A Unifying Framework for Version Control in a CAD Environment

Hong-Tai Chou and Won Kim

Microelectronics and Computer Technology Corporation
9430 Research Blvd.
Austin, Texas 78759

ABSTRACT

Version control is one of the most important functions which need to be supported in integrated computer-aided design (CAD) systems. In this paper we address a broad spectrum of semantic and operational issues in version control for a public/private distributed architecture of CAD systems. The research issues we address include the semantics of version creation and manipulation, version naming and name binding, and version change notification. We develop solutions to these issues under a unifying framework, and discuss implementation and application interface issues.

1. Introduction

In recent years there has been a tremendous surge of interest in research and development of computer-aided design (CAD) systems for aiding and controlling the design efforts for a wide variety of engineering artifacts, including VLSI circuits, mechanical parts, software systems, multimedia documents, etc. There is a general consensus in the computer-aided design (CAD) community that version control is one of the most important functions in an integrated CAD system. Designers often need to generate and experiment with multiple versions of a design, before selecting one that satisfies the design requirements. A complex design consists of lower level components. A component may be shared by any number of designs, and may in turn consist of lower level components. When a lower level component is changed, the higher level component that contained it may become invalid, and thus need to be notified of the change.

The literature abounds with proposals that address various aspects of version control for CAD applications [ROCH75, TICH82, KAIS82, WIED82, NEUM82, McLE83, PLOU83, HAYN84, KATZ84a, DADA84, BATO85b, ATWO85, DITT85, KATZ86]. In spite of these efforts, to our knowledge no comprehensive framework for version control in an integrated CAD system exists. We believe that there are three major reasons for this. First, most of the existing proposals address only limited subsets of the spectrum of semantic and operational issues in version control. Second, most of them fail to take into account the characteristics of CAD environments, namely, the system architecture and the way in which users and applications share data and interact among themselves. Third, most of them fail to consider the characteristics of CAD databases, namely, the way in which CAD objects are represented and used.

In this paper, we attempt to take significant first steps towards establishing a unifying framework for version control in integrated CAD systems. The framework incorporates explicitly the characteristics of CAD environments and CAD databases. Within this framework, we identify what we hope to be a comprehensive set of semantic and operational issues in version control, and indicate solutions to these issues that are consistent within the framework. We have adopted existing solutions to some aspects of these issues, and developed our own solutions to other issues.

The system architecture often envisioned for CAD systems consists of a public system (central server) and a collection of private systems communicating with the public system [HASK82, KATZ84b, LORI83, KIM84]. The public system manages the public database of stable design data and design control data. A private system manages the private database of a designer on a design workstation. Private systems check out versions of design data from the public system, update them, and check them in as new versions to the public database.

The versions residing in the public system are supposed to be more 'stable' than those in the private systems. The public versions are supposed to be sharable among many users, while the private versions are owned and accessed by a single user. It is clear then that public versions should have a different set of capabilities from private versions; that is, the users will manipulate the public versions differently from private versions. This is the basis of our notion of *version capabilities*, which forms an important basis of our model of versions.

To keep a complex design humanly manageable, a design, which we will call a *CAD object*, is often represented as a *configuration hierarchy* in which a component of the design consists of progressively more detailed lower level components. One of the most important requirements of a CAD database is to allow any subtree of such a hierarchy to be shared among any number of designs. Without such sharing, the size of the database and the difficulty of maintaining the consistency in multiple copies of the same component will quickly get out of hand. To support sharing, an object at any level of a configuration hierarchy *references* (points to) its lower level components. One of the problems this imposes on a CAD database system is that of change notification. When a lower level component is updated or deleted, the higher level component that references it should in general be notified.

The remainder of this paper is organized as follows. We summarize the characteristics of CAD environments and CAD databases in Section 2. In Section 3, we propose our model of versions, based on the notion of version capabilities. We extend the discussions of the semantic issues of our model in Section 4 to version naming and name binding, that is, binding versions with a version that references them. In Section 5, we explore issues in change notification, including timing and scope of notification, and notification techniques. Then in Section 6, we present data structures for implementing our model of versions, including those for version history and change notification. In Section 7, we provide a provisional list of commands that applications and users may issue to use our model.

2. CAD Environment and CAD Databases

In this section, we briefly summarize the characteristics of CAD environments and CAD objects. Incorporation of these characteristics is the cornerstone, and in our view one of the significant contributions, of our model of versions.

Proceedings of the Twelfth International Conference on Very Large Data Bases **Kyoto, August, 1986**

2.1. CAD Environment

Integrated CAD systems of the future will consist of intelligent workstations and central server machines on local-area networks. The designers will perform much of the design and design validation work on workstations, to minimize the need to send long batch jobs to remote servers. Although the role of workstations will grow, central servers will continue to play important roles in CAD systems. A server may be a large mainframe or a local-area distributed network of superminis. There will be two types of central servers: compute servers (number crunchers) and database servers. The compute servers will process long batch jobs sent from the workstations; for example, in VLSI CAD, the simulation and design-rule check of full designs. The database server will manage the *public database* of stable design data and design control data, and coordinate the sharing of data among database systems running on workstations.

A workstation in general will have a *private* database system. A private database system manages the private database of a designer on a workstation. An application running on a workstation checks copies of design data out of the public database, inserts them into the private database on the workstation, retrieves and manipulates data in both the private database and the public database, and checks (returns) modified design data into the public database.

In [BANC85] we defined a model of transactions in a CAD environment in terms of projects and cooperating designers within each project. The design objects (versions) in the public database are accessible to any designer. The design objects in a private database are owned and accessed by only one user. A designer may share data with other designers who belong to the same project. This has led us to the notion of a project database, which serves as the repository of design objects that are being passed back and forth among designers within the same project. A designer places design objects in his or her project database. Another designer checks them out, updates them, and returns (checks) them, as new versions, in the project database. The database server manages both the public database and the project database of each of the projects.

The database hierarchy in our model of CAD environment then includes the private databases in workstation database systems, the public database of the database server, and the project databases associated with projects. This is shown in Figure 1.

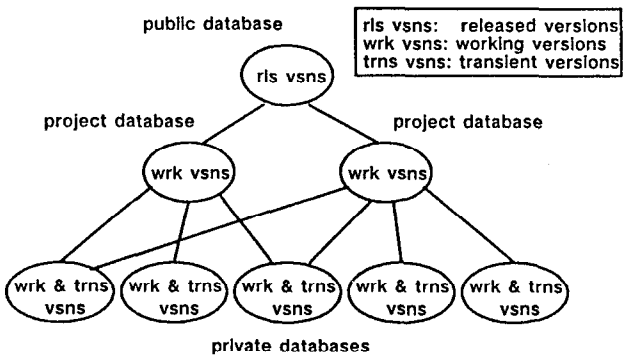

Figure 1. Database Hierarchy

More precisely, the differences in the three databases are as follows. The public database holds released design objects and data about design objects and design status. A released design has two properties: it cannot be updated or deleted, and it is accessible to all authorized designers within a CAD environment. The data about design objects and design status include the directory of design objects, checkout/checkin status, etc. They are also accessible to all authorized designers; however, they can be updated, either by the designers, or by the database server. There will be a public database administrator for the public database.

A private database holds non-released designs that a designer is currently working on, and any information the designer wishes to maintain. That is, the designer who creates the private database is the owner and database administrator of the database. A private database of a designer is not accessible to any other designer.

A project database contains those designs and data about designs that are accessible only to cooperating designers within a project. The design objects in a project database are considered stable; however, the assumption is that they have not gone through validation tests and, as such, they cannot be released to the public. A project administrator will have the database administrator privileges over a project database.

2.2. Model of CAD Objects

In this subsection, we highlight the model of CAD objects, developed in [KATZ86] and our earlier work [BATO85a]. The model captures three types of relationships that exist among CAD objects in a database. We assume that the applications may query and manipulate (insert, delete, update) these relationships through the normal capabilities of CAD database systems.

First, a CAD object may have any number of versions. And any number of versions may be derived from an existing version. The description of a CAD object consists of two parts: its interface and implementation [EDIF84, McLE83, BATO85b]. We define *versions* to be objects that share the same interface but have different implementations. The *interface*, for example, of a circuit contains inputs and outputs, and specifies the function of the circuit. The *implementation* consists of descriptions of less complex, component circuits, and their interconnections, where each component circuit has its own interface and implementation. In computer-aided software development environments, an object may be a procedure. The interface part of a procedure is the parameter list; the implementation part is the code segment.

Second, a CAD object of any complexity is represented as a hierarchy of progressively more primitive objects. This is done by including in an object *references* to other, more primitive objects. The hierarchical composition of a complex CAD object is often called the *configuration* of the object. A component object may be referenced by any number of other objects, and may in turn reference any number of other objects. Therefore, the database of a set of CAD objects forms a directed-acyclic graph (DAG) of independent, sharable objects.

Third, a version of a CAD object can have a number of equivalent representations. For example, a VLSI CAD object may be in register transfer, Boolean, logic, circuit, or layout representations. In our earlier work [WOEL86], we noted that a chapter of a book in a multimedia document may be in text, audio, or movie representations. One representation of an object is often derived from another representation. In VLSI, for example, circuit representations are generated from logic representations and circuit representations are extracted from layout representations.

3. Version Capabilities

In this section, we define three distinct types of versions, in terms of operational capabilities, and provide a formal characterization of them. The three types of version capabilities arise directly from the three types of databases in the database hierarchy of a CAD environment.

3.1. Version Semantics

After the initial creation of a design object, new versions of the object can be derived from it, and new versions can in turn be derived from them, forming a *version-derivation hierarchy* for the object. A version-derivation hierarchy captures the evolution of the design and indicates a partial ordering of the versions of the

object.

We will distinguish three types of versions, on the basis of their location in the database hierarchy in a distributed CAD environment, and thus of the types of operations that may be allowed on them. They are transient versions, working versions and released versions, shown in Figure 1. We characterize their properties below.

A *transient version* has the following properties.

1. It can be updated by the designer who created it.
2. It can be deleted by the designer who created it.
3. A new transient version may be derived from an existing transient version. The existing transient version then is 'promoted' to a working version (this will be discussed shortly).
4. It is stored in the private database of a designer who created it.

A *working version*, called an 'effective version' in [KATZ84a], has the following properties.

1. It can exist in a private database or in a project database.
2. It is considered stable and cannot be updated.
3. It can be deleted by its 'owner' (this is explained shortly).
4. A transient version can be derived from a working version.
5. A transient version can be 'promoted' to a working version. Promotion may be explicit (user-specified) or implicit (system-determined).

A transient version which is promoted to a working version may continue to reside in the private database in which it was created, or it may be checked into a project database. There are two ways in which a transient version V can be implicitly promoted to a working version. One is when a new transient version is derived from V. Another is when V is checked into a project database.

A working version that resides in a private database is owned by the designer who created it as a transient version and promoted it to a working version. However, a working version in a project database is owned by the database administrator for the project. That is, a working version can be deleted by a project database administrator.

There are two reasons we impose the update restriction on working versions. One is that it is considered stable and thus transient versions can be derived from it. If a working version is to be directly updated, after one or more transient versions have been derived from it, we need a set of careful update algorithms (for insert, delete, update) which will ensure that the derived versions will not see the updates in the working version. In fact, we developed such a set of algorithms in [BATO85b]. However, in view of the fact that all a designer has to do to 'update' a working version is simply to create a new transient version, we agree with [KATZ85] and [NEWT85] that the added complexity of such algorithms is not justified.

Another reason for the update restriction on working versions is that, as we will show in the next subsection, it eliminates the need to support update-mode checkouts. In other words, all checkouts under our model will be read-only, which in turn means that transactions issuing checkout requests will not be blocked.

We impose no restriction on the number of working versions on a version-derivation hierarchy of a given design object. In particular, we allow more than one transient versions derived from a working version to be promoted to working versions. For example, the designer may wish to place two transient versions in a project database, so that other designers may work on them. To be consistent with our view of project databases, both transient versions then need to be promoted to working versions.

A *released version* has the following properties.

1. It resides in the public database, and is managed by the database server.
2. It is not updatable.
3. It is not deletable.
4. A transient version can be derived from a released version.
5. A working version can be promoted to a released version.

3.2. Version Creation

There are three ways to create versions: checkout and checkin, derivation, and promotion. As different authors use the terms checkout and checkin in somewhat different ways, we will define them here.

Definition: A *checkout* of a version Vi from a database Ds to a database Dt involves installing a copy of Vi as version Vj in Dt, without destroying Vi in Ds.

Definition: A *checkin* of a version Vj from a database Ds to a database Dt involves installing a copy of Vj as version Vk in Dt, without destroying Vj in Ds.

Now we state how the three types of versions can be created.

A transient version can be

1. created by a checkout of a released version from the public database,
2. created by a checkout of a working version from a project database,
3. derived from a transient or working version in a private database, or
4. created from scratch from a workspace copy on a private system.

We note that there is no notion of a checkout from a private database. To reference other designer's work, a designer can check a version out of either the public database or a project database. After a version is checked out, it becomes a transient version in the private database, ready for direct manipulation by the designer who checked it out.

A working version can be created

1. in a private database by promoting (explicitly or implicitly) a transient version in the private database, or
2. in a project database by a checkin of a working version from a private database.

A released version is created by

1. a checkin of a working version from a private database, or
2. a checkin of a working version from a project database.

Our model of versions only requires read-mode checkouts. This should be contrasted with other proposals. [HASK82] defines two modes of checkout; read and write. The two modes conflict, which means that, in case of a conflict, a checkout request must be either rejected or forced to wait until the designer(s) who had checked out the design object checks it back in. [LORI83, KIM84] add a third mode, a *copy mode*. A copy mode checkout does not conflict with either the read or write mode checkout. The motivation for this mode is to allow designers to get a copy of a version, which may be in the process of update, and use some portions of it in some other designs.

In practice, a user may wish to create in his database more than one version-derivation hierarchy for any design object. Further, as we will show in Section 7, a version-derivation hierarchy may be *split* into a number of independent hierarchies for the same design object. However, for expository simplicity, we will assume a single version-derivation hierarchy for a design object in any database.

Further, when a user checks out or checks in a version V from a database Ds to another database Dt, he needs to specify the new parent of V in the database Dt. For example, when a user checks a version Vr out of the public database and installs it as a new transient version Vt in his private database, he should indicate the identity of the version in the private database which will become the parent of Vt on the version-derivation hierarchy. Similarly, when the user checks a working version Vw into the public database, he will specify the identity of the released version which will become the parent of Vw in the public database.

4. Version Naming and Name Binding

In this section, we discuss how versions are uniquely identified in a database. The reader may find some aspects of the

discussions to be rather complex. We emphasize, however, that most of the apparent complexity are very natural consequences of the distributed nature of a CAD environment. In fact, we hope that the discussions of this section will help the reader to recognize the inherent inadequacy of the existing proposals for version control.

4.1. Version Names

As versions may exist anywhere in the database hierarchy, the full name for each version of an object is a triplet <object name, database name, version number>, where 'database name' is the name of a private database, a project database, or the public database. Versions on a derivation hierarchy in a particular database are assigned monotonically increasing integers in the order of their creation.

We associate a distinct version-name server with each database in our database hierarchy. Each private database system is the version-name server for all its transient and working versions. The database server is the name server for all released versions in the public database. The database server also functions as the name server for the working versions in a project database, by providing one logical name server for each project database.

4.2. Name Binding

There are two ways to *bind* an object with another versioned object: static and dynamic. In *static binding*, the reference to an object includes the full three-part name of the object. In *dynamic binding* [PLOU84, DITT85, ATWO85, KATZ86], the reference needs to specify only the object name, and may leave one or both of the other two parts unspecified. The system selects the default database name and version number. If the database name of the referenced version is unspecified, the system will assume it to be the same as that for the object that makes the reference. Clearly, dynamic binding is useful, since existing transient or working versions may be deleted, and new versions created.

Contexts

A natural extension of the idea of dynamic binding is that of contexts [DITT85, ATWO85]. We define a *context* to mean the specification of default versions for a particular configuration of a complex design object. The user may define a number of contexts, and, by switching from one context to another, can experiment with various alternative configurations of a design object.

For example, suppose a design object consists of three component objects, A, B, and C. The object A has two versions, Va1 and Va2; B has one version, Vb1; and C has two versions, Vc1 and Vc2. This gives rise to four possible configurations, or contexts, of the design object: (Va1 Vb1 Vc1), (Va1 Vb1 Vc2), (Va2 Vb1 Vc1), and (Va2 Vb1 Vc2). The user may want to evaluate the design object in any of these contexts.

Default Selection

There is nothing novel about the idea of dynamic binding or contexts. However, we need to examine the issue of selecting default versions for dynamic binding. In other proposals, the default selected is often the 'most recent' version. This simple defaulting scheme is not appropriate in our model. One difficulty is that in our model version history is represented in a hierarchy, the version-derivation hierarchy. In a linear-derivation scheme, where only one version may be derived from any version [DADA84], the most recent version has the implicit meaning that it is the 'most correct' or 'most complete'. However, a version-derivation hierarchy, where any number of new versions may be derived from any node on the hierarchy any time, potentially has any number of 'most recent' versions in this sense. Therefore, we need to allow the user to specify a particular version on the version-derivation hierarchy as the default version. In the absence of a user-specified default, the system will select the version with the 'most recent' timestamp as the default.

Another difficulty with dynamic binding in our model is that versions of different capabilities reside in different databases. A transient version resides in a private database of a user. A working version resides either in a private database of a user or in a project database to which the user has access. A released version resides in the public database. We need to define a *search order* over the database hierarchy as follows. Suppose Va, a version of an object A, references an object B.

1. If Va is in the private database of a user, search first the private database, then the project database of the user, followed by the public database, selecting the default version in the first database that has any version of B.
2. If Va is in a project database, search first the project database, followed by the public database, selecting the default version in the first database that has any version of B. (The reason private databases are not searched will be explained later.)
3. If Va is in the public database, search the public database, for the default version of B. (The reason private and project databases are not searched will be given later.)

Checkout and Checkin

It is clear that when a version of an object is released (checked into the public database), all versions it references must also be released. This is why we do not search any private or project database for an object which is referenced by an object in the public database.

Similarly, when a version in a private database is checked into a project database, we assume that all objects it references will also be checked in, if they are needed. Therefore, we do not search any private database for an object which is referenced by an object in a project database. If the user checks an object into a project database but fails to also check in some of the objects it references, the behavior of the system will be unpredictable. It may find some working versions in the project database, or some released versions in the public database.

Surprisingly, static binding also presents problems when versions are checked into the public database or project databases. When a version and all versions it references are checked in, in general all static references the version has to other versions must be converted to new static references that have meaning in the new database.

For example, suppose a working version Va references a lower level version Vb, and the reference specifies <Vb, my-private-db, version-5>. Suppose also that both versions reside in a private database, my-private-db. When the versions are checked into the public database, the reference to Vb in Va must be converted to a new static reference that has meaning in the public database, say <Vb, public-db, version-3>.

When a version is checked out or checked in, the version-name servers generate version numbers as follows.

1. When a private database system checks out a version, it generates a version number for the newly created transient version.
2. When a private database system checks a working version into a project database, the name server for the project database assigns a version number for the new working version in the project database.
3. When a working version is checked into the public database, either from a private database or a project database, the name server for the public database assigns a new version number to the newly released version.

We note that when a transient version in a private database is promoted to a working version, the status of the version is updated, but no new version number is generated.

The reason for assigning a new version number to a transient version at the time of checkout, rather than later at checkin time, is that it is often necessary to reference the new version from other objects, before the new version is ready for checkin.

4.3. Versions of Schemas

In a database, the schema is used to control the creation and manipulation of design objects. In a CAD environment, the users tend to arrive at the schema for design objects through trial and error [WOEL86]. As such, it is important to allow flexibility in the definition and modification of schemas. If systematic query and update capabilities are desired for versions of design objects based on any particular schema, we must support versions not only for design objects, but also for schemas of the design objects. We define the following semantics for versions of schemas, which are consistent with our model of versions of design objects.

1. A version of schema for a design object X is in general shared by multiple versions of X. For example, if a transient version is derived from a working version in a private database, both versions may use the same version of schema.
2. The version of schema used for version Vi of a design object may be different from that used for version Vj derived from Vi. For example, after a designer creates a transient version by checking out a version, he may modify the schema for the transient version. Then the original version and the transient version will use different schemas.
3. A version of schema for a design object X resides in the database along with versions of X that are based on that version of schema. For example, if a transient version of a design object has been derived from a released version, the schema must exist in both the public database and the private database in which the transient version has been created.

The above discussion suggests that when a version V is checked into the public or project databases, or when it is checked out, the version of schema for V must precede V, if the version of schema does not already reside in the database to which V is being sent.

5. Version Change Notification

In [BATO85b] we identified some issues to consider, and proposed preliminary solutions, to have a database system react to changes in versions in a CAD environment. In this section, we extend our earlier work, and provide a framework for the discussions in the next section of the implementation issues for our model of versions and change notification.

5.1. Change Notification Requirements

We have stated all along that a version of an object may reference any number of versions of other objects. We now make this precise in the context of our database hierarchy.

1. A transient or working version in a private database may reference other transient or working versions in the same private database, working versions in the project database of the user of the private database, or released versions.
2. A working version in a project database may reference other working versions in the same project database or released versions.
3. A released version may reference other released versions.

From the above characterization of references that the system will support, the following situations may require change notification.

1. A transient or working version in a private database references a transient version in the same private database, and the transient version is updated, deleted, or a new version of it is created.
2. A transient or working version in a private database references a working version in the same private database or in a project database, and the referenced version is deleted or a new version of it is created.
3. A working version in a project database references another working version in a project database, and the referenced version is deleted or a new version of it is created.

5.2. Message/Flag-Based Notification

In a distributed CAD environment, two types of notification techniques must be supported: message-based and flag-based. In the *message-based* approach, the system sends messages to notify (human) users of potentially affected versions. The message-based approach is further distinguished as *immediate* or *deferred*, depending on whether the affected users are notified immediately after the changes to a version are committed or at some later time that the users may have specified.

In the *flag-based* approach, the system simply updates data structures that it maintains, so that affected users will become aware of changes in a version only when they explicitly access the version. The flag-based approach is necessarily a deferred notification strategy.

We see that an object has a number of change-notification options at its disposal: message vs. flag-based, immediate vs. deferred (in the case of message-based notification), and types of changes to post notification (update, delete, creation of a new version). When the application defines an object, it must specify these options with respect to the versioned objects it references. However, it is impractical to require the user to specify a possibly different set of options for each of the references in an object, since an object may reference a large number of other objects. We believe that a more sensible approach is to have a single set of options specified for an object, and apply it across all objects the object references.

5.3. Flag-Based Notification Technique

In this subsection, we present our flag-based notification technique. We will describe a message-based notification technique in Section 6.2. In [BATO85b] we presented a preliminary proposal for a flag-based notification technique. Since then we have refined the technique and found that it compares favorably against another interesting technique that has recently come to our attention.

In our scheme, each version of an object has two distinct timestamps. One timestamp, called the *change-notification timestamp (CN)*, indicates the time the version was created or the last time it was changed. The other, called the *change-approval timestamp (CA)*, indicates the last time at which the designer of the version approved of the changes to the version. Let V.CA and V.CN denote the change-approval and change-notification timestamps of version V. Let I be the set of versions that are referenced by version V. If no version in I has a change-notification timestamp that exceeds the change-approval timestamp of V (i.e, for all X in I, X.CN $<=$ V.CA), then V is *reference consistent*. V is *reference inconsistent* if there are one or more versions in I that have been updated, but the effects of these updates on V have not been determined.

To make V reference consistent, the effects of the updated versions in I must be acknowledged. This is done in one of two ways. Either the updates to I have no effect on V, in which case V.CA will be set to the current time, or V will need to be modified, in which case V.CN (and possibly V.CA if the changes are approved) will need to be set to the current time. Until such actions are taken, V will remain reference inconsistent.

For each object we need to maintain the version number of each version the object references. This is necessary to support dynamic binding. Suppose a version Va references a transient version Vb3, which was derived from a working version Vb2. Vb3 is updated, and subsequently Va approves the changes in Vb3. Then Vb3 is deleted. With dynamic binding, Va will now be bound to Vb2. The change approval timestamp alone does not capture the fact that the approval was for the changes in Vb3, not for Vb2.

A different notification technique, called *version percolation*, is presented in [ATWO85]. In this scheme, when a new version is derived from an old version of an object, the system automatically

generates new versions of objects that directly or indirectly reference the old version of the object. This technique has some annoying shortcomings. One is that it may generate a large number of useless versions. In the example shown in Figure 2, suppose V1 of object A references V1 of object B and V1 of object C. If the user derives new versions V2 of objects B and C, even if the user's intention was to only create a new version V2 of object A, the system will have generated three new versions of A. V2 of A will reference V1 of B and V2 of C; V3 of A will reference V2 of B and V1 of C; and V4 of A will reference V2 of B and V2 of C!

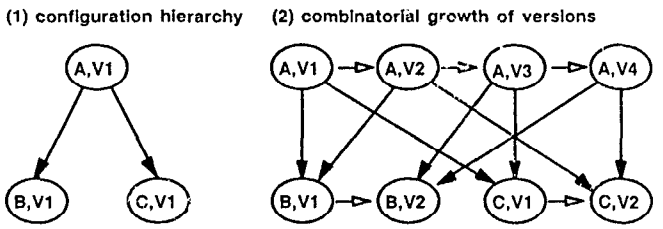

Figure 2. Version Percolation

Another shortcoming of the percolation technique is that, when used by itself, it is not useful when versions of a component object are deleted. In our current example, the user of V1 of object A will not be notified of the deletion of V1 of object B or V1 of object C!

5.4. Notification Scope

The fact that a version in general references other versions in a recursive manner presents a problem with the *scope* of change notification. Suppose, for example, that a version Vi references Vj, and Vj references Vk. If version Vk is deleted, should both Vj and Vi be notified, or only Vj be notified? In general, the possibilities are

1. to notify only the versions that directly reference the changed version, or
2. to notify all versions that directly or indirectly reference it.

The philosophy behind the first approach, which we are adopting, is that the (human) users of Vd, the version that directly references Vc, the changed version, should react to the change. The users may determine that no corrective actions need to be taken on Vd, and thus there is no need to notify Vi, the version that references Vd. Only if the user updates Vd, in response to the changes in Vc, the changes in Vd will then cause a notification to the users of Vd.

The case for notifying only Vd is especially strong under our model of CAD object. As discussed in Section 2.2, a CAD object often has the interface part and the implementation part. The interface part is not updatable; only the implementation part may be updated, resulting in new versions. We expect that in practice the types of changes that require notification are mostly on the interface part, which by definition is not updatable. As such, we believe that there will only be a low probability that Vd needs to be updated in response to any updates to the implementation part of Vc, and thus even lower probability for Vi to be updated. If Vc is deleted, Vd will need to be updated, possibly to reference another version; however, it is not very likely that Vi will also need to be updated.

6. Implementation Issues

In this section, we specify the minimal set of data structures that a database system must maintain, in order to support our model of versions. In particular, we will identify the types of information necessary to implement version-derivation hierarchies and change notification, both flag-based and message-based.

6.1. Version-Derivation Hierarchy

The version-derivation hierarchy of a design object is recorded in a *version table* associated with the object. A version table consists of

1. an object name,
2. a default version number,
3. a next-version number,
4. a version count, and
5. a set of version descriptors, one for each existing version on the version-derivation hierarchy of the object.

The default version number, which is zero initially, determines which existing version on the version-derivation hierarchy should be chosen when a partially specified reference is dynamically bound. The next-version number is the version number to be assigned to the next version of the object that will be created. It is incremented after being assigned to the new version.

A version descriptor contains control information for each version on a version-derivation hierarchy. It includes

1. the version number of the version,
2. the version number of the parent version,
3. the change-notification and change-approval timestamps,
4. the storage location of the version,
5. the schema version number associated with the version, and
6. a pointer to the list of versions the version directly references (this list is the component table, described in the next subsection).

Further, depending on the database type, a version descriptor can be in one of the two formats shown in Figure 3. For a version in a private database, we need to specify the version type as either transient or working. For a version in the public or a project database, we may record the identity of the creator of the version (the designer who checked in the version).

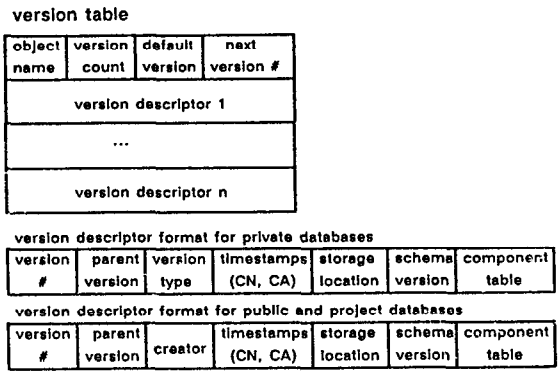

Figure 3. Version Table Format

6.2. Change Notification

In this section, we describe data structures necessary to implement the flag-based change notification technique discussed in Section 5.2, as well as a message-based notification technique.

Flag-Based Notification

We need a data structure, we call a *component table*, to implement the configuration hierarchy and flag-based change notification. A component table, shown in Figure 4, is associated with each version, and contains the following information about the versions it directly references as components.

1. the number of components of the version,
2. the method of change notification (either flag-based or message-based),
3. the type of event to post a notification (any combination of version creation, deletion, and update), and
4. a set of component descriptors.

Each component descriptor contains the identity of a referenced version and the type of binding being used. For static binding, the full name of the referenced version is recorded. For dynamic binding, as explained in Section 5.3, the database name and/or the version number of the version whose changes were last approved by the parent version are recorded for the unspecified database name and/or the version number.

Message-Based Notification

To support message-based notification, we need to maintain, for each version V, an *inverted reference list* of versions which reference V and which require notification of changes to V. When a new reference to V is created, the name of the version that references V is appended to the inverted reference list of V. As shown in Figure 5, for each reference, the event type (a combination of update/deletion/creation of versions) and the timing of notification (immediate or deferred) are also recorded. When a version with a

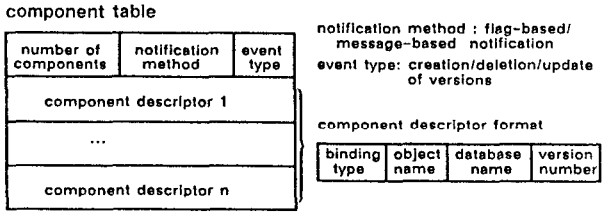

Figure 4.1. Component Table Format

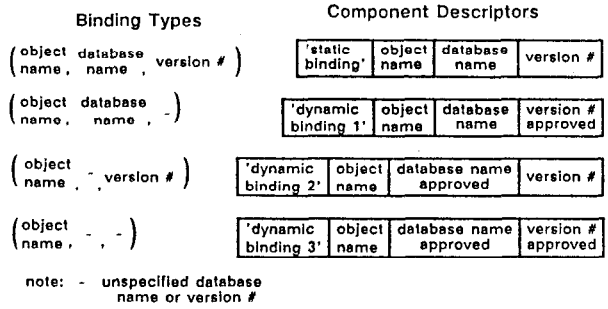

Figure 4.2. Component Descriptors for Different Binding Types

non-empty inverted reference list is changed, the list is scanned for the databases that currently contain those versions with a matching event type, and messages are sent to the owners of those databases.

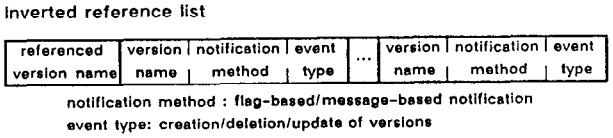

Figure 5. Inverted Reference List Format

6.3. Other Miscellaneous Data Structures

To support efficient identification of a version table for a given object name, we need to maintain a hash table for each database in our database hierarchy. Using the object name as a key, the hash table returns a pointer to the version table associated with the object.

To allow a designer to query the status of a shared version, we need to maintain a *checkout table* for the public database and each project database. A checkout table keeps track of all the versions that have been checked out. A designer can access a checkout table, provided that the corresponding database is accessible to the designer, through a normal query interface. As shown in Figure 6, each entry of a checkout table contains:

1. the name of the version checked out,
2. the version number of the checked-out version,
3. the time the version was checked out, and
4. the identity of the designer who checked out the version.

For the purposes of security and addressability, two other types of data structures need to be maintained. In each project database, members of the project are recorded in a list. Upon a checkin or checkout request, the identity of the requestor is checked against this list. Only registered members of a project are allowed to access the corresponding project database. Conversely, the names of the project databases that are accessible to a designer are recorded in the designer's private database. As each designer may participate in several projects concurrently, the identity of all the project databases to which the designer belongs should be recorded in his private database.

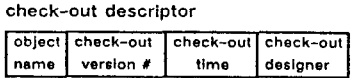

Figure 6. Check-out Registration Table Format

7. Application (User) Interface

It may appear to the reader that our model of versions is rather complex. However, we emphasize that there are two major reasons for this. One is that our model takes into account the distributed nature of a CAD environment and the complex configuration of CAD objects. Another reason is that we have provided detailed discussions of a very broad spectrum of semantic and operational issues on versions. Therefore, the reader may find it surprising that the application (user) interface we present in this section contains a very small number of commands that application programs (users) need to issue in order to take full advantage of our model of versions. We provide a BNF definition of the commands in Figure 7.

We note that, in addition to these commands, we need language constructs to declare schemas of objects and configuration hierarchies. In particular, we must be able to specify change notification options for version references in a configuration hierarchy in the schema definition. However, these language constructs are part of a data definition language, and their discussion is beyond the scope of this paper.

The commands can be grouped into three categories: intra-database operations, inter-database operations, and name-binding declarations. The effect of an intra-database operation is confined to a single database, while an inter-database operation affects a pair of databases in our database hierarchy. A name-binding declaration is for setting default version for an object or defining a context for dynamic binding. In the next subsections, we discuss the commands in each group.

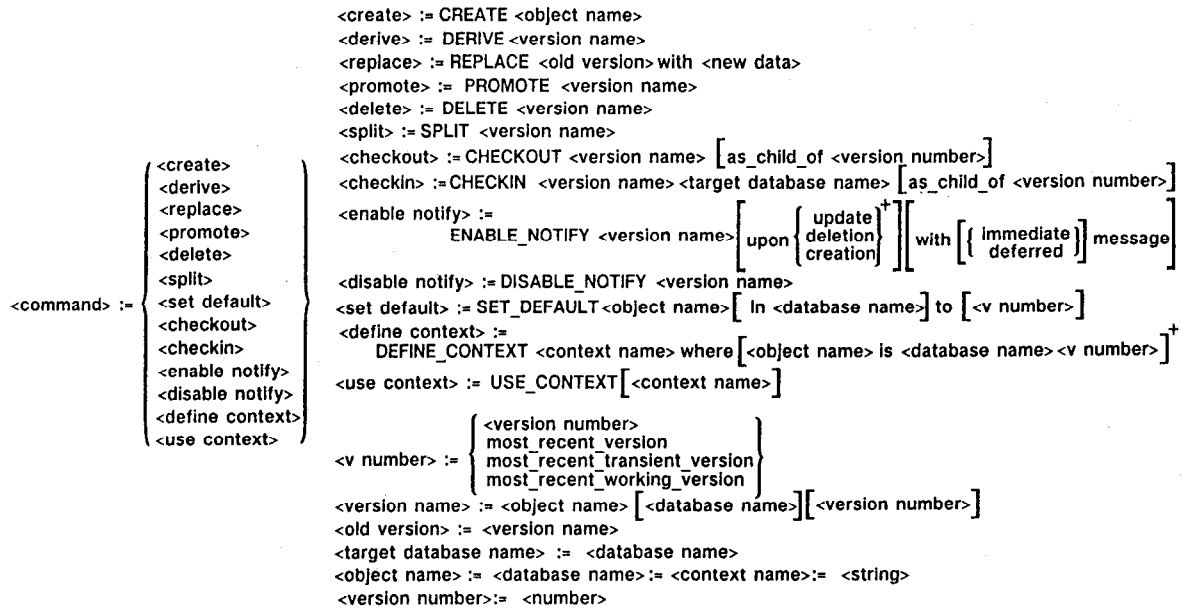

Figure 7. BNF Definition of Application (User) Commands

7.1. Intra-Database Operations

A versioned object is created initially by the *create* command, which sets up the appropriate data structures for the object as described in Section 6.1. The *derive* command is used to derive a new transient version and allocate a new version number for it. The version numbers of the new version and its parent are recorded in the private database. If the parent was a transient version, it is automatically promoted to a working version. The *replace* operation causes the contents of a transient version to be replaced by a workspace copy the user specifies. A transient version is explicitly promoted to a working version, making the version non-updatable, through the *promote* command. A transient version can also be promoted implicitly as a side effect of a checkin.

The user may delete a version or a subtree of a version-derivation hierarchy using the *delete* command. This is a recursive operation that deletes a specified version and all its descendant versions. If the user intends to delete a non-leaf version on a version-derivation hierarchy, he will specify the full three-part name of the version, rather than just the object name. This is to provide a measure of protection against accidental deletions of non-leaf versions (along with their descendants).

The *split* operation establishes a subtree of a version-derivation hierarchy as a new derivation hierarchy. With this operation, a version-derivation hierarchy can potentially become a forest of version-derivation hierarchies. A subtree marked by the split operation can either be deleted, using the delete operation, or exist as a new version-derivation hierarchy. To be consistent with the semantics of versions, none of the intra-database operations are applicable to versions in the public database.

7.2. Inter-Database Operations

The *checkout* command allows a user to check out a version from the public database or a project database, while the *checkin* command allows a user to check a new version into the public or a project database. Both commands provide an option for selecting a parent version in the destination database. For example, a user can check out a version from a project database and install it as a child of a particular version in his private database. If no parent version is specified in the command, the default version on the version-derivation hierarchy is chosen.

The two remaining commands deal with change notification of checked-out versions. The *enable_notify* command results in the insertion of an entry in the inverted reference list of a checked-out version in its originating database. The user can specify the types of events that will post a notification, and whether a message should be sent immediately or delayed till the checkin time. If no options are specified, by default a message will be sent immediately upon deletion of the original version. The *disable_notify* command is used to cancel a previously issued enable-notify command, when the user decides that notification is no longer desired. As we mentioned earlier, change notification options for version references in a configuration hierarchy can be specified in the schema definition, and need not be set (cleared) by the *enable_notify* (*disable_notify*) command.

7.3. Name-Binding Declarations

The user uses the *set_default* command to specify the default version on a version-derivation hierarchy of an object. The *define_context* command provides additional flexibility for establishing alternative (customized) dynamic binding of versions. Once defined, a context can be invoked by the *use_context* command. The binding defined in a context takes precedence over the default binding when the context is in use, and the definition of the context must be searched first to resolve a dynamic binding. The current active context is replaced by another context specified in the next *use_context* command. If no context name is given in a *use_context* command, no context will be active after the command is executed.

Concluding Remarks

In this paper, we presented a model and implementation of version control in a CAD environment. The paper made three contributions. The first is in the development of a model which, unlike most existing models, takes a major step towards incorporating the distributed nature of a CAD environment and the complex configuration of CAD objects. The second is in the detailed exploration of a very broad spectrum of semantic and operational issues in version control, and in unifying the solutions to these issues in a way that is consistent within our model. The third is in our proposal for the implementation and an application interface for the model.

The distributed architecture of an integrated CAD system, and the way in which users of a CAD system interact, led us to logically partition a global database into a database hierarchy which consists of the public database, a set of project databases, and a set of private databases. The public database is sharable among all users, a project database among members of a project, and a private database is accessible to only a single user. This view suggested the notion of different capabilities for versions in different databases, and led us to define and characterize the notions of transient, working, and released versions.

A CAD object is represented as a configuration hierarchy, in which a design is decomposed into a progressively more detailed lower level components. Each node of a configuration hierarchy contains references to lower level components. In general, any component may be referenced by any number of higher level components, and it may itself reference any number of lower level components. A set of CAD objects thus forms a directed acyclic graph of independent, sharable components. When a lower level component is updated, deleted, or a new version of it is created, higher level components that reference it may need to be notified of the changes. The characterization of a CAD database consisting of a set of design objects as a directed-acyclic graph led to our characterization, and solution, of the problem of change notification.

We developed our model of version control by exploring a spectrum of semantic and operational issues, including version naming, static and dynamic binding of versions of a component object to higher level components, versions of schema for CAD objects, and change notification. We also provided discussions of the data structures that a CAD database system must maintain to support our model, and a provisional list of the commands that application programs and users may issue to use our model. We plan to integrate this version model into a prototype database system under construction so that the model can be better evaluated through experimentation and practical experiences.

References

[ATWO85] Atwood, T. "An Object-Oriented DBMS for Design Support Applications," in Proc. IEEE COMPINT 85, pp. 299-307, Sept. 1985, Montreal, Canada.

[BANC85] Bancilhon, F., W. Kim, and H. Korth. "A Model of CAD Transactions," in Proc. Intl Conf. on Very Large Data Bases, August 1985, Stockholm, Sweden.

[BATO85a] Batory, D., and W. Kim. "Modeling Concepts for VLSI CAD Objects," ACM Trans. on Database Systems, vol. 10, no. 3, Sept. 1985.

[BATO85b] Batory, D., and W. Kim. "Supporting Versions of VLSI CAD Objects," to appear in IEEE Trans. on Software Engineering.

[DADA84] Dadam, P., V. Lum, and H. Werner. "Integration of Time Versions into a Relational Database System," in Proc. Intl. Conf. on Very Large Databases, August 1984, pp. 509-522.

[DITT85] Dittrich K. and R. Lorie. "Version Support for Engineering Database Systems," IBM Research Report: RJ4769, IBM Research, Calif., July 1985.

[EDIF] Electronic Design Interchange Format, preliminary specification, version 0.8

[HASK82] Haskin, R. and R. Lorie. "On Extending the Functions of a Relational Database System," in Proc. ACM SIGMOD Conf., June 1982, pp. 207-212.

[HAYN84] Haynie, M. and C. Gohl. "Revision Relations: Maintaining Revision History Information," IEEE Database Engineering bulletin, vol. 7, no. 2, June 1984, pp. 26-33.

[KAIS82] Kaiser, G. and A. Habermann. "An Environment for System Version Control," Tech Report, Dept. of Computer Science, Carnegie-Mellon University, November 1982.

[KATZ84a] Katz, R. and T. Lehman. "Database Support for Versions and Alternatives of Large Design Files," IEEE Trans. on Software Engineering, vol. SE-10, no. 2, March 1984, pp. 191-200.

[KATZ84b] Katz, R. and S. Weiss. "Design Transaction Management," in Proc. ACM/IEEE 21st Design Automation Conf., Alburquerque, NM, June 1984.

[KATZ85] Katz, R. private communication, March 1985.

[KATZ86] Katz R., E. Chang, and R. Bhateja. "Version Modeling Concepts for Computer-Aided Design Databases," Submitted to ACM SIGMOD Intl. Conf. on Management of Data, 1986.

[KIM84] Kim, W., R. Lorie, D. McNabb, and W. Plouffe. "A Transaction Mechanism for Engineering Design Databases," in Proc. Intl. Conf. on Very Large Data Bases, August 1984.

[LORI83] Lorie, R. and W. Plouffe. "Complex Objects and Their Use in Design Transactions," in Proc. Databases for Engineering Applications, Database Week 1983 (ACM), May 1983, pp. 115-121.

[McLE83] McLeod, D., K. Narayanaswamy, and K. Bapa Rao. "An Approach to Information Management for CAD/VLSI Applications," in Proc. Databases for Engineering Applications, Database Week 1983 (ACM), May 1983, pp. 39-50.

[NEUM82] Neumann, T., and C. Hornung. "Consistency and Transactions in CAD Databases," in Proc. Intl Conf. on Very Large Data Bases, Sept. 82, Mexico City, Mexico.

[NEWT85] Newton, A.R. private communication, March 1985.

[PLOU83] Plouffe, W., W. Kim, R. Lorie, and D. McNabb. "Versions in an Engineering Database System," IBM Research Report: RJ4085, IBM Research, Calif., October 1983.

[ROCH75] Rochkind M. "The Source Code Control System," IEEE Transactions on Software Engineering, vol. SE-1, no. 4, December 1975, pp. 364-370.

[TICH82] Tichy W. "Design, Implementation, and Evaluation of a Revision Control System," IEEE 6th International Conference on Software Engineering, September 1982.

[WIED82] Wiederhold, G, A. Beetem, and G. Short. "A Database Approach to Communication in VLSI Design," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. CAD-1, no. 2, April 1982, pp. 57-63.

[WOEL86] Woelk, D., W. Kim, and W. Luther. "An Object-Oriented Approach to Multimedia Databases," ACM SIGMOD Intl. Conf. on Management of Data, 1986.