SUPPORT FOR VERSIONS OF VLSI CAD OBJECTS

D. S. Batory & Won Kim

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-18 September 1985

Support for Versions of VLSI CAD Objects

D.S. Batory
Dept. of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

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

ABSTRACT

Version control is an essential requirement for VLSI CAD, and has at least three important implications for support in database systems: (1) versions of design objects are usually created from existing versions by making relatively minor changes; thus the sets of records that define different versions are largely redundant. Storage sharing among versions should be supported. (2) Several versions, not just the most recent one, will be accessed. All versions should have uniformily low access times. (3) Changes made to versions may affect the validity of versions of other related design objects. Such changes must be monitored and affected designs should be notified.

In this paper, we describe a storage technique that balances the requirements for reducing storage redundancy and fast access to different versions of the same design object. We also identify specific problems of change monitoring and notification, and propose solutions.

1. Introduction

Many researchers have addressed aspects of the problem of version control, both for CAD applications and software development [KATZ82, KAIS82, McLE83, PLOU83, HAYN84, DADA84]. In spite of such efforts, many of the major issues are still not well-understood or clearly defined. Indeed, it is not even clear from earlier work what is meant by *version*, and what aspects of version control should be supported by database systems and what by users.

In a recent paper, we have proposed a framework for modeling VLSI CAD objects [BATO85]. Informally, versions of a design object share the same interface and behavioral characteristics, but differ in implementation details. Versions are usually created from existing

versions by making relatively minor changes. As a consequence, the sets of records that define different versions are largely redundant. It is therefore important to have a storage structure that promotes record sharing among versions.

At the same time, minimizing storage requirements should not incur a significant penalty for access times to versions. Several versions of the same design object may be accessed with similar frequency. For example, a designer may be interested in the most recent version of a design object, while a tester may be validating an earlier version. Thus it is important that a storage structure be used which allows efficient access to any specific version.

Monitoring changes in versions and notifying designers or affected designs (of possibly different design objects) has long been an important issue in VLSI CAD. Certain types of changes may be of interest only to some designers; other types of changes will affect the validity of versions that utilize the changed version as a component. However, the issues involved in change monitoring and notification are still poorly understood and no comprehensive framework for identifying problems and their solutions exists as yet.

In this paper, we present a version storage structure and change monitoring and notification mechanisms which address the above problems for VLSI CAD database systems. In Section 2 we review concepts for modeling the structural properties of VLSI CAD objects and their versions. Section 3 presents our version storage structure and access algorithms. Issues in change monitoring and notification are examined and solutions are proposed in Section 4.

2. VLSI CAD Objects and Their Versions

A VLSI CAD object manifests itself in a number of representations: for example, register transfer, Boolean, logic, circuit, layout, etc. [ULLM84]. Lower-level (more detailed) representations are usually derived from higher-level representations. For example, circuit representations are generated from logic representations and layout representations are generated from circuit representations. Designers often experiment with versions of a particular representation of a

design object. Fig. 1 shows three representations of a design object. Representation A has two versions, B three versions, and C only one version. Versions v-1 and v-2 of B were derived from version v-1 of A, while v-3 of B was derived from v-2 of A, as indicated by directed arcs.

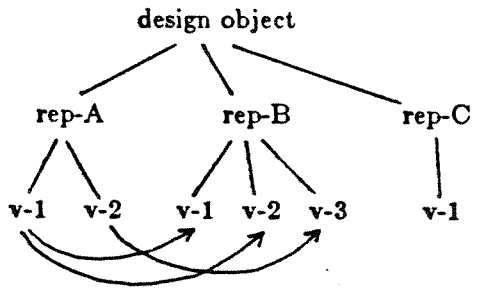

Fig. 1 Representations and their versions

The structural properties of a design object outlined below are applicable to any of the representations. For concreteness, however, we will illustrate the concepts in the context of circuit representations. The description of a CAD object (circuit) consists of two parts: its interface and implementation [EDIF84, McLE83]. The *interface* of a circuit contains inputs and outputs, and specifies the function (behavioral characteristics) 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. Thus circuit description, as well as the design itself, is often hierarchical; it enables implementation details to be developed or revealed in a progressive manner.

Consider a circuit for a 4-bit adder. Fig. 2 shows its interface specification: a pair of 4-bit numbers (X,Y) are input and a 5-bit number representing their sum (Z) is output. Details about the circuit's internals are not shown, but are specified in the circuit's implementation.

One possible implementation of an adder is shown in Fig. 3. It is realized as a ripple-carry through four adder-slice circuits. According to its interface, an adder-slice takes a pair of 1-bit numbers (X_i , Y_i) and the carry from a previous slice (C_{i-1})and produces their 1-bit sum (Z_i) and carry (C_i).

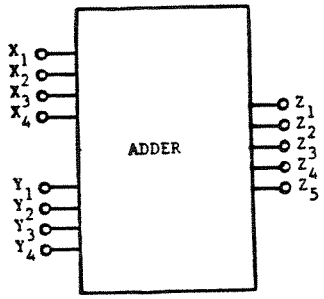

Fig. 2 Interface of a 4-bit Adder

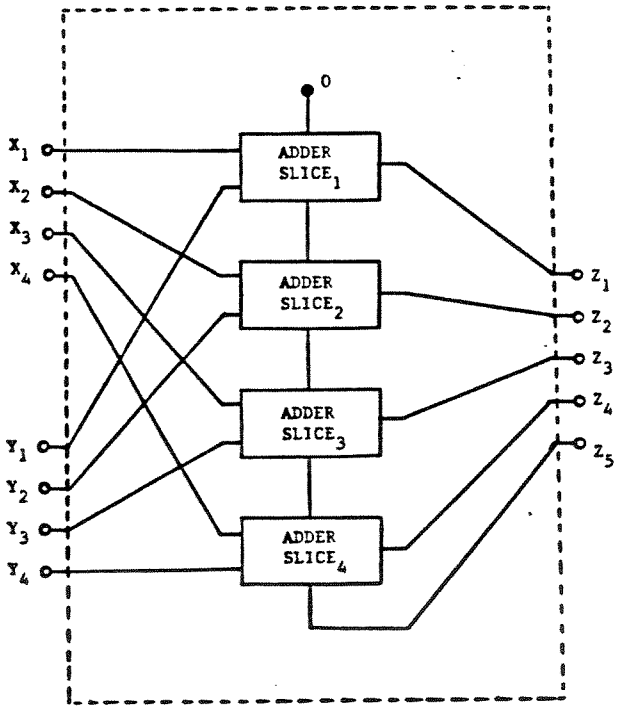

Fig. 3 An Implementation of a 4-bit Adder

From our definition of a design object as consisting of an interface description and an implementation description, we define *versions* to be objects that share the same interface but have different implementations. A term often mentioned in the context of version control is *design alternative* [KATZ82]. Currently there appear to be at least two interpretations of this term. One is to make no distinction; versions and design alternatives are merely different names for implementations of a design object. The other is to classify implementations of a design object

by implementation strategies, optimizing criteria (area, power consumption, speed, etc.), technology (nMOS, cMOS, etc.), and so forth. Implementations within a single class are called versions, and those in different classes are design alternatives.

For the purposes of this paper, it is not important which interpretation will eventually prevail. This issue is orthogonal to the problems of storage structure and change notification in version control, and for expository simplicity, we will make no distinction between versions and design alternatives in this paper.

Versioned objects are said to be occurrences of a single design object (or object interface). Fig. 4 illustrates the general relationship between an object and its versions. Three objects are shown. Object A has three implementations (versions), object B has none (no implementation for it has yet been specified), and object C has two.

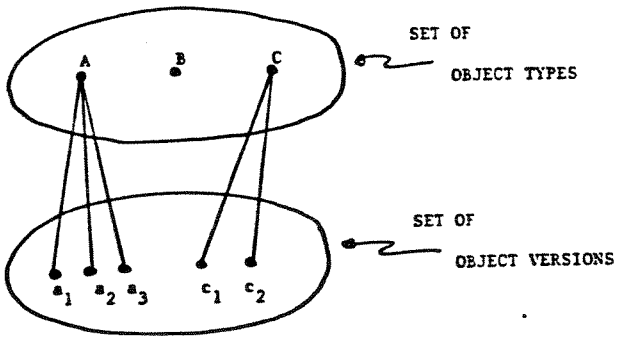

Fig. 4 Relationship Between Objects and Their Versions

The relationship between objects and their versions, called *version generalization*, has special properties. First, there is the notion of attribute inheritance; all attributes of an object interface are inherited by its versions. Second, objects have update restrictions. Modifying the interface of an object results in a new object; the original object remains unchanged. In addition, versions of the original object are not considered versions of the new object.

Another concept which is essential for modeling objects is *instantiation*. Fig. 3 is an example. An adder uses four copies, or *instances*, of the adder-slice circuit. Each instance is distinct (i.e., each has its own separate set of inputs, outputs, and coordinate positions on a circuit

diagram), yet each shares the same adder-slice features. Instantiation distinguishes an object or its version from its copies. Copies are simply reproductions of a master; the copies themselves are *not* versions.

Instantiation, like version generalization, involves attribute inheritance. An instance of an object inherits all attributes of that object. Instances also have attributes that are not inherited (e.g., coordinate position of the instance on a circuit diagram, input and output labels that are specific to the instance). It is the non-inherited attributes which enable different instances to be distinguished.

Instantiation is applicable to both object interfaces and versions. If an object interface is instantiated, no implementation of the object is specified; only the interface is copied. If a version is instantiated, both the interface and its implementation are copied. An instance of a version inherits the attributes of both the object interface and its implementation; an instance of an object interface just inherits the interface attributes of the object. Fig. 5 shows the attribute inheritance graph relating object interfaces, versions, and their instances.

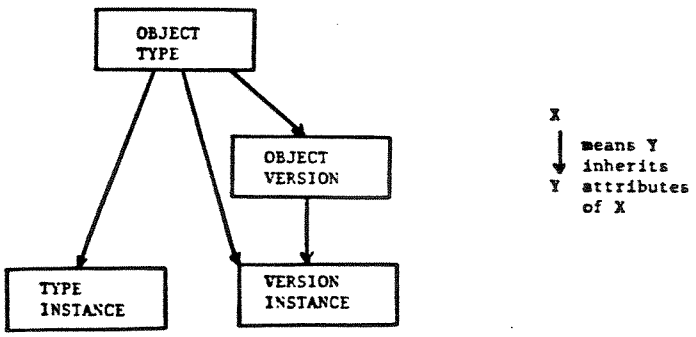

Fig. 5 Attribute Inheritance Graph for Object Interfaces, Versions, and Instances

There is an important semantic difference between instances of object interface and instances of versions. Let V be a version of an object T and let X be a version of some other object. Whenever an instance of V is used in the implementation of X, V is an inherent or fixed part of X's design. However, when an instance of T is used instead, this creates a "socket" or "template" in X in which *any* version of T may be placed. This gives rise to the concept of

parameterized versions.

Parameterized versions can be understood as templates. Each socket can be plugged with versions (parameterized or non-parameterized) of a specified object interface. Parameterization naturally supports hierarchical relationships among version instances. Fig. 6 shows an instance of object version A which has two parameters; one is filled with object version B (which itself is parameterized) and the other contains object version C (which is not parameterized). B has three parameters; the first two are filled with non-parameterized versions D and E, and the remaining parameter is unplugged.

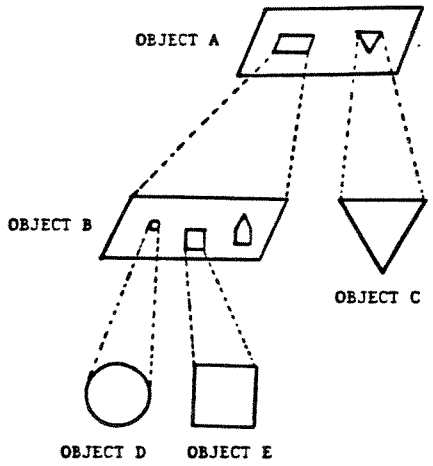

Fig. 6 Hierarchical Relationship Among Object Instances

Consider a more concrete example. Fig. 3 shows an implementation of an adder interface which utilizes four instances of an adder-slice. Let A (as_1, as_2, as_3, as_4) denote this version, where as_1 ... as_4 are the parameters of type adder-slice. Suppose there are two unparameterized versions of an adder-slice, S1 and S2. By plugging in S1 or S2 for each of the parameters of A, we can define a spectrum of versions {A(S1,S1,S1,S1) ... A(S2,S2,S2,S2)} of an adder.

3. A Storage Structure for Versions

As mentioned in the Introduction, it is often the case that there is a substantial similarity between a new version and an older version; many of the same records are shared. Other researchers have recognized this (e.g., [KATZ82, HAYN84, DADA84]) and have proposed storage

structures based on differential file techniques [SEVE76]. Records belonging to, for example, the most recent version are stored in one file, while records of all other versions are maintained in differential files. An important shortcoming of these techniques is that they support efficient access to only one version, usually the most recent. Accessing other versions can be very slow and complicated.

Adopting a storage structure that favors access to one version over all others is not necessarily a good idea. For example, a designer may be interested in the most recent version of a design, while a tester may be validating an earlier version. These considerations led to the proposal in [PLOU83] not to support storage sharing among versions.

In the following sections, we describe a storage structure for versions which is based on differential file techniques but is quite different from previous proposals. Our structure has the following features: (1) It allows the creation of versions from any existing version with minimal replication of records, thereby reducing storage redundancy. (2) It can provide nearly uniform access times to any specific version.

3.1 The Basic Storage Structure

A version of a design object can be created in one of two basic ways. Either it is created from scratch or by modifying an existing version of the same design object.[1]

The history of version creations for a design object can be captured in one or more directed trees, which we will call ***version-derivation hierarchies.*** The root of a version-derivation hierarchy (tree) is a version which was created from scratch; all other versions are descendants of the root version. Fig. 7 illustrates a version-derivation hierarchy. Versions v-1, v-2 and v-3 were derived from v-0; v-4 and v-5 from v-2, and so on. When a version v-j is derived from v-i, we say v-i is the *parent version* of v-j, and v-j is a *child version* of v-i. The meaning of *ancestor versions, des-*

---

[1] Two other ways are known, but they too can be cast into one of the above basic ways. The first additional way is to modify a version of a different design object. This can be treated as if the version was created from scratch. The second way to create a version is by modifying two or more existing versions. In this case, only one of the old versions is selected as the parent of the new version.

cendant versions, *leaf versions*, and *non-leaf versions* follows naturally.

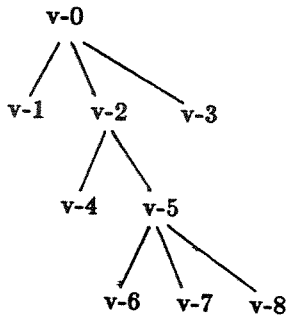

Fig. 7 A version-derivation hierarchy

Version-derivation hierarchies are important because they suggest which versions are most likely to share the greatest number of records. Intuitively, the chances are greater that two versions will share more records if one is a direct ancestor of the other than if they were not related at all. Thus, version-derivation hierarchies provide an important clue to storage sharing among versions.

Another clue to designing a version storage structure concerns the modifiability of versions. Versions can be classified as either working or released. A *released version* is not subject to updates, while a *working version* may be updated at any time. Thus it is possible for a working version to be updated before and after new versions have been generated from it.[2] A version storage structure, therefore, should allow both released and working versions to coexist in a single hierarchy and should permit any working version to be modified.

The above clues suggest that a single storage structure be used to store the set of versions that belong to a single version-derivation hierarchy. If multiple hierarchies describe the versions of a design object, multiple instances of the storage structure would be used.

[2] We are aware of unpublished suggestions by others that working versions should not be updated once new versions have been generated from them. This would result in creating a new version each time a change (however trivial) is made. Although we feel this approach leads to undesirable consequences, we note that our storage structure will support it.

To understand how the storage structure would work, consider what happens when a new version is created. All records that the new version shares with its parent would (obviously) be already stored. The only records that need to be stored are those that are not present in the parent version. We will say that the records that a child version shares with its parent are *inherited*. A record that is not inherited is said to be *owned* by that version. With these ideas in mind, the following is a list of rules that characterize the version storage structure:

1. All records of the root version are owned by the root version; some records of each non-root version are owned by that version and others are inherited from ancestor versions.

2. When a record is inserted into a version v-i, the record is owned by v-i and the record is visible to all versions that are subsequently derived (created) from v-i. The record is not visible to any other versions.

3. When a record is deleted from a non-leaf version, the original record must continue to be visible to descendant versions that inherited it.

4. When a record is updated in a non-leaf version, all descendant versions that inherited the record must continue to see the original record.

5. All records owned by a version should be physically clustered so that they can be efficiently accessed together.

6. The accessing of the records of version v-i is accomplished by reading the records owned by the versions that appear along the chain from the root version to v-i in the version-derivation hierarchy.

To support the above, every record of a version must be augmented with a bitmap and a field that identifies which version owns the record.[3] Each bit in the bitmap corresponds to a version on the version-derivation hierarchy. The assignment of versions to bits will be called the bitmap *legend*. A zero-bit means that the record is *potentially* visible to the corresponding version, and a one-bit means that it is not. (As we will see shortly, it is possible for a record not to be inherited by (i.e., visible to) version v-i yet its v-i bit is zero). For simplicity, we will assume that the bitmap has a fixed size, which is equal to the maximum number of versions that can be on a version-derivation hierarchy. (This is not a serious limitation; in Section 3.2 we will show how to work around it).

[3] The owner field is needed as it is not always possible to determine the owner of a record simply by inspecting the record's bitmap.

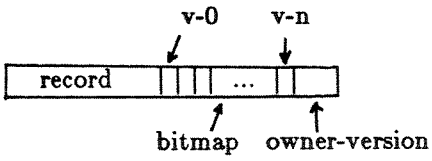

Fig. 8 An augmented record

Additional support is needed in the form of a *version directory*. This directory will encode the parent-child relationships of each version-derivation hierarchy and will have an entry for every version. Stored with each entry is a pointer to a chain of one or more storage blocks that contain the records owned by that version. As these blocks are allocated when a version is created, they can be assigned sequential locations. Subsequent record insertions and updates may cause additional blocks to be allocated. The new blocks probably will not be physically contiguous to the last of the sequential blocks.

Before presenting detailed descriptions of our algorithms for retrieving and manipulating records from a version, we will provide an intuitive description of them. Our algorithms are based on two requirements. First, it is essential that when a new version is created, the records of its ancestor versions should NOT be updated to reflect their visibility (invisibility) to that version. That is, the inheritance (noninheritance) of records should be automatic. This means, for example, that when a record is inserted into version v-i, its bitmap entries for all versions that don't yet exist must be set to 0 so that new versions can be presumed to inherit the record. (Note that the bitmap entry for v-i is also set to 0 while entries for versions that already exist are set to 1).

The second requirement is that a version v-i inherits record r from ancestor version v-j only if v-j owns r and no version along the path from v-j to v-i deleted r. That is, the bits of the bitmap of r that correspond to versions on the path starting with v-j and ending at v-i must be all zero. We will refer to this as the *clear path* requirement.

Both requirements are needed for efficient performance; the first requirement is obvious, but the second may not be. Suppose we dropped the need for a clear path. Assume for simplicity that the maximum number of versions on a version-derivation hierarchy is 3. Suppose that there

is only one version, v-0, and a record r is inserted into it. The bitmap in the record is [0,0,0] for versions v-0, v-1, and v-2, respectively, where v-1 and v-2 do not yet exist.

Now version v-1 is created from v-0, inheriting record r. Then version v-1 deletes record r. The bitmap of r would contain [0,1,0]. Version v-2 is created from v-1. Since v-1 deleted r, the record must not be visible to v-2. However, since a clear path from v-0 (the owner) to v-2 is not required and that the bitmap entry of r corresponding to v-2 is 0, it is necessary for that bit to be set to one. This violates our first requirement that the inheritance (or noninheritance) of records be automatic and incur no overhead. Thus, clear paths are required.

Surprisingly, however, these simple requirements introduce some unpleasant complications in the algorithms for the update and deletion of records. Consider a different example. Again suppose v-0 has one record r. Its bitmap is [0,0,0]. Version v-1 is created from v-0, inheriting r, and version v-2 is created from v-1, also inheriting r. Now v-1 deletes r. Record r must still be visible to v-0 and v-2. Because of the clear path requirement, it is necessary to insert a copy of r into v-2, with v-2 as the owner, while keeping the original record owned by v-0. The bitmap of both copies of r is [0,1,0]. Record replication would have been required if v-1 instead updated r. There would then be three instances of record r: two copies of the original (one owned by v-0, the other by v-2), and the updated r (owned by v-1).

Clearly, replicating records compromises our objective of minimizing storage redundancy among versions. However, we note that records need to be replicated only when they are deleted or updated from a non-leaf version where one or more of its descendants inherited the records. Further, when a potentially large number of records are to be deleted or updated from a non-leaf version, the user can simply create a new version thereby avoiding record replication. Thus, we believe that our approach will have only a negligible impact on minimizing storage sharing.

We now describe our algorithms for inserting, deleting, updating and retrieving records in a version.

Algorithm_Insert:

Assume that record r is to be inserted into version v-i.

1. [initialize bitmap] Set the bitmap entries for all versions on the version-derivation hierarchy (except v-i) to 1. Bit v-i and all other bits (that do not correspond to existing versions) are 0.

2. [initialize owner] Set v-i as the owner of record r.

Note: Step 1 means that record r is visible only to v-i and to no other currently existing versions.

Step 2 ensures that record r is automatically inherited by versions that are subsequently derived from v-i.

Algorithm_Delete:

Assume that record r is to be deleted from version v-i.

1. [logical deletion] Set bit v-i in the bitmap of r to 1.

2. [replication] Let D be the set of child versions of v-i that inherit r. Insert a copy of r (with the bitmap as updated in Step 1) into each version in D, with that version as the owner.

3. [physical deletion]. If v-i is the owner of r, physically delete the record.

Note: Step 1 makes r invisible to v-i. By the clear path requirement, r is also made invisible to the descendants of v-i.

Step 2 replicates r for each child version of v-i that inherited r from v-i. A copy of r will be owned by each version in D. All instances (the original and its copies) will share the same bitmap, and thus preserve the inheritance of r by descendant versions.

Step 3 physically deletes r if no version inherits r.

Algorithm_Update:

Assume that record r of version v-i is to be updated.

1. [replication] Let D be the set of child versions of v-i that inherited r. Insert a copy of r into each version in D with that version as the owner. (The bitmaps of the copies should be identical to the original to preserve inheritance information).

2. [physical update] If v-i is the owner of r, update r. Also, set to one the bits corresponding to D, as defined in Step 1. Quit.

3. [logical update] Otherwise, set the bit for v-i to 1 in r. Insert, using Algorithm_Insert, the updated copy of r into v-i with v-i as the owner. Quit.

Note: Step 1 ensures that versions that inherited the record see its original state.

Step 2 physically updates r if r was not inherited from v-i's ancestors. It also ensures the

updated copy is not seen by versions that inherited the non-updated record.

Step 3 inserts an updated copy of r into v-i when the owner of r is an ancestor. Version v-i no longer sees the original copy but does see the updated r. Ancestor versions continue to see the original.

Algorithm_Retrieve:

Assume the records of version v-i are to be retrieved.

1. [version scan] Let C be the set of versions that form a chain from the root version to v-i. Using the version directory, scan the set of records owned by each version in C.

2. [qualification] A record r belongs to version v-i if:

    1. the v-i bit in the bitmap of r is 0, and

    2. if v-i does not own the record, then the bits of the versions that form the path beginning with the owner version to version v-i must be all 0 (i.e., the clear path requirement).

Note: Step 1 eliminates the examination of versions whose records v-i could not have inherited.

Step 2 eliminates a record if it is known that an ancestor of v-i (and a descendant of the owner version) deleted the record.

A proof of correctness for the above algorithms is given in the appendix.

We illustrate these algorithms below. Suppose that the maximum number of versions on a version-derivation hierarchy is 4.

Version v-0 is the first version created, and it owns the three records shown in Fig. 9a; that is, these records are inserted into v-0. Note that bitmap entries for each of these records are all set to 0, indicating that versions to be derived from v-0 can inherit these records.

Next version v-1 is derived from v-0 by inheriting Record 1 and Record 2, and inserting a new record Record 4. This requires deletion of Record 3 with respect to version v-1, and insertion of Record 4 into v-1. The resulting state of bitmaps in the 4 records is shown in Fig. 9b.

Then version v-0 inserts a new record Record 5. Since v-1 has already been derived from v-0, the insertion of this new record should be transparent to v-1. However, versions yet to be derived from v-0 should be able to inherit this record. Therefore, only the bit corresponding to

v-1 is set to 1 in Record 5, as shown in Fig. 9c.

Now the designer of v-0 deletes Record 2, which v-1 had inherited. The record should continue to be visible to v-1; however, it should not be visible to v-0. The resulting bitmap for Record 2 is shown in Fig. 9d. Note that Record 2 was physically deleted and re-inserted with v-1 as its owner.

Next, a new version v-2 is derived from v-1. The records it inherits are Record 1, Record 2, and Record 4. Now suppose Record 1 is deleted from v-1. Record 1 should now be visible to v-0 and v-2. Record 1 is now duplicated, as shown in Fig. 9e. The bitmap in Record 1 indicates that versions other than the current descendants of v-0 may inherit the record. Record 1' (a copy of Record 1) is owned by v-2, and versions to be derived from v-2 may inherit it.

a)

Record 1	|0|0|0|0|	v0
Record 2	|0|0|0|0|	v0
Record 3	|0|0|0|0|	v0

b)

Record 1	|0|0|0|0|	v0
Record 2	|0|0|0|0|	v0
Record 3	|0|1|0|0|	v0
Record 4	|1|0|0|0|	v1

c)

Record 1	|0|0|0|0|	v0
Record 2	|0|0|0|0|	v0
Record 3	|0|1|0|0|	v0
Record 4	|1|0|0|0|	v1
Record 5	|0|1|0|0|	v0

d)

Record 1	|0|0|0|0|	v0
Record 2	|1|0|0|0|	v1
Record 3	|0|1|0|0|	v0
Record 4	|1|0|0|0|	v1
Record 5	|0|1|0|0|	v0

e)

Record 1	|0|1|0|0|	v0
Record 1'	|0|1|0|0|	v2
Record 2	|1|0|0|0|	v1
Record 3	|0|1|0|0|	v0
Record 4	|1|0|0|0|	v1
Record 5	|0|1|0|0|	v0

Fig. 9 An example of insertion and deletion of records

The algorithms presented in this section are the building blocks for such higher level version operations as creation, deletion, and replacement of versions. To delete a leaf version, all the records owned by the version are physically deleted. For a non-leaf version, Algorithm_Delete is used to delete both the records the version owns and those it inherited. To create a new version, Algorithm_Insert is used to insert records the version will own and Algorithm_Delete to suppress

the inheritance of selected records from the ancestor version. Replacing a leaf version can be cast into deletion of the version followed by creation of a new version. To replace a non-leaf version, Algorithm_Delete is used to delete records that the new version will not see, and Algorithm_Insert to insert both updated records and new records.

To gain some insight into the performance of these algorithms, let n be the average number of records in a version and let u be the average fraction of records in a child version that are not inherited from the parent version. Suppose there are k versions. If k versions were stored separately (as is done in [PLOU83]), S(k) = k*n records are stored. If k versions are stored using our structure, VS(k) = n + (k-1)*u*n records will be stored. The ratio VS(k)/S(k) = 1 + (k-1)*n would be the compression factor. It is believed that a typical value of u is .2. That is, 80% of the records of a child version are inherited from its parent. For k=5 the compression factor is .4 (i.e., a savings of 60%). For larger k, the savings would be even greater.

An overhead of the version storage structure is that the retrieval of records belonging to a specific version requires a traversal of all of its ancestors to find inherited records. Thus, the farther a version is from the root, the greater the overhead. Suppose a version is at level m in the version-derivation hierarchy (m=0 is the root). The average number of records that will be retrieved in accessing the version is VA(m) = n + m*u*n. (n records for the root, u*n for the level 1 descendant, u*n for the level 2 descendant, etc.). If versions were stored separately, the number of records accessed would be A = n. The ratio VA(m)/A = 1 + m*u is the fraction overhead. Again with u=.2 and taking m=4, the fraction overhead is 1.8, or less than twice the cost of accessing a separately stored version.

We see from these simple calculations that the version storage structure is likely to save a considerable amount of storage. However, for it to support quick access times, the version-derivation hierarchies should be short in height, and preferably dense. In the case that hierarchies become rather high, an alternative approach is needed. One such approach is the use of version segments, discussed in Section 3.2. Another approach is discussed in Section 3.3.

3.2 Version Segments

Thus far we have assumed, for simplicity, that all versions on a version-derivation hierarchy inherit records from their ancestors. However, as the version-derivation hierarchy grows, it becomes advantageous to partition it into a number of independent version-derivation hierarchies. One reason, as discussed in the last section, is that the farther a version is from the root version in a version-derivation hierarchy, the greater the overhead for accessing its records. This is compounded by the reality that the greater the distance from the root, the fewer records the version is expected to share with its ancestor versions closer to the root.

A second reason for partitioning a version-derivation hierarchy is to overcome the size limitation in the bitmap within a record. That is, when there are more than N versions on a version-derivation hierarchy, and the size of the bitmap is fixed at N, the version-derivation hierarchy will have to be partitioned into more than one independent hierarchies.

It is with these observations that we introduce the notion of version segments. A *version segment* is a connected subtree of versions on the version-derivation hierarchy. Fig. 10 shows version segments VS-1, VS-2, and VS-3 defined on the version-derivation hierarchy of Fig. 7. VS-1 consists of versions v-0, v-1, v-2, and v-4; VS-2 has only v-3; and VS-3 contains v-5, v-6, v-7, and v-8.

Although we recognize that it is conceivable for two version segments to share some versions, we will assume, for expository simplicity, that a version segment of a given version-derivation hierarchy does not overlap with any other version segment. Further, we note that it may be difficult and potentially expensive to have the database system determine an "optimal" set of version segments from a given version-derivation hierarchy. As such, we expect that the users will have to specify the formation of version segments.

Two algorithms are required to support version segments. Algorithm_Split is needed to split off a version segment from an existing version segment; and Algorithm_Merge merges a version segment back to the version segment from which it was split off. We assume that both the new

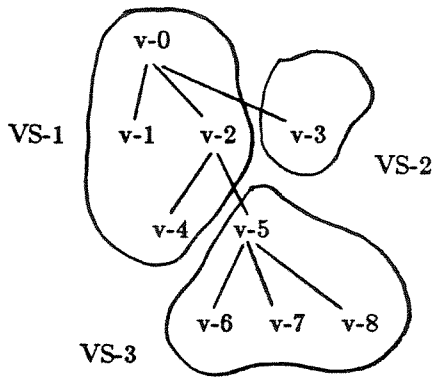

Fig. 10 Version Segments

version segment and the version segment from which it is split off reside in the same physical area (e.g., same disk volume). As before, we will assume the existence of a version directory for each version segment.

Algorithm_Split:

Assume that a new version segment VS-new, with v-new as its root version, is to be split off an existing version segment VS-current with v-current as its root. Assume that v-new is different from v-current.

1. [version scan] Let C be the set of versions that form a chain from the root version to v-new, ***but not including v-new***. Using the version directory, scan the set of records owned by each version in C.

2. [qualification] A record r is inherited by version v-new if
    1. the v-new bit in the bitmap of r is 0, and

    2. the bits of the versions that form the path beginning with the owner version to version v-new are all 0.

3. [insertion] For each record r that is inherited by v-new, insert r, using Algorithm_Insert, into v-new with v-new as its owner.

4. [redefine legends] Optionally, construct a new assignment of the bitmap positions to versions in VS-current and VS-new. Update the legends of the records in VS-current and VS-new by scanning the records in each version segment and updating them.

    Note: Steps 1 and 2 are almost identical to Algorithm_Retrieve, which identifies all records inherited by v-new in VS-current.

    Step 3 eliminates the need to examine ancestor versions of v-new to locate records of v-new.

    Step 4 redefines the assignment of the bitmap positions to versions belonging to each version segment. This may be necessary to accommodate many versions with a fixed-size bitmap.

Algorithm_Merge:

Assume that a version segment, VS-old, with root v-old, is to be merged as a child of version v-parent of version segment VS-current.

1. [augment legend] Augment the legend of VS-current by assigning bits for each of the versions in VS-old.

2. [determine inheritance] Using Algorithm_Retrieve, retrieve the records that are visible to v-parent. Let this set of records be P. Retrieve the set O of records that define v-old. Let I be the set of records that v-old would inherit from v-parent; that is, I = P intersect O. Let NI be the set of records that v-old would not inherit; that is, NI = P difference I.

3. [physical deletion] Physically delete each record in I from version v-old, without replicating it for child versions of v-old. Copy the inheritance information in the bit maps of these records to their corresponding records in VS-current.

4. [register noninheritance] For each record in NI, set the v-old bit to 1.

Note: Step 1 enlarges the legend of VS-current by the versions in VS-old.

Step 2 identifies the records that are not inherited by v-old from v-parent.

Step 3 removes records from v-old that will be inherited from v-parent in the new version segment VS-current. To preserve the inheritance information of these records, the bitmaps of the deleted records are recorded in their corresponding records of VS-current. In this way, both the records and their inheritance information is preserved; only the owner information is changed.

Step 4 makes selected v-parent records non-inheritable by by setting the inheritance bit of v-old.

We have also developed variations of Algorithm_Split and Algorithm_Merge that assign a different physical storage area for a version segment that are split off another version segment. They are only slightly more complex than the above algorithms and will not be presented here.

3.3 Further Extensions

In earlier works ([KATZ82],[DADA84]), the history of version creations was not modeled by version-derivation hierarchies as we have done, but rather as a linear chain or *time-series* of versions. Although our approach generalizes these works, some of the concepts that they propose are directly applicable to our version storage structure.

[DADA84] provides a brief discussion of two basic ways of applying the principles of differential file techniques to promote storage sharing in the context of supporting time-series versions. It explains these ways in terms of *delta files*, where a delta file D(V1,V2) contains the

differences between version V1 and V2, and *complete files*, which contain all the records associated with a single version. One of the basic ways is the *forward differential-file technique*, in which the oldest version, v-0, is stored in a complete file and the sequence of subsequent versions are stored in delta files. Version v-i is reconstructed by reading the complete file of v-0, and the delta files D(v-0,v-1), D(v-1,v-2), ..., D(v-(i-1),v-i). The other way is the *backward differential-file technique*, in which the newest version, v-n, is maintained as a complete file and the sequence of earlier versions are stored in delta files. Version v-i is reconstructed by reading the complete file of v-n and the delta files D(v-n,v-(n-1)), ..., D(v-(i+1),v-i).

Both techniques have a common variation. For the forward technique, the delta files D(v-0,v-j) for j=1..n are stored. Thus version v-i is reconstructed by reading the complete file of v-0 and the delta file D(v-0,v-i). For the backward technique, the delta files D(v-n,v-j) for j=0..n-1 are stored. From the perspective of access efficiency, this variation is better than the original techniques. However, it is worth noting that the degree of storage sharing should decrease; the similarity of versions v-i and v-(i+1) is likely to be greater than that between versions v-0 and v-i or v-n and v-i.

Our storage structure is based on forward-differential file techniques. As mentioned earlier, we noted that the farther a version is from the root of its version-derivation hierarchy, the greater the overhead will be to access it. We introduced version segments as a way to minimize version retrieval times. The variation of the forward-differential technique described above is another way to improve access efficiency.

When this variation is applied to our storage structure, the semantics of the version-derivation hierarchy changes slightly (i.e., a child version of the root need not have been derived directly from the root). However, this does not affect the algorithms of Section 3.1; they will work regardless of the true ancestry of versions. It is worth noting, however, that if version v-i was derived from v-k, and v-i is to be made a child of v-a, an ancestor of v-k, the records that v-i inherited that are not visible to v-a will need to be duplicated (and owned) by v-i. We emphasize that, unlike the variation considered in [DADA84], which implies that version v-i, being derived

from version v-k, must become a child of the root version v-0, we allow v-i to become a child of any ancestor of v-k.

We conclude this section by noting that at first glance a version storage structure based on backward-differential file techniques seems appealing in that the most recent versions are stored in complete files. However, it does not appear possible to convert our storage structure and algorithms so that they are based on a backward approach. The difficulty lies in supporting record inheritance and maximizing storage efficiency. For example, suppose version v-0 is created. Version v-1 is derived from v-0, and so too is v-2. All three versions contain the same set of records. v-1 and v-2 will be stored in complete files; v-0 would be represented by an empty delta file. Since the records in the v-1 and v-2 files are duplicated, it is not clear what is gained by this approach. We note that there are additional difficulties which make the backward approach even more unattractive than this example indicates.

4. Change Notification

In this section we first identify issues involved in supporting the automatic monitoring of changes in a version and subsequent change notification to affected versions. We then propose techniques for monitoring and notifying changes.

4.1 Issues in Change Notification

Broadly, there are two major issues. One is understanding the types of changes that need to be monitored. Another is the notification mechanisms and timing (when to notify).

4.1.1. Types of Changes Monitored

There are three types of changes that need to be monitored. First, as discussed in Section 2, a design object in general has multiple representations that are hierarchically related. For example, a logic representation can have many circuit representations, a circuit representation can have many layout representations, etc. Usually, lower level representations of design objects are derived from their higher level representations, and thus changes made to one representation may

invalidate more detailed, lower level representations [WIED82]. For example, changing the logic representation of a design object may invalidate its circuit representations. Therefore, changes in a design object across multiple representations need to be monitored.

Second, as also shown in Section 2, a design object is constructed by instantiating (referencing) more primitive objects. Suppose object A is instantiated in the implementation of object B. If A is changed, then its changes are automatically propagated to B. Object B is thus changed, and objects that instantiated B in their implimentation are changed, and so on. This can lead to a potentially messy situation where an exponential propagation of changes occurs [WIED82]. Therefore, changes to versions of a design object definitely need to be monitored.

Third, changes to a version may be of interest to versions derived from it along the same version-derivation hierarchy. We note from our algorithms of the previous sections that changes made to one version are localized in that they do not propagate or affect the validity of other versions. Nevertheless, changes to versions within version-derivation hierarchies should be monitored.

It is worth noting that one type of change can give rise to other types of changes, thus complicating the change monitor/notification process. For example, when a version v-i of design d-i is changed, it may affect the validity of version v-j of another design d-j which instantiated it. This can affect the validity of versions of different representations of v-i and v-j.

4.1.2. Notification Mechanisms and Timing

Two mechanisms can be used to notify changes to affected designs or interested designers. One is the use of auxiliary directories, in which a summary of the changes may be posted. The other is to use timestamps to indicate when a version was last modified and approved. For both mechanisms, an *active* or *passive* notification strategy may be used. In the active approach, the system would generate interprocess messages to inform interested designers once changes to a version have been committed. In the passive approach, designers are expected to check the auxiliary directories periodically for changes, or if timestamps are used, the system would notify the

designer of changes to a version when the version is accessed.

Both mechanisms and notification timing strategies are well suited for design systems that are organized around a public/private distributed architecture [HASK82, KIM84]. In such architectures, multiple workstations are running single-user database systems which communicate with a central host that manages the public database of stable design data and design control data. Applications extract (take copies of) versions of design objects from either the public or private database and insert them into the application work space. It is at this time that the system would notify a designer of version modifications if timestamps with passive notification are used. Once transferred, the versions can be updated or new versions created. Later they can be checked-in (returned) to the central database for public examination. It is at this point when the system would notify affected versions of design modifications if an active notification approach is used.

In the following section, we will explain in more detail how the timestamp and directory notification mechanisms work.

4.2 Change Notification Techniques

Although either of the notification mechanisms can be used for any of the three types of changes, there is a preference for a particular type of change. We believe that the following is an appropriate matching:

1. Timestamps:
   a. changes in an object that was instantiated by another object, both objects residing in the same private database.
   b. changes in a representation of a design from which other representations were derived.

2. Auxiliary Directories:
   a. changes in versions along a version-derivation hierarchy.

   b. changes in an object that was instantiated by a design, which was checked out from the public database into a private database.

   c. changes in an object residing in the public database from which a new representation or a new version was derived in a private database.

A brief description of these mechanisms is given below.

4.2.1 Timestamp Mechanisms

From a research point of view, the more interesting notification mechanism is the use of timestamps with versions. Each version will have two distinct timestamps. One timestamp, called the *change-notification time stamp (CN),* indicates the last time the version 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. We say that version V is *implementation consistent* if V.CA >= V.CN. An *implementation inconsistent* version is one whose implementation has been updated but not yet approved. (We presume that versions stored in the central database would always be implementation consistent, although it is quite possible for versions to undergo a series of changes before being approved in a private database).

Let I be the set of versions that are instantiated in the implementation of 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 we say that 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 the implementation of 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.

Timestamps can also be used to test for consistency across different representations of a version. For example, if the logic representation of a version changes, then the circuit representations of the version may no longer be valid. This, in turn, may invalidate the layout representations of the design, and so on. Let H be a version and L be version which defines a lower-level

representation of H. We say that L is *representation consistent* with H if H.CN <= L.CA. L is *representation inconsistent* if H has been modifed and the effects of these modifications have not been determined on L. Acknowledgements of these changes are handled in the same way as that for referential consistency.

The timestamping mechanism described above can result in a version that is seen as consistent, while a related version (i.e., one that it instantiates or is a lower-level representation) is inconsistent. For example, suppose version V is created by instantiating version A which is implementation and reference consistent. Now suppose A is made reference inconsistent by a modification of a version that it instantiates. The implementation and reference consistency of V is affected only if the indirect modifications to A cause A to be directly modified. Until this determination is made, V remains consistent.

Clearly there is a synchronization problem. We say that a version is *totally consistent* if it and its instantiated versions are both implementation consistent and reference consistent. Total consistency is required, for example, when a design object is to be released. In the release process, a totally consistent version of the object for each representation is selected as the "final" definition of that object in that representation. Furthermore, all selected versions be representation consistent. Testing for total consistency using the above mechanism is straightforward and will not be described in detail here.

We noted in Section 4.1.1 that changes can propagate up a design hierarchy, and there is the possibility of an exponential explosion of notifications. In [WIED82], an algorithm for change notification was proposed where a trivial change in a low-level design causes notifications to propagate to the highest-level designs. This is clearly undesirable. Our timestamping technique limits the possibility of exponential notifications in a practical way. If a small change is made to version A, version V (which instantiates A) will be notified. But if the changes to A cause no modification of V, then all versions that instantiate V in their implementation need not be notified. In this way, the propagation of change notifications is localized.

We envision that consistency tests will be performed at the specific request of users (e.g., procedures to test for a particular consistency can be called) and checks for reference and implementation consistency can be done when versions or their instantiations are read.

4.2.2 Auxiliary Directories

The use of directories to keep track of changes to a version along a version-derivation hierarchy and the check-out/check-in status of versions of design objects that reside in the public database is straightforward and obvious idea.

The only important observation that perhaps needs to be mentioned here is that the directories must be *persistent,* that is, they must persist (survive) across crashes of the public system, and user sessions. As suggested in [KIM84], this can be very simply achieved by maintaining the directories as part of the public database, taking advantage of the concurrency control and recovery features of the public database system.

5. Concluding Remarks

We have presented results on two aspects of supporting version control in VLSI CAD. First, we described a storage technique for versions that allows both the sharing of records that are common to related versions and also fast access to any version. This is a generalization and an improvement over earlier proposals which supported storage sharing and fast access only to a single (usually the most recent) version and much slower access to remaining versions. Also, our structure allows the history of version derivations to be captured by a tree of parent-child relationships, rather than earlier methods of a linear chain of versions. Moreover, both working versions and released versions can coexist in the same storage structure and working versions can be updated at any time. The ability to access efficiently many versions, to derive several versions from a single version, and to update working versions are definite requirements of VLSI CAD databases.

Second, we identified issues in monitoring changes in versions and notifying affected designs. A rather simple, but effective, techniqe using timestamps to flag inconsistencies was presented. Its

advantage over earlier proposals is that the affects of changes can be localized. That is, minor changes which can be confined to lower-level designs should not cause higher-level designs to be notified. The ability to selectively propagate changes and notifications is also a definite requirement of VLSI CAD databases.

Further work needs to be done on change propagation (i.e., the automatic propagation of changes from one version to another) and a more detailed analysis of the performance of the version storage structure. It is hoped that the analysis might lead to decision rules for automating the creation of version segments.

Acknowledgements

We thank Drs. Terry Welch and Jay Banerjee of MCC for their helpful comments on an earlier draft of this paper.

References

[BATO84] Batory, D. and A. Buchmann. "Molecular Objects, Abstract Data Types, and Data Models: A Framework," in Proc. Intl. Conf. on Very Large Databases, August 1984, pp. 172-184.

[BATO85] Batory, D. and W. Kim. "Modeling Concepts for VLSI CAD Objects," ACM SIGMOD Intl Conf. on Management of Data, May 1985.

[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.

[EDIF84] Electronic Design Interchange Format, preliminary specification, version 0.8

[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.

[KATZ82] Katz, R. and T. Lehman. "Storage Structures for Versions and Alternatives," Technical Report no. 479, Computer Sciences Dept., U. of Wisconsin, July 1982.

[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.

[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.

[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.

[SEVE76] Severance, D. and G. Lohman. "Differential Files: Their Application to the Maintenance of Large Databases," ACM Trans. on Database Systems, vol. 1, no. 3 (Sept. 1976), pp. 256-267.

[ULLM84] Ullman, J. Computational Aspects of VLSI, Computer Science Press, 1984.

[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.

Appendix: Proof of Correctness of the Storage Structure Algorithms

Algorithms_Insert, Delete, and Update manipulate the bitmap entries to ensure that Algorithm_Retrieve will retrieve only those records that properly belong to a version. And Algorithm_Retrieve states that a record belongs to a version v-i if the bits in the bitmap corresponding to the versions along the chain of ancestors from the owner of the record to version v-i are all 0. Therefore, to prove correctness of Algorithms_Insert, Delete, and Update, we need only show that the algorithms preserve the bitmap in the format that Algorithm_Retrieve requires.

Algorithm_Insert is trivially correct. It inserts a new record into version v-i, establishes v-i as the owner, and initializes the bitmap by setting the bits for v-i and all non-existing versions to 0. The consecutive 0 bits from v-i and its future descendant versions will enable Algorithm_Retrieve to pick up the record.

Algorithm_Delete either results in a physical deletion of a record or a logical deletion and replication of a record. The algorithm is trivially correct, with respect to the physical deletion, since a physically deleted record cannot be incorrectly retrieved by Algorithm_Retrieve.

We only need to examine the logical deletion. Algorithm_Delete updates the bitmap in the original record and inserts a copy of the record into each of the immediate child versions that inherited the original record from v-i. The algorithm updates the bitmap in the original record owned by an ancestor of v-i, by setting the bit for v-i to 1. This 1 bit breaks the string of 0 bits between the owner of the record and v-i and any of its descendants, and thus prevents Algorithm_Retrieve from seeing it. On the other hand, the algorithm inserts a copy of the original record into each of the immediate child versions of v-i, with the child version as the owner. The records thus inserted are visible to the new owner and all its descendant versions, trivially conforming to the bitmap format that Algorithm_Retrieve requires.

Algorithm_Update is virtually identical to Algorithm_Delete, and arguments similar to those given above can be used to show its correctness.