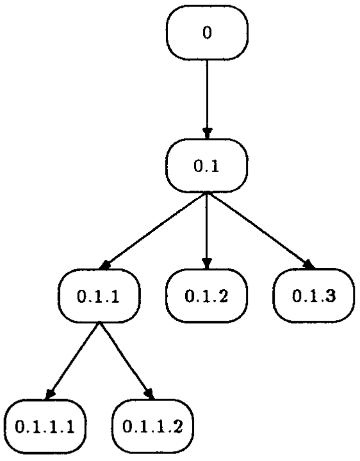


Figure 3. Derivation tree of database versions.

versions 0.1.2 and 0.1.3 which share it implicitly. If object A is updated again in database version 0.1.1, the stamp association remains unchanged (Table 4c). Table 4d presents object A after its update in database version 0.1.

a)

Value	DB Version Stamps
a_0	0
a_1	0.1

b)

Value	DB Version Stamps
a_0	0
a_2	0.1.1
a_1	0.1 , 0.1.1.1 , 0.1.1.2

c)

Value	DB Version Stamps
a_0	0
a_3	0.1.1
a_1	0.1 , 0.1.1.1 , 0.1.1.2

d)

Value	DB Version Stamps
a_0	0
a_3	0.1.1
a_4	0.1
a_1	0.1.2 , 0.1.3 , 0.1.1.1 , 0.1.1.2

Table 4. Updating.

Object creation and deletion are reduced to updating. Formally, all the objects that exist in the multiversion database appear in one version at each database version. Thus, to create a new object, which appears only in a particular database version, but not in the others, or to delete an object in a particular database version, we have to express its non–existence in a database version. To this end a special value *nil* is used. It means ***does not exist.*** The *nil* value of each object is contained in the root database version stamped 0 (in the above example $a_0 = nil$).

Deletion

To delete an object in a particular database version it is sufficient to update it with the *nil* value.

Creation

To create an object in a particular database version, its *nil* value is first introduced in the root database version. Then, the object is updated in the standard way.

5 Concurrency Control

Transactions are executed concurrently and they are serialized by the concurrency controller. However, as values of object versions may be shared between database versions, the concurrency control must be studied at the logical level of database versions and at the physical level of multiversion objects.

On the logical point of view, an access conflict happens only if two transactions addressed to the same database version access the same object: conflict concerns the version of this object belonging to this database version. On the contrary, no conflict happens if the transactions are addressed to different database versions: as two database versions are logically independant, on the logical point of view, their object versions are different.

It follows that access conflicts happen between non-versioning transactions addressed to a database version p and a versioning transaction which wants to derive a child database version c from p. The reason is that the versioning transaction makes a logical copy of the parent database version p to create the child c. Physical copy is avoided because the values of object versions are shared between parent and child database versions. Since the reading of database version p by the versioning transaction is only virtual, locking it may be avoided and the versioning transaction may be serialized in such a way that it precedes all the other non-versioning transactions working on database version p [7].

Conflicts that do not appear at the logical level may appear at the physical level of multiversion objects, because database versions may share values of object versions. The solution to this problem is simple because, as explained in the preceding section, when an object version, whose value is shared, is to be updated in a database version s, its value is replicated and associated only with s (unshared value). Then changes are introduced to this object version belonging to database version s, while the original value remains unchanged in the other database versions. Thus there is no problem if this replication is done in a critical section.

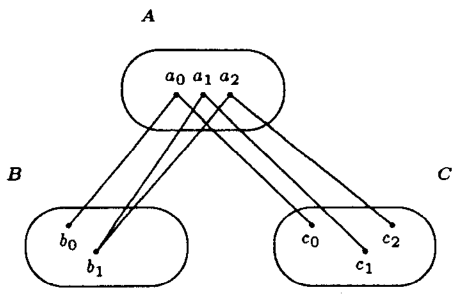
6 Management of Versions of Composite Object

In this section, the database version approach is compared with the other approaches to version management of composite objects. In these approaches, principally, ***object versions refer object versions***, i.e. reference resolution is static. This way of referencing deeply influences version management as presented in [20,22] and briefly described below.

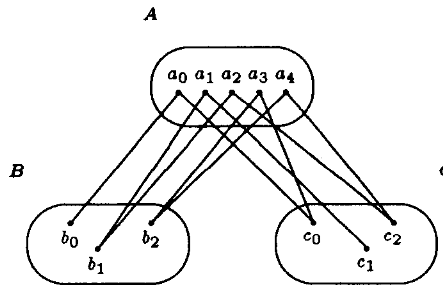
Consider Figure 4a. A composite object A has two components B and C. Each version of A refers to a version of B and C. Figure 4b shows the impact of the creation of a new version of B, b_2: one or more new versions of A must be created. Each new version of A associates b_2 with a version of C consistent with it: for instance a_3 composed of b_2 and c_0, and a_4 composed of b_2 and c_2. If A is itself a component of a composite object of a higher level, E for instance, then several new versions of E must be created. The process of creation of new object versions will continue up to the root of the composite object.

As noticed in [22] and shown in the previous example, the creation of object versions in composite objects may be done through the use of the ***is-part-of*** link, which permits reading a composite object bottom-up, but such links must be maintained by the system. If it does not exist, bottom-up object identification can only be done by memorizing the access path from the object root of the composition hierarchy. However, in this case, no versionable object may be shared by several composite objects [20].

At each level of a composition hierarchy the number of object versions created grows geometrically. This reduces database system performance, since cas-



a) Before creation of b_2.



b) After creation of b_2.

Figure 4. References between object versions.

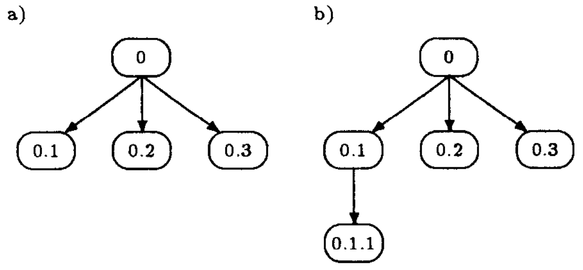


Figure 5. Database version derivation trees.

cading creations require extra read and write operations, and extra overhead of the concurrency control if the database is multiuser.

The process of object version creation may be performed by the user without any system support; then he decides at each step which object versions have to be created. Since this may be very cumbersome, another solution is *percolation* [3] : the version manager automatically creates all the possible versions of composite objects at each level of the composition hierarchy. In this way the user avoids work, but the database is burdened by a large number of useless object versions.

The counterpart of this complex process of creation of object versions for composite objects is that many composite object versions must be deleted in the case of deletion of a version of a component object.

On the other hand, in the database version approach, version management is orthogonal to the object model, and dynamic reference resolution is used. Consider, as an example, three classes $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$, such that each object of class $\mathcal{A}$ is composed of one object of class $\mathcal{B}$ and one object of class $\mathcal{C}$. The database version derivation tree is given in Figure 5a, and five objects: A, B, B', C and C', whose versions are given in Table 5 a,b,c,d,f. From Table 5e, four different versions of object A exist: $\{\alpha_0, b_0, c_0\}$ in database version 0, $\{\alpha_1, b_1, c_1\}$ in database version 0.1, $\{\alpha_2, b'_1, c_1\}$ in database version 0.2 and $\{\alpha_3, b'_2, c'_1\}$ in database version 0.3. Different versions of A are composed of different versions of different objects of the same class and different versions of the same object.

Suppose now that a user wants to create a new version of A composed of b'_1 and a new version c_2 of C. He derives a new database version 0.1.1 from 0.1 (cf. Figure 5b) and updates C in database version 0.1.1: the result is the insertion of c_2 stamped by 0.1.1 (cf. Table 5f). All the other objects remain unchanged and the DBMS is now able to recognize that $\{\alpha_1, b'_1, c_2\}$ is the value of the version of A belonging to database version 0.1.1.

This example shows that in the database version approach the version management does not generate cascaded creations or deletions of object versions when a component object version is created or deleted.

Another consequence of the dynamic reference resolution used in the database version approach is that the internal structure of the value of a nonversionable complex object is the same as the internal structure of the value of a version of a complex object, because both use object identifiers. It is exactly the same as the internal structure of the objects in monoversion databases. Thus, in contrast to other versioning strategies the decision whether an object is versionable does not need to be made when classes are defined.

a) *Oid_B*

Value	DB Version Stamps
b_0	0
b_1	0.1, 0.2, 0.3

b) *Oid_B'*

Value	DB Version Stamps
b'_0	0
b'_1	0.1,0.2
b'_2	0.3

c) *Oid_C*

Value	DB Version Stamps
c_0	0
c_1	0.1, 0.2, 0.3

d) *Oid_C'*

Value	DB Version Stamps
c'_0	0
c'_1	0.1, 0.2, 0.3

e) *Oid_A*

Value			DBV Stamps
Oid_B	*Oid_C*	α_0	0
Oid_B	*Oid_C*	α_1	0.1
Oid_B'	*Oid_C*	α_2	0.2
Oid_B'	*Oid_C'*	α_3	0.3

f) *Oid_C*

Value	DB Version Stamps
c_0	0
c_1	0.1, 0.2, 0.3
c_2	0.1.1

Table 5. Version stamp association in a composite object.

7 Conclusions

The database version approach offers a very powerful tool for managing multiversion databases, because of version stamp semantics. It allows to establish: object version identification, consistency of database versions, history of each object, history of the database versions and the difference between database versions.

This semantics must be compared with the semantics of the version identifiers used in other approaches to version management. In those approaches an object version is identified using a pair (object identifier, version identifier), where version identifier is a local reference to the object. As a consequence, the only possibility offered besides identification is object history.

The difference between the semantics of version stamps and version identifiers explains why the capabilities of versioning mechanism using version stamps includes all the capabilities of versioning systems using version identifiers.

The main advantage of the database version approach is its orthogonality to the object model, object addressing, concurrency control, access authorization and other object management problems.

Version stamps are easy to implement and economical with respect to space. In comparison with other approaches, version management overhead does not grow significantly with the number of object versions.

Our work on the database version approach is in progress. We are extending it to versions of methods and schemas and are implementing it in the O_2 system [4,33] under development at Altaïr.

Acknowledgements

The authors are grateful to François Bancilhon, Claude Delobel, David DeWitt, David Maier, Michel Raoux and Fernando Velez for their helpful comments and suggestions.

References

[1] Adiba M. *Histories and Versions for Multimedia Complex Objects.* IEEE Data Engineering Bulletin, Dec. 1988, pp. 1-8.

[2] Agrawal R., Jagadish H.V. *On Correctly Configuring Versioned Objects.* Proc. 15th VLDB, Amsterdam, Aug. 1989, pp. 367-374.

[3] Atwood T. M. *An Object-Oriented DBMS for Design Support Applications.* COMPINT 85, Montréal, Sept. 1985, pp. 299-307.

[4] Bancilhon F., Barbedette G., Benzaken V., Delobel C., Gamerman S., Lécluse C., Pfeffer P., Richard P., Velez F. *The Design and Implementation of O_2, an Object-Oriented Database System.* Proc. OODBS II Workshop, Bad Munster, FRG, Sept. 1988.

[5] Bobrow D., Goldstein I. *Representing Design Alternatives.* Proc. Conf. on Artificial Intelligence and Simulation of Behavior, Amsterdam, July 1980.

[6] Cellary W., Gelenbe E., Morzy T. *Concurrency Control in Distributed Database Systems.* North-Holland, Amsterdam, 1988, 349 pages.

[7] Cellary W., Jomier G. *Versioning and Concurrency Control in Object-Oriented Databases.*

Altaïr Research Report, 1990, submitted to publication.

[8] Hong-Tai Chou, Won Kim. *A Unifying Framework for Version Control in a CAD Environment.* 12th VLDB Conf., Kyoto, August 1986, pp. 336-344.

[9] Hong-Tai Chou, Won Kim. *Versions and Change Notification in an Object-Oriented Database System.* 25th ACM/IEEE Design Automation Conf., Anaheim, June 1988, pp. 275-281.

[10] Davidson J. W., Zdonik S. B. *A Visual Interface for a Database with Version Management.* 3rd ACM-SIGOIS Conf. on Office Information Systems, Providence, RI, Oct. 1986, p. 52.

[11] Dittrich K. R., Lorie R. A. *Object-Oriented Database Concepts for Engineering Applications.* IBM Res. Rep. RJ 4691 (50029), San Jose, Calif., 5/8/85.

[12] Dittrich K. R., Lorie R. A. *Version Support for Engineering Database Systems.* IBM Res. Rep. RJ 4769 (50628), San Jose, Calif., 7/18/85.

[13] Fauvet M. C. *Etic: un SGBD pour la CAO dans un Environment Partagé.* Thèse de l'Université de Grenoble 1, France, Sept. 1988.

[14] Gray J. *Notes on Data Base Operating Systems*, in: R. Bayer, R. M. Graham and G. Seegmuller (Eds.) *Operating Systems: An Advanced Course*, Springer Verlag, Berlin, 1978, pp. 393-481.

[15] Katz R. H., Chang E. *Managing Change in a Computer-Aided Design Database.* 13th VLDB Conf., Brighton, GB, 1987, pp. 455-462.

[16] Katz R. H., Chang E., Bhateja R. *Version Modeling Concepts for Computer-Aided Design Databases.* ACM SIGMOD Int. Conf. on Data Management, 1986, pp. 379-386.

[17] Katz R. H., Lehman T. J. *Database Support for Versions and Alternatives of Large Design Files.* IEEE Trans. on Soft. Eng., Vol. SE-10, No 2, March 1984, pp. 191-200.

[18] Kim W., Ballou N., Chou H. T., Garza J. F., Woelk D. *Integrating an Object-Oriented Programming System with a Database System.* OOPSLA '88 Proc., San Diego, Calif., Sept. 1988, pp. 142-152.

[19] Kim W., Banerjee J. *Support of Abstract Data Types in a CAD Database System.* COMPINT 85, Montréal, Sept. 1985, pp. 381-385.

[20] Kim W., Banerjee J., Hong-Tai Chou, Garza J. F., Woelk D. *Composite Object Support in an Object-Oriented Database System.* OOPSLA 87 Proc., Orlando, Fla., Oct. 1987, pp. 118-125.

[21] Kim W., Batory D. S. *A Model and Storage Technique for Versions of VLSI CAD Objects*, in: *Foundations of Data Organization*, Plenum Press, pp. 427-439.

[22] Kim W., Bertino E., Garza J. F. *Composite Objects Revisited.* SIGMOD Record, Vol 18, No 2, June 1989, pp. 337-347.

[23] Kim W., Hong-Tai Chou. *Versions of Schema for Object-Oriented Databases.* 14th VLDB Conf., Los Angeles, Calif., 1988, pp. 148-159.

[24] Kim W., Woelk D., Garza J. F., Chou H. T., Banerjee J., Ballou N. *Enhancing the Object-Oriented Concepts for Database Support.* Third Int. Conf. on Data Engineering, Los Angeles, Calif., Feb. 1987, pp. 291-292.

[25] Klahold P., Schlageter G., Unland R., Wilkes W. *A Transaction Model Supporting Complex Applications in Integrated Information Systems.* Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas, May 1985. SIGMOD Record Vol. 14, No 4, Dec. 1985, pp. 388-401.

[26] Klahold P., Schlageter G., Wilkes W. *A General Model for Version Management in Databases.* 12th VLDB, Kyoto, August 1986, pp. 319-327.

[27] Landis G.S. *Design Evolution and History in an Object-Oriented CAD/CAM Database.* CH2285-5/86/0000/0297, IEEE 1986, pp. 297-303.

[28] Lum V., Dadam P., Erbe R., Guenauer J., Pistor P., Walch G., Werner H., Woodfill J. *Designing DBMS Support for the Temporal Dimension.* SIGMOD 84, Boston MA, June 1984, SIGMOD Record Vol. 14, No 2, 1985, pp. 115-130.

[29] *Vbase Integrated Object System, Technical Overview.* Ontologic Inc., 47 Mannings Road, Billerica MA, 1987.

[30] Stam R., Snodgrass R. *A Bibliography on Temporal Databases.* Dept. of Computer Science, Univ. North Carolina, Chapel Hill, NC 27514, Sept. 1988.

[31] Stonebraker M., Rowe L. *The Postgres Papers.* Memorandum No. UCB/ERL M86/85, Nov. 1986.

[32] Tichy W.F., *Tools for Software Configuration Management.* Proc. 11th International Conference on Software Engineering, May 1989.

[33] Velez F., Bernard G., Darnis V. *The O_2 Object Manager: an Overview.* 15th VLDB Conf., Amsterdam, Aug. 1989.

[34] Woelk D., Kim W. *Multimedia Information Management in an Object-Oriented Database System.* MCC Technical Report Number DB-046-87, Feb. 1987.

[35] Zdonik S.B. *Version Management in an Object-Oriented Database.* Int. Works. on Adv. Programming Environments, Trondheim, Norway, 1986, pp. 139-200.