CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

Versions and Alternatives of Structural Engineering Design Objects

Jamal A. Abdalla and Graham H. Powell

TECHNICAL REPORT
Number 51

July 1991

Stanford University

Copyright © 1991 by

Center for Integrated Facility Engineering

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020

Versions and Alternatives of Structural Engineering Design Objects[1]

Jamal A. Abdalla[2] and Graham H. Powell[3]

[1]This report is a modified version of a paper to appear in the Journal of Engineering with Computers 1991.

[2]Postdoctoral Research Affiliate, Center for Integrated Facility Engineering, Stanford University. Formerly Graduate Research Assistant, University of California at Berkeley.

[3]Professor of Civil Engineering, University of California at Berkeley.

ABSTRACT

As design objects are created during the design process, it is not sufficient to represent only a single current state of each object. Instead, it will usually be necessary to represent a number of past and current states, existing parallel to each other at various design stages. This requirement has led to the development of various schemes to help manage and control multiple occurrences of a design object. In this paper "versions" of structural engineering design objects are defined, and a scheme for organizing these versions is proposed. A design for a Version Manager in an object-based design system is then presented. The manager has limited capabilities, but can be used as a step toward defining a more complete version management and control system.

Content

Abstract 2

Content 3

1. General 4

2. Need for Version Management 4

3. Versions and Alternatives 6

3.1 Basic and Complex Design Objects 6

3.2 Versions and Alternatives of Basic Design Objects 6

3.3 Versions and Alternatives of Complex Design Objects 8

4. Work from Other Fields 11

4.1 Business Data Base Systems 13

4.2 VLSI Design Version Systems 13

5. A Simple Structural Engineering Version Model 14

5.1 General 14

5.2 Version Characteristics 14

5.3 Characteristics of the Version Tree 15

6. The Version Manager 16

6.1 General 16

6.2 Features of the Version Manager 16

6.3 Version Management Operations 17

6.3.1 Action Operations 17

6.3.2 Query Operations 23

7. Creation and Management of Master Objects and Versions 24

8. Some Details for the Implementation of the Version Manager 25

9. Conclusion 26

10. Acknowledgement 28

11. References 28

1. General

A typical building structure will consist of many different components, such as frames, floors, beams, columns, slabs, etc. In an object-based computer aided design system, each component can be identified as a *design object*, which is a member of some object class and has attributes that define its dimensions, properties, state, etc. Design objects are likely to be complex, and to evolve continuously during the design process. Hence, as design objects are created, it is not sufficient to record only their current attributes. Instead, it is necessary to record the attributes defining a number of past and current occurrences of each object, existing parallel to each other at various design stages. This requirement exists for all types of design, not just structural engineering design, and has led to the development of schemes to help manage multiple occurrences of a design object. The terms which have been introduced to mean different copies or representations of the same design object include "versions", "alternatives" and "variants". In the early sections of this paper, the term "version" is used. Later, the term "alternative" is added, and the two terms are given specific definitions.

This paper identifies a number of issues associated with the management of design objects in structural engineering, and proposes a limited scheme for organizing design object versions. Before details are presented, it is useful to emphasize the nature of the design process, and to identify the features of design objects that necessitate the creation and management of versions.

2. Need for Version Management

The engineering design process is typically a *multi-path* one, in which any of several paths may yield a valid design, and an *incremental* one, in which the development proceeds incrementally along each path. This warrants some elaboration, as follows.

(1) Engineering design is a multi-path process in which many valid designs can be obtained by following different development paths. For example, if there are several design requirements that need to be satisfied, they can be imposed in a variety of different orders, and each order will constitute a path. The designer may need to explore design objects along several paths, without destroying all previous trials. A mechanism to keep track of design object versions created along different design paths is essential.

(2) Engineering design is an incremental process. When an object is under design, it is likely that only certain criteria will be used at any one time for defining the object attributes, and that the attributes will be defined progressively. It may be necessary to preserve versions along an incremental path, to record the development history and to permit backtracking.

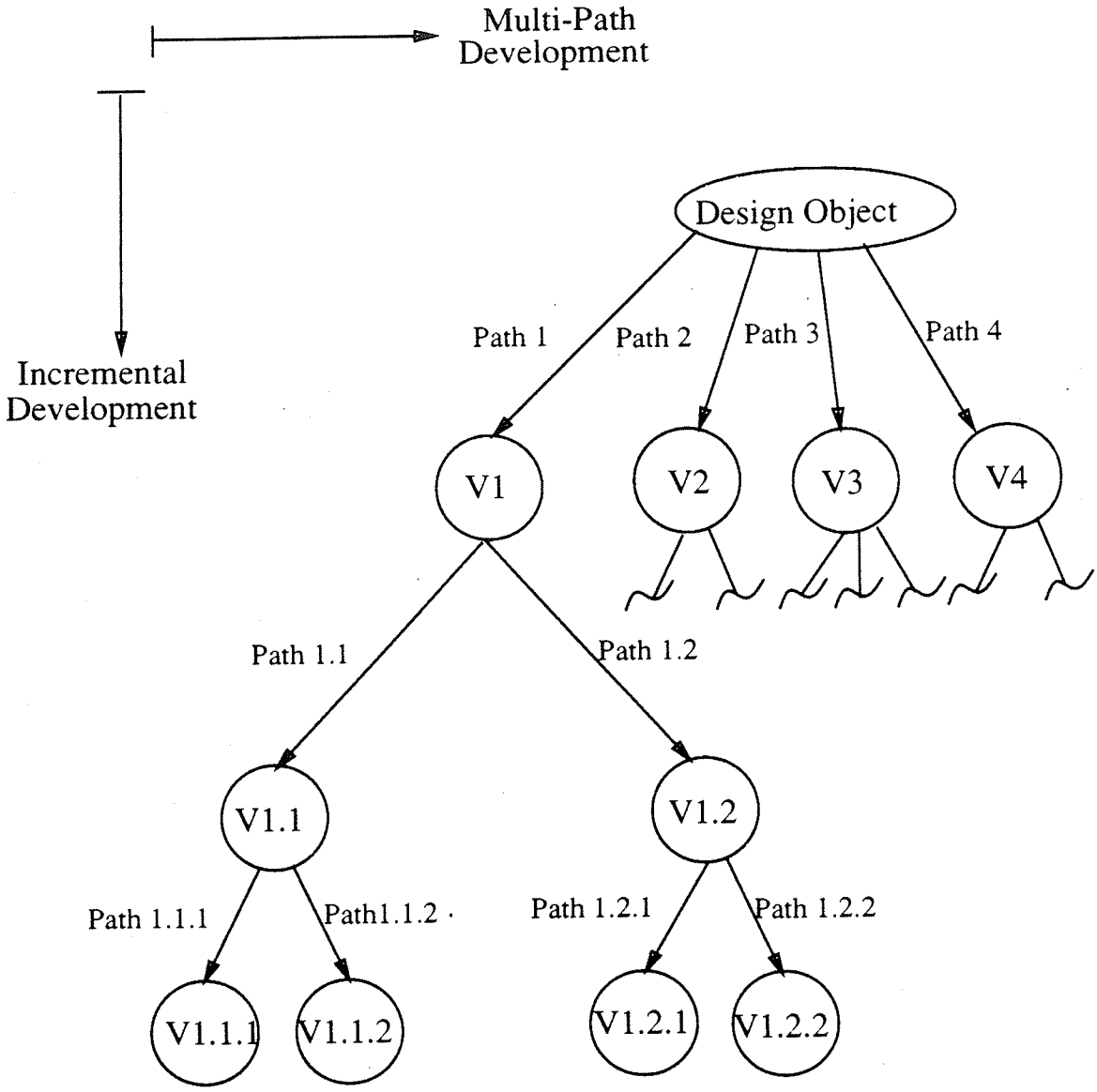


Figure 1 Version Tree

The multi-path and incremental nature of the design process can be represented using a tree structure, where the branches of the tree represent different design paths and the levels represent design increments, as indicated in Figure 1. The tree structure has the advantage that it separates the path aspects from the incremental aspects of the design process.

3. Versions and Alternatives

3.1 Basic and Complex Design Objects

In structural engineering design, it is useful to distinguish "basic" or "simple" design objects from "complex" or "assembled" objects [1,2,3]. Basic design objects are treated as a unit, and for design purposes are not decomposed into simpler ingredient objects. Examples include beams and columns. Complex design objects are assemblies of basic design objects and/or other complex objects, and must be decomposed into their ingredient objects. Examples include frames and floors. Versions and alternatives for these two types of objects are addressed separately in the following sections.

3.2 Versions and Alternatives of Basic Design Objects

Consider the design of a reinforced concrete (RC) column. Typically, different types of RC column will be considered, for example rectangular columns with rectangular ties (RectTied), square column with square ties (SquaTied), square columns with spiral ties (SquaSpiral), and circular columns with spiral ties (CircSpiral). Each type performs the same function, but is a distinctly different type of design object. In an object-based system, each type will correspond to a different object class. Objects of different types will be defined as *alternatives*. For example, a RectTied column might be replaced by a SquaSpiral column alternative, which serves the same purpose of carrying axial force, bending moment and shear.

The first step in designing a column will typically be to select a number of alternatives. For example, in Figure 2 RectTied, SquaSpiral, SquaTied and CircSpiral alternatives are chosen. For each alternative, a *master object* is created, which is an instance of the corresponding object class. Each master object will have several attributes, to which values must be assigned during the design process. In general, some of these attributes will be defined for the master object, and the rest will initially be undefined. For example, in Figure 2 one dimension of the RectTied column alternative is required to be 20 inches, but the other attributes are initially undefined. Several *versions* can then be derived from each master object, as indicated in Figure 2. Each version is a variant of the master object, obtained by assigning additional attribute values. One approach to the process of assigning these additional values is to require that they follow a strict inheritance hierarchy, as shown in Figure 2. The process is as follows.

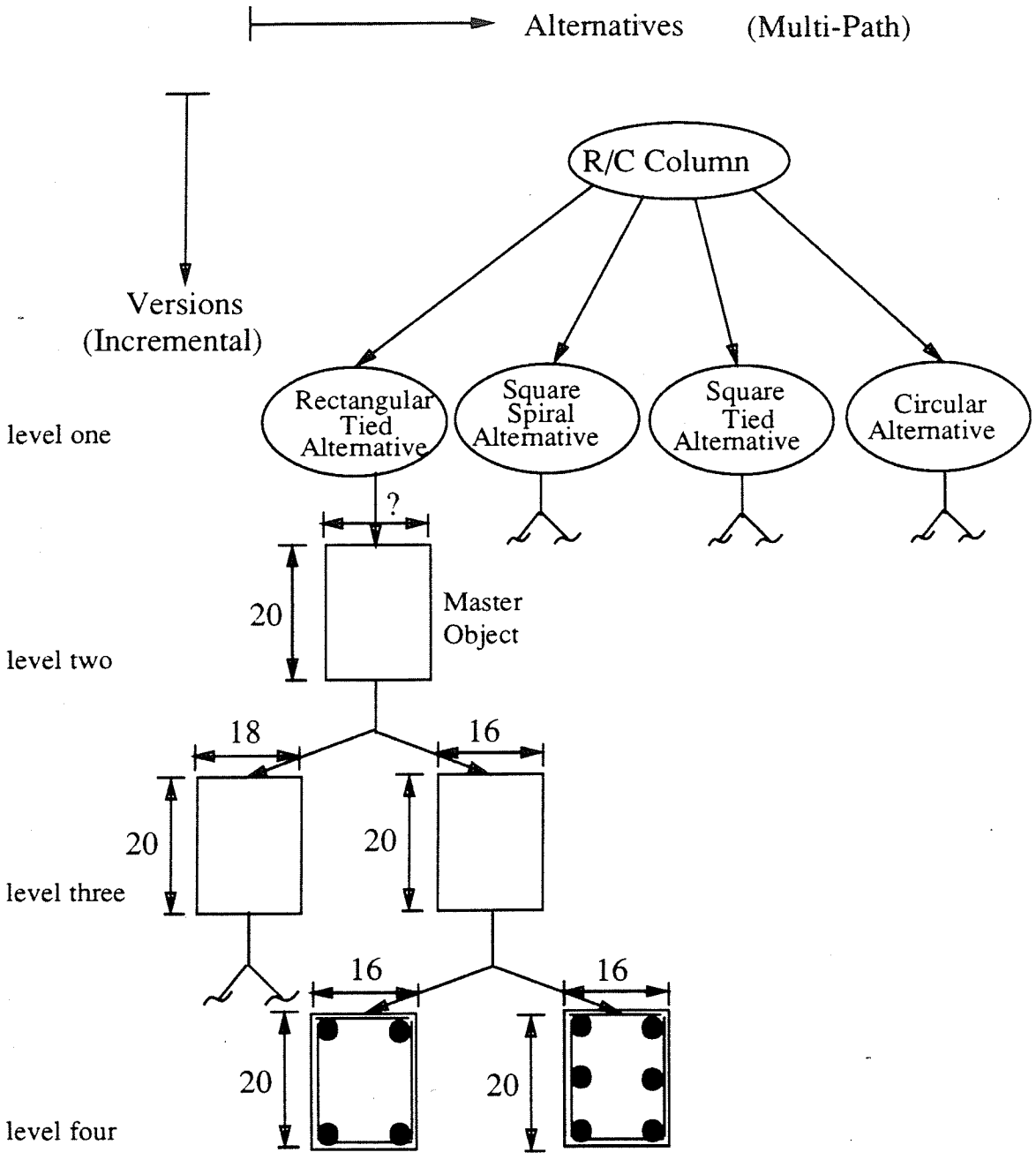


Figure 2 Alternative-Version Hierarchy: Alternatives and Versions Separated

In Figure 2, level one contains the alternatives for the column object. Level two then contains the master objects, each of which is the root of a *version tree*. Levels three and four then contain versions, each of which is derived from the version at the next higher level, by specifying additional attribute values. For example, for the RectTied alternative, the column width has been specified at level three, and the longitudinal reinforcement at level four. In this approach, attribute values can only be added at each level, and values specified for designs at higher levels can not be changed. Hence, all design versions inherit the attribute values specified for the master object.

This scheme is a restrictive one, but it is simple to manage and it indicates some of the issues which must be decided in creating a version management system. Note that Figure 2 has the same tree structure as Figure 1. Horizontally, the branching corresponds to choosing alternative design paths. Vertically, the levels correspond to incremental addition of detail to the design.

A simple separation of alternatives from versions is not the only possible scheme. For example, Figure 3 shows a scheme in which alternatives and versions are mixed. In this scheme alternatives are derived from versions, which is not allowed in the previous scheme. When all alternatives are exhausted, the rest of the tree contains only versions, and is similar to the lower levels in Figure 2.

This second scheme is more complex than that in Figure 2, since the tree no longer consists of distinct levels separating versions and alternatives. A version management system which allows this type of tree would clearly be more complex than one which allows alternatives only at the highest level of the tree.

3.3 Versions and Alternatives of Complex Design Objects

Since complex design objects are composed of other objects, it is necessary to consider decomposition as well as alternatives and versions. Consider a simple frame as an example. For a frame design object, several alternatives might be considered, as shown in Figure 4. These alternatives include a reinforced concrete frame (RCFrame), an unbraced steel frame, and a braced steel frame. Figure 4 shows these alternatives at the top level, and also shows the development of the RCFrame alternative. Assume that the frame is known to have two stories, but that the number of bays is undefined. The master object thus has two stories and an unspecified number of bays, as shown in Figure 4. Several versions can be derived from this master object, each with a different number of bays. The figure shows two versions, with two and four bays, respectively. Since an RCFrame object is composed of several lower-level design objects, it is next necessary to decompose it. In Figure 4, the two-bay version is shown decomposed into three (multistory) columns and two (multibay) girders, which will be assumed to be basic objects. It is now necessary to deal with alternatives and versions of these basic objects. The figure shows two alternatives for one of the columns, namely Rectangular/Square and Circular, as considered in the example of Figure 3. Below this point, the version tree might be identical to that in Figure 3.

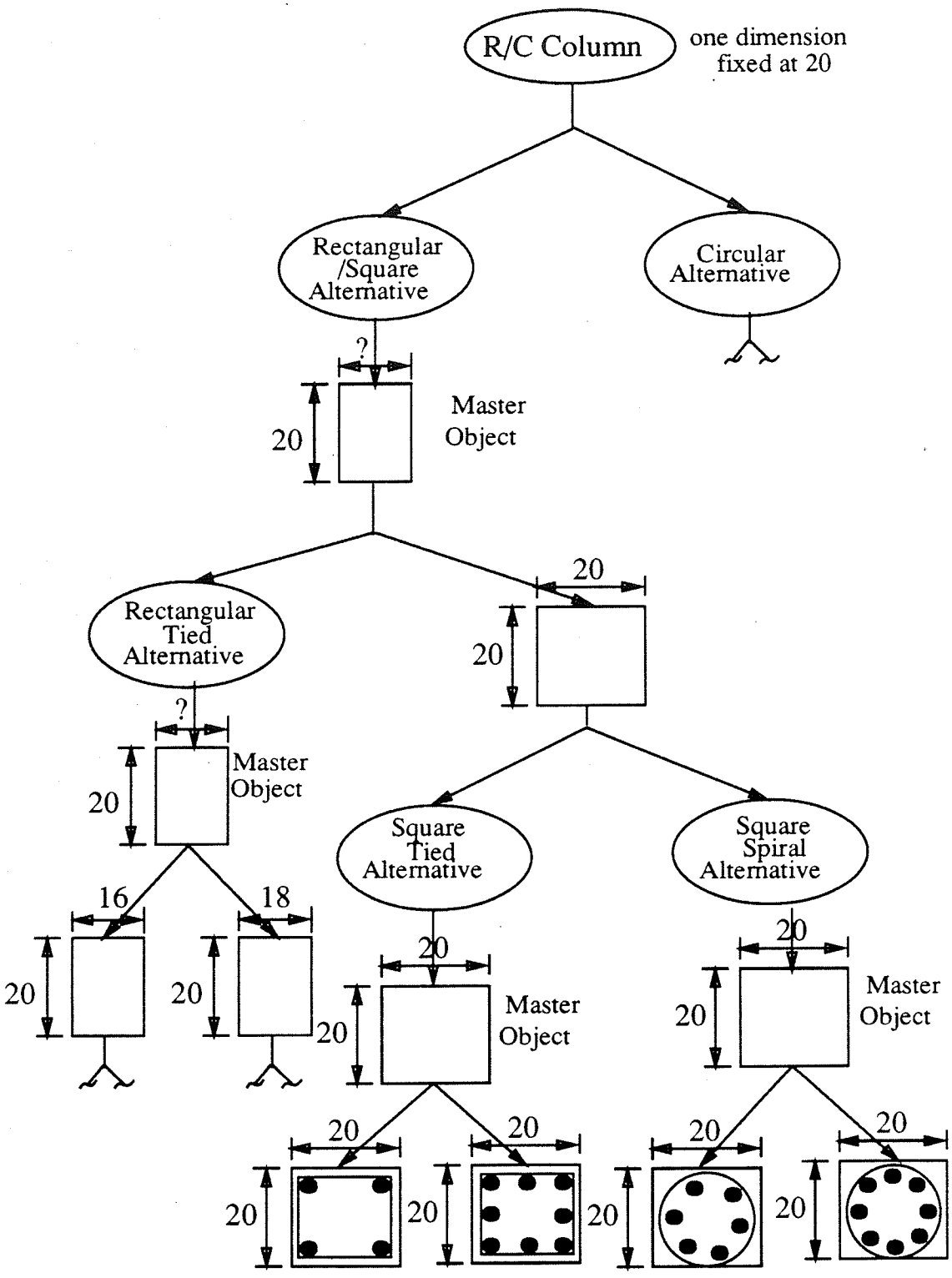
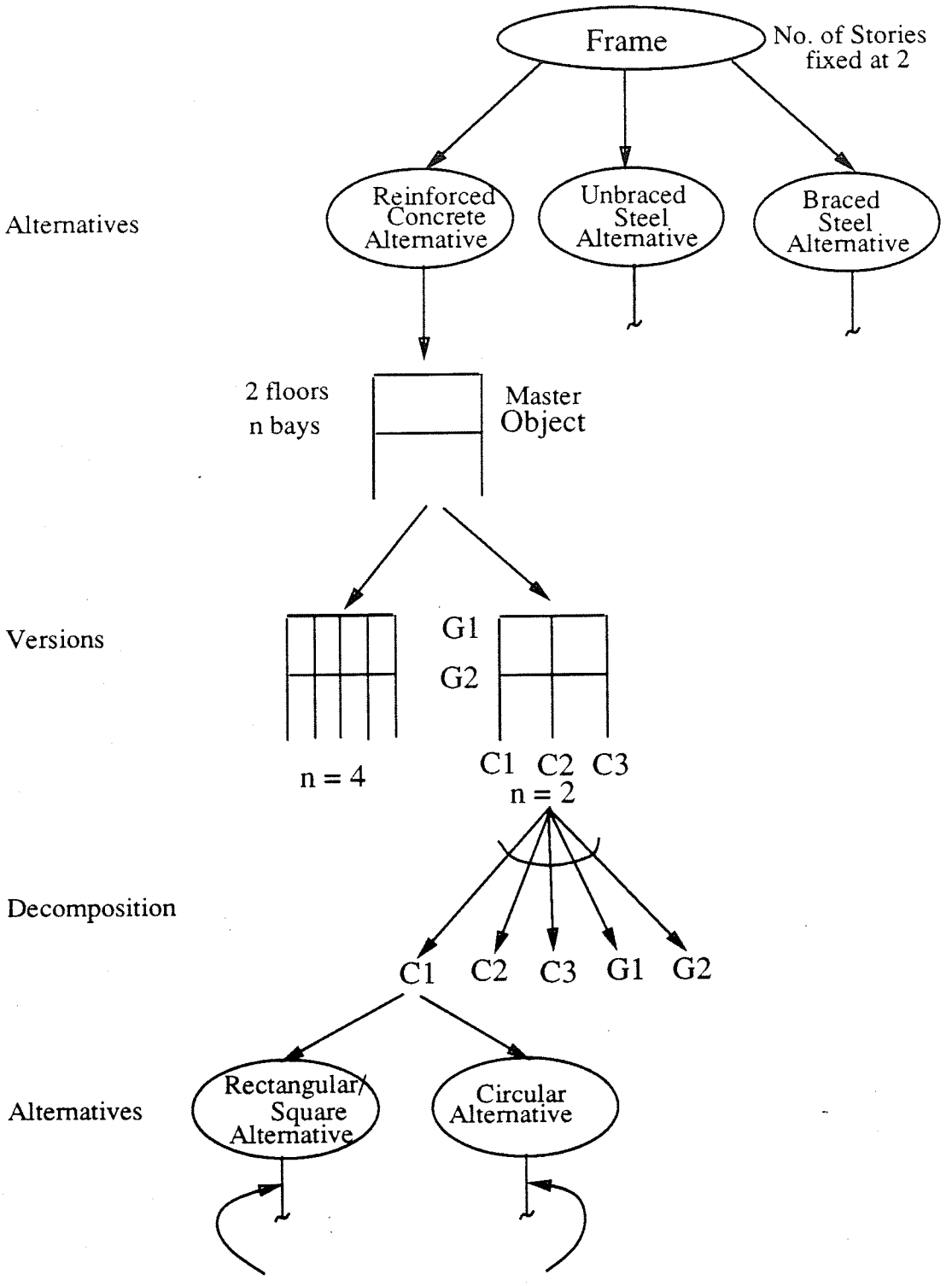


Figure 3 Alternative-Version Hierarchy: Alternatives and Versions not Separated



Same as Figure 8.3

Figure 4 Frame Assembly Object: Versions, Alternatives and Decomposition

The version scheme for complex design objects must thus allow for another level of complexity, because of the decomposition of the object into its ingredient objects. The scheme must include nested alternatives, versions, and decompositions. A comprehensive version management system which accounts for alternatives, versions and decompositions in a general way is likely to be very complex.

4. Work from Other Fields

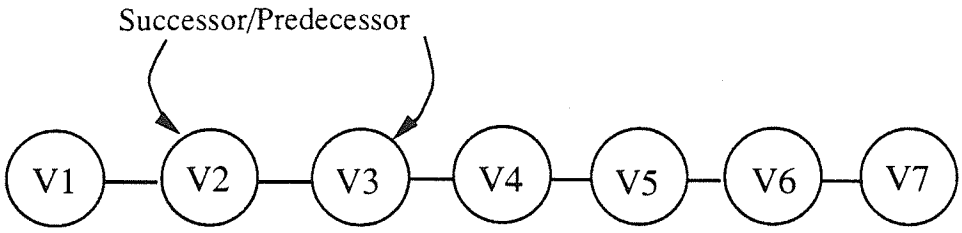
The importance of version management has been realized by researchers in different fields of application, including data base systems [4, 5, 6, 7, 8], VLSI CAD systems [4, 9, 10, 11, 12] and office automation systems [13].

As previously indicated, versions are the result of changes that accumulate over time. Depending on the nature of the relationships between versions, there are several ways to organize them. The following three schemes have been used.

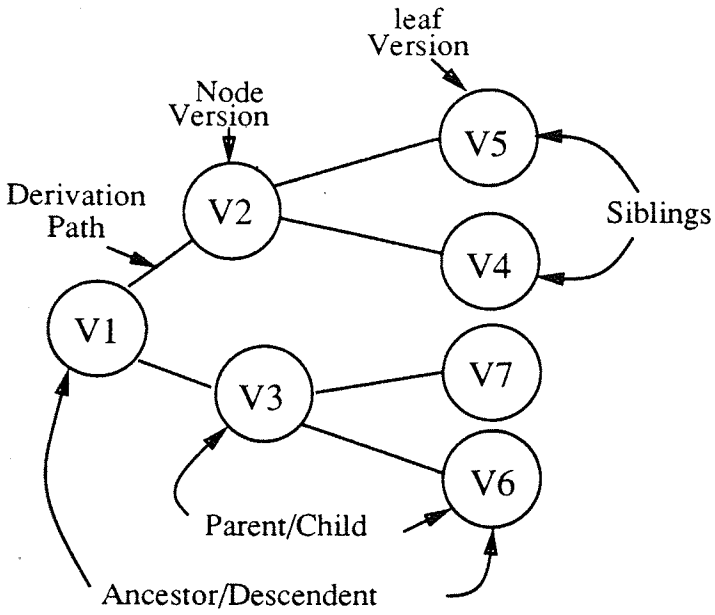
1. Linear Scheme. In this scheme, versions are organized linearly in an ordered collection form, such that each version has a unique predecessor and successor (except, of course for the oldest and youngest versions), as shown in Figure 5a. Such a scheme has been described in [14] and [15]. It has the advantage that the scheme is simple, but the disadvantage that it mixes the incremental and multi-path aspects of the design process.

2. Tree Scheme. In this scheme, many versions can be derived from one version, as shown in Figure 5b. This scheme has been used in [16]. It is similar in principle to the schemes described in the preceding section, with the advantage that it separates the incremental and multi-path aspects of the design process.

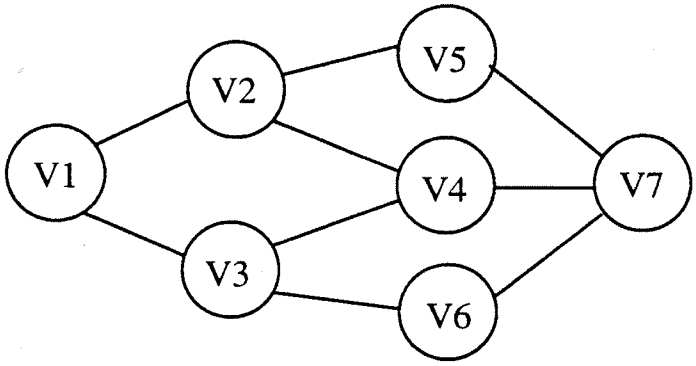
3. Network Scheme. In this scheme, two or more versions can be merged to form a new version, as indicated in Figure 5c. This scheme has been described in [17]. it allows a version to have more than one parent, and supports consolidation of many properties into one. This scheme not only allows separation of the incremental and multi-path aspects of the design process, it allows paths to be combined. It is, however, inherently more complex than the tree scheme.



(a) Linear (Refinement)



(b) Hierarchical (Derivation)



(c) Graph (Consolidation)

Figure 5 Version Models

4.1 Business Data Base Systems

In a typical business data base system, the data base reflects the current status of the data, and in effect contains the most recent version of all objects. The term "version" for such a data base is commonly used to denote a different state of the whole data base. This is reasonable because business transactions tend to be simple and of short duration. In engineering design applications, however, transactions are usually long and complex. Also, different versions of a single object may need to be retained for long periods, and multiple users may need to access objects concurrently. This makes the version management problem much more complex, and the difficulties in using business-type data bases to support design has stimulated work on data bases specifically for engineering purposes. Most of the work has been in the area of VLSI design.

4.2 VLSI Design Version Systems

A distributed version server for VLSI design data has been described by Katz [11,16]. In this system, several relationships among the object versions are tracked and managed by the version server. The server uses a tree structure for organizing versions, and classifies the data into three "planes". The first of these is the version plane, which organizes the copies of an object into a version history that relates the object's derivative versions and alternative versions. The second plane is the configuration plane, which combines versions of different objects into composites (similar to complex design objects). The third is the equivalence plane, which relates different representations of the same object. This plane is important for VLSI design because design objects are dealt with at a number of well defined levels of abstraction, each with a distinctly different data representation. The version server allows multiple designers to work on one project by supporting multi-read access and single write access to the data for any object.

A version control "environment" for VLSI design has also been suggested in [9, 18]. This environment distinguishes among three version types, namely released versions, working versions, and transient versions. The version classification determines which of three data bases each version resides on, and accordingly the types of operation that can be applied to each version. The environment defines three types of data base, namely public, project and private. The version server uses checkin, checkout, promotion, and derivation operations to achieve multi-read and write access to object data.

Several other systems has been suggested, for example the Design Object Storage System (DOSS) by [19]; the Interface for a Semantic Information System (ISIS) by [14]; and AVANCE by [6]. Since no system has been developed for structural engineering, it is not clear whether they address the version management issues which are important in computer aided structural design.

5. A Simple Structural Engineering Version Model

5.1 General

In this section, a simple structural engineering version model is suggested. This model allows for precise definition of version representation and version relationships, and defines rules for version management. The model does not have the range of features needed for practical version representation and management. However, it indicates several of the features which are believed to be needed in a practical system, and it could be used as a starting point for further development.

There are two major aspects of the Version Manager, namely (1) organization of the versions in a Version Tree, and (2) definition of operations on this tree. The characteristics of versions, and the organization of the Version Tree, are described first. Features of the Version Manager and the operations which it supports are then described. The emphasis is on concepts and features, rather than implementation.

5.2 Version Characteristics

When a design object is first created (instanced as a member of a design object class), it becomes a Master Object, at the root of a Version Tree (e.g., as in Figure 2). That is, the choice of design object class defines the *alternative*, and the Version Manager is concerned only with *versions* of that alternative.

Each attribute of the Master Object will be of one of the following status types:

(a) *Frozen* –the value of the attribute is specified for the master object, and can not be changed by the design application program that uses the object.

(b) *Redefinable* –the value of the attribute is specified, but it can be changed by the design application program that uses the object.

(c) *Undefined* –no value has been assigned to the attribute, and a value may be assigned by the design application program that uses the object.

If the attributes of the Master Object are changed, and if versions are derived from the master object, certain rules are imposed for inheritance and updating of the attribute values. These rules are as follows.

(1) As stated, values may be assigned to redefinable or undefined attributes of the Master Object, but not to frozen attributes. If a value is assigned to an undefined attribute, it

switches to redefinable status. A redefinable attribute may also be switched to undefined status.

(2) A version inherits all of the frozen and redefinable attribute values of its parent. That is, attributes which are either frozen or redefinable for the parent become frozen for the version. Undefined attributes retain their undefined status. Hence, when a version is first created, its attributes are either frozen or undefined. If a value is assigned to any undefined attribute of the version, that attribute switches to redefinable. Redefinable attributes can be set to undefined provided that the version has no child versions.

(3) If a version is a parent for other versions, the values of its redefinable attributes can not be changed. This is because these values have been inherited by the child versions, and are frozen in those versions. To change them in the parent would be inconsistent.

(4) If a version is a parent for other versions, it is possible to assign a value to an undefined attribute, provided the attribute is still undefined for all of the child versions. The attribute value is then inherited by the child versions, and becomes frozen for those versions. By the preceding rule, the value can not be changed further. It follows that redefinable attributes can be assigned new values in a version only if the version possesses no children.

5.3 Characteristics of the Version Tree

The Version Tree reflects the history of version creation, and the relationships of the versions to each other and to the Master Object. The following are the characteristics of the Version Tree:

(1) A Master Object is the oldest version, at the root of the tree. Most of the attributes of this object will typically be undefined.

(2) Versions with a common parent are *siblings*. Sibling versions have the same frozen attributes, inherited from their common parent. However, each of them defines some or all of its own undefined attributes, independent of its siblings.

(3) A node version (as distinct from a leaf) will have at least one attribute which is undefined. Node versions are thus *versionable* (i.e., new versions can be derived from them).

(4) A leaf version is either *versionable* (some of its attributes are undefined) or *non-versionable* . A *non-versionable* version is either *complete* (where all of its attributes are either fixed or redefinable) or else *dry* (where some of its attributes are undefined, but it is known that some design constraints are violated, and hence there is no point in deriving further versions).

(5) The operations which can be performed on versions depend on their relationships to each other, which include parent-of, child-of, descendent-of, ancestor-of, sibling-of, etc. Each relationship implies certain permissible operations, as described later.

6. The Version Manager

6.1 General

Some researchers have considered Version Managers that know certain details of the versions they manage, and therefore have some capabilities for carrying out design operations. The Version Manager envisaged here does not know any such details, and does not support design operations. Hence, operations such as checking to determine whether a version satisfies the design constraints, comparing versions properties, and so on must be performed by application programs.

In the following section, features of the Version Manager are described, and the operations which it performs are defined.

6.2 Features of the Version Manager

Each time a new Master Object is created, an associated Version Manager object is also created. Subsequent versions created from the Master Object are automatically logged with the Version Manager, and they are under its control. The Version Manager maintains information on the the Version Tree, and also maintains information such as the time at which each version was created. Each version managed by the Version Manager is a separate object, of the same class as the Master Object, and derived from the Master Object or one of its descendents.

The Version Manager and the versions it manages are created during the design process, and they are owned by the Master Object. The versions are destroyed once a final selection for the design object is made. Each object in the Version Tree is identified by a unique Version Identifier, which is assigned by the Design Application Program that creates the versions.

During the design process, the operations which must be performed can be grouped into three broad categories, as follows.

(1) Design Operations. These are the lowest level operations, performed on a specific design object version to determine values for its attributes or to check its design. These operations are performed under the control of the Design Application Program, and must be supported by the class to which the design object belongs. The Version Manager knows nothing about these operations, and they are beyond the scope of this paper.

(2) Version Management Operations. These are intermediate level operations, performed essentially on the Version Tree. These are the operations supported by the Version Manager, and are the main topic of the next section.

(3) Object Management Operations. These are high level operations, performed on complete Version Trees. They particularly include creating new Master Objects and Version Trees, storing Master Objects and all of their versions in a data base, and activating objects to permit design and version management operations to be performed. The details of these operations are beyond the scope of this paper, although some aspects of this issue are considered in a later section.

6.3 Version Management Operations

For the version model defined in the preceding sections, a set of operations can be defined that is supported by the Version Manager. These operations allow a Design Application Program to define and organize versions of design objects. There are two types of operations supported by the Version Manager:

- Action operations, which change the configuration of the Version Tree.
- Query operations, which provide information about the tree and individual versions.

The following subsections describe these operations.

6.3.1 Action Operations

Figure 6 shows a version tree of several versions derived from the master object. This version tree will be used as an example to show the effect of action operations in the version tree.

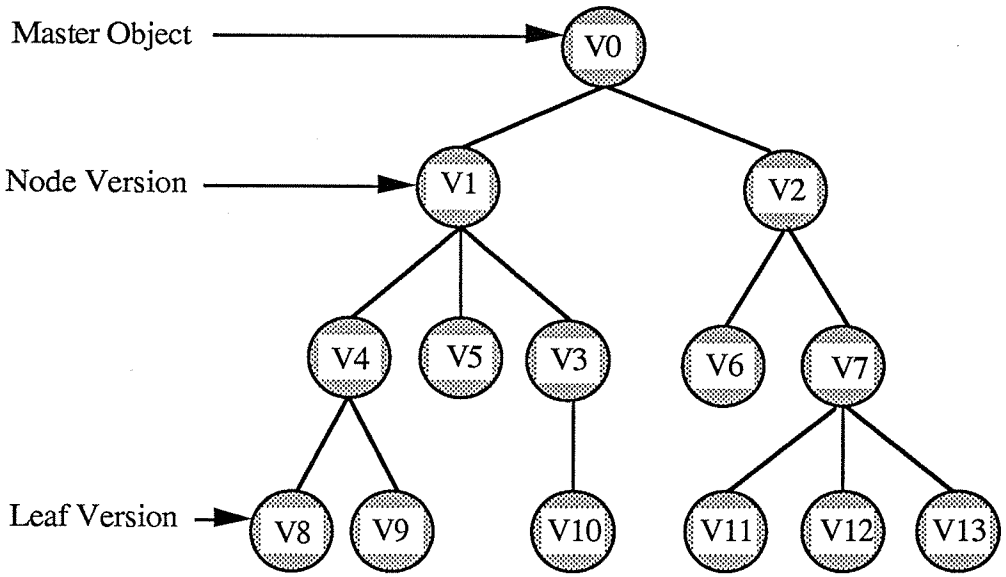


Figure 6 Version Derivation Tree

• **Add**(version_Id,parent_Id):

This method adds the given version as a child of the given parent. It adds only one version at a time. Versions can be added to leaves, nodes or the Master Object.

The process of adding a new version to the Version Tree involves deriving a new version from the given parent, then adding it to the version tree. The Version Tree is searched to locate the parent. A copy of the parent is made, and assigned the given version_Id. This new version is then added as a child of the parent version. Design operations can now be performed on the new version, to assign values to its undefined attributes. Figure 7 shows some **Add** operations that have been applied to the Version Derivation Tree of Figure 6.

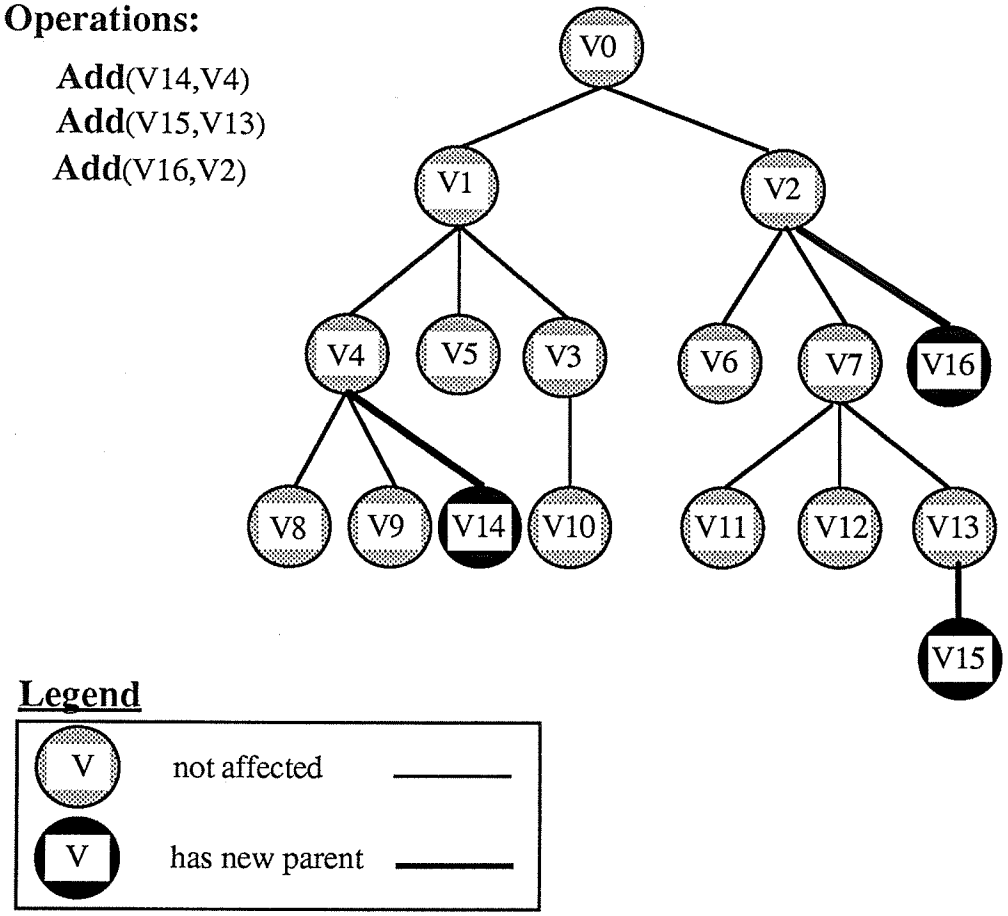


Figure 7 Add Operation

• **Delete**(version_Id):

The Delete operation applies to both leaves and nodes. Deleting a leaf simply detaches it from the version tree. Deleting a node results in "grafting" the children of the deleted node to the parent of the deleted version.

For leaf deletion the Version Tree is searched to locate the target version, and this version is simply deleted from the tree. For node deletion the Version Tree is searched to locate the target version, and the immediate children of the target version are grafted to its parent. The target version is now a leaf, which is deleted using the algorithm for leaf deletion. Figure 8 shows some **Delete** operations that have been applied to the Version Derivation Tree of Figure 6.

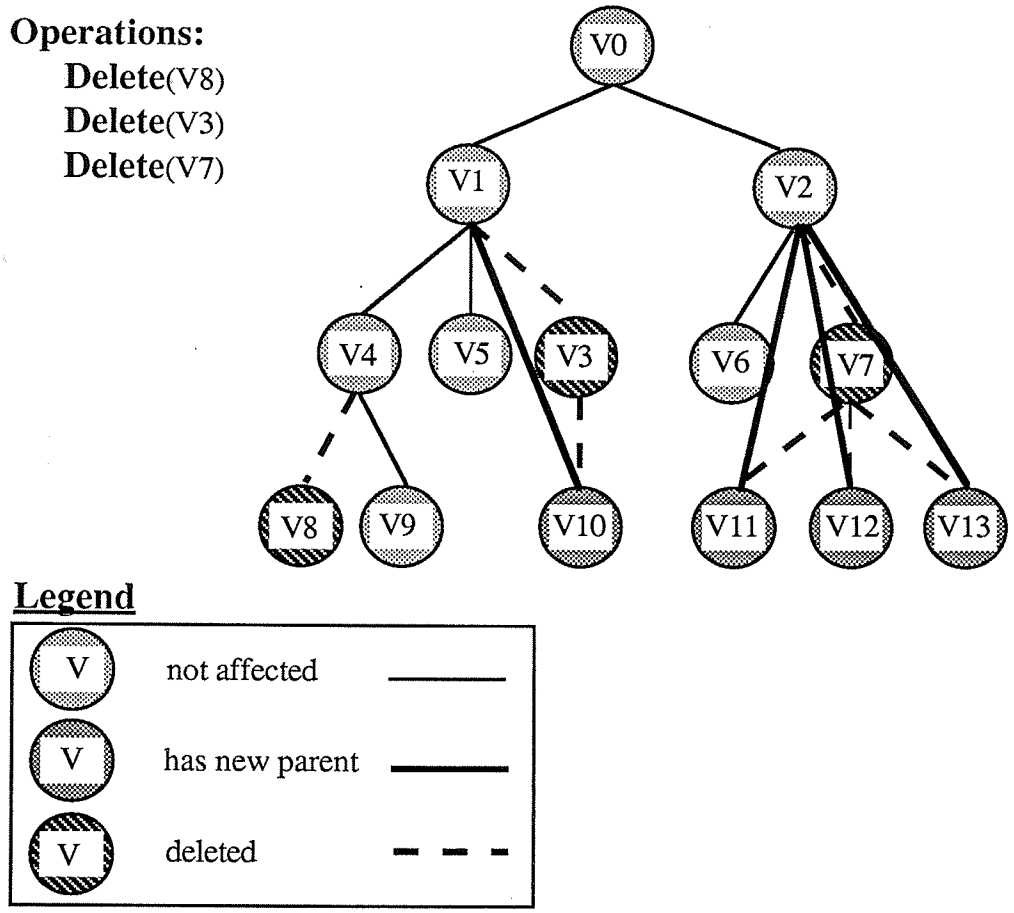


Figure 8 Delete Operation

• **Prune**(version_Id):

The Prune operation deletes the target version and all its descendent versions. The Prune operation is applicable to both nodes and leaves, but will usually be used for nodes, in order to prune a subtree of the Version Tree.

For pruning a subtree, the Version Tree is searched to locate the target version (root of the subtree). The subtree is then pruned recursively. Figure 9 shows some **Prune** operations that have been applied to the Version Derivation Tree of Figure 6.

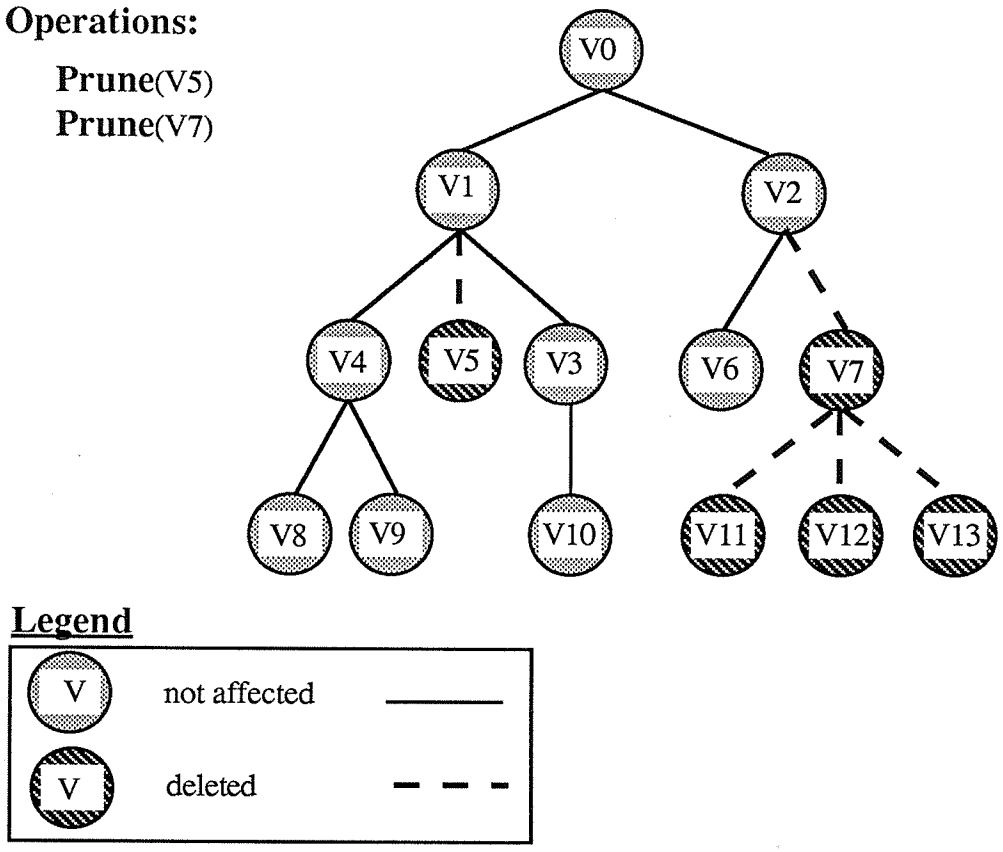


Figure 9 Prune Operation

• **Graft**(version_Id,parent_Id):

The Graft operation moves a single existing version or version subtree to a new location in the Version Tree. Grafting is done only upward, along the derivation path of the version (i.e., the version to be grafted must be a descendent of the version that it is to be grafted to).

For grafting a version, the Version Tree is searched to locate both the parent version and the target version (the version to be grafted). The target version is then grafted to the given parent. Figure 10 shows some **Graft** operations that have been applied to the Version Derivation Tree of Figure 6.

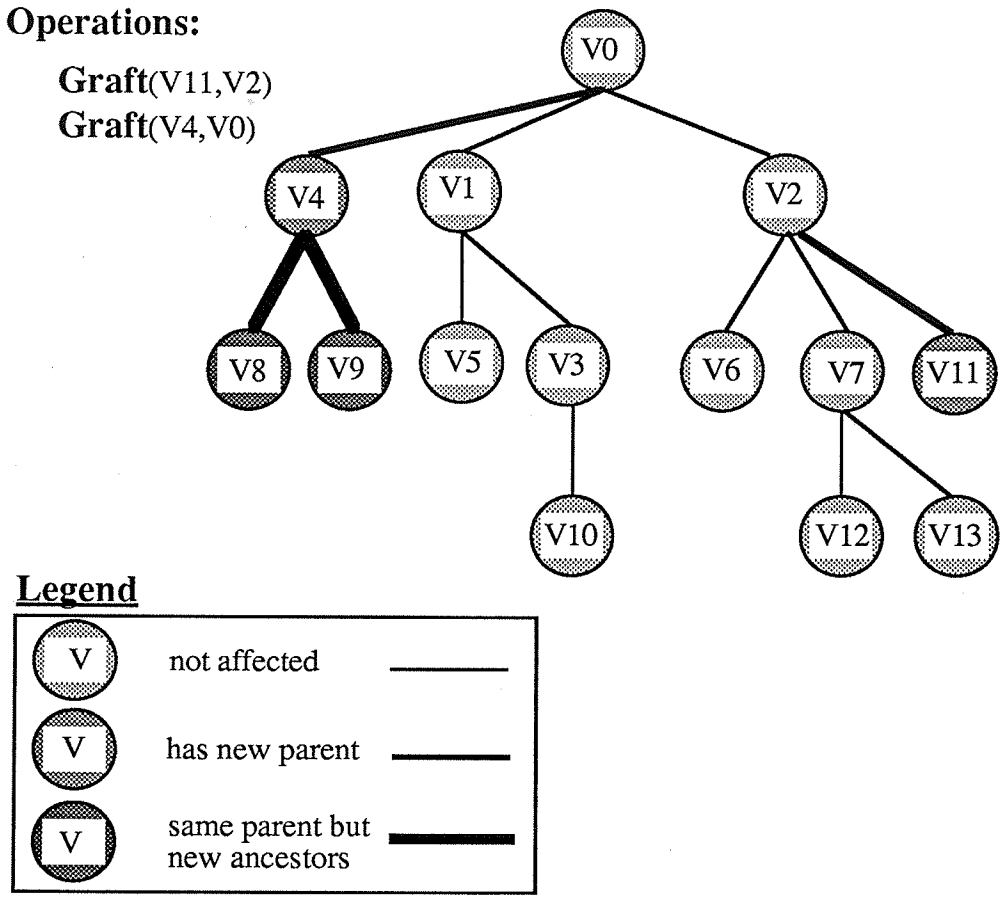


Figure 10 Graft Operation

• **Promote**(version1,version2):

The Promote operation is for promoting a node or leaf version along its derivation path (i.e., version1 must be a descendent of version2). When version1 is promoted to version2, version2 is deleted together with all its descendents. For promoting a version, the Version Tree is searched to locate both version1 (the version to be promoted) and version2 (the version it is to be promoted to). The subtree of version1 is grafted to the parent of version2. The subtree of version2 is then pruned. Figure 11 shows some **Promote** operations that have been applied to the Version Derivation Tree of Figure 6.

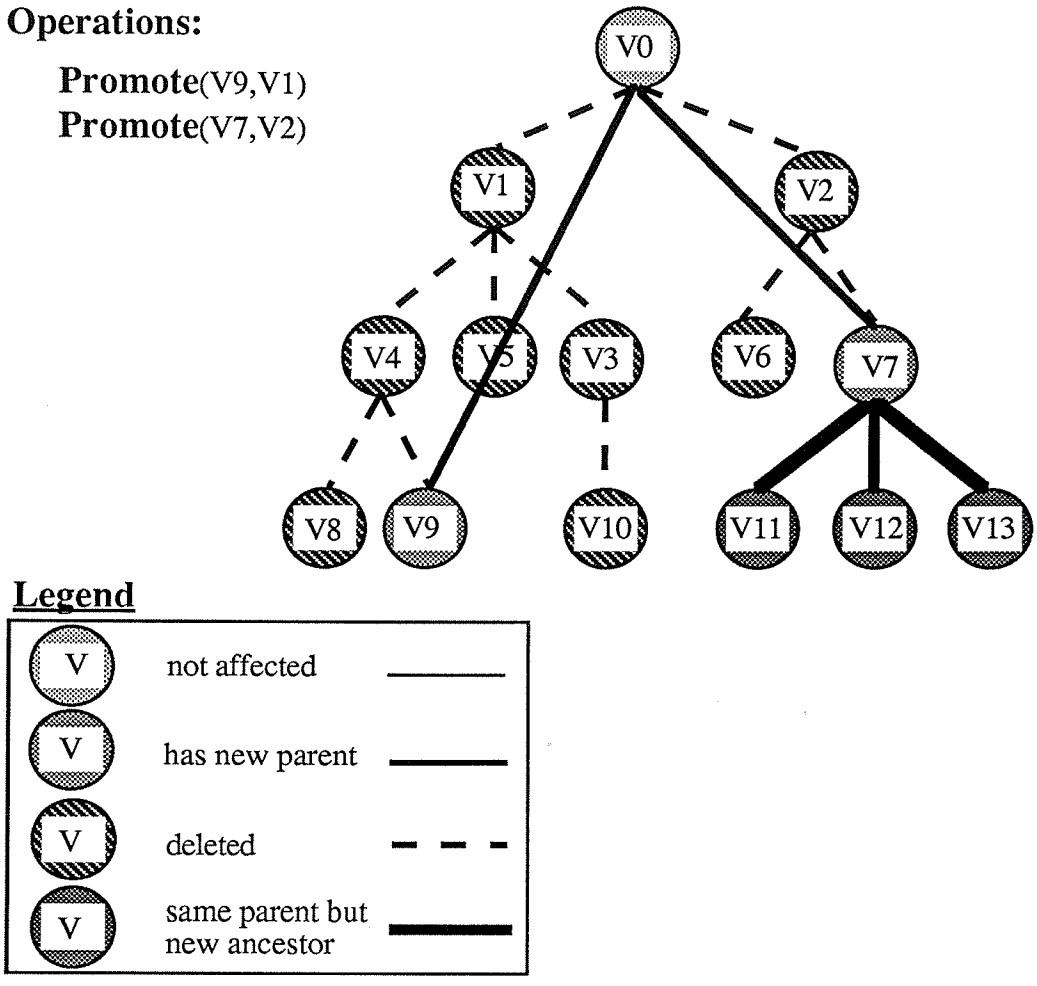


Figure 11 Promote Operation

6.3.2 Query Operations

There are many query operations that could be defined [1]. In this section a sample of some possible queries is shown.

• **VersionTime**(version_Id):

Return the time at which the version was created.

• **VersionStatus**(version_Id):

Return an indicator showing whether the version is versionable or non-versionable.

• **SetVersionCurrent**(version_Id):

Make the given version the current version. By default, operations are performed on the current version.

• **GetCurrentVersion**():

Return the identifier of the current version.

• **YoungestVersion**():

Return the identifier of the youngest version created (i.e., the last version), by comparing the times of creation.

• **OldestVersion**():

Return the identifier of the oldest version created (i.e., the first version), by comparing the times of creation.

• **VersionParent**(version_Id):

Return the identifier of the version from which version_id is derived.

• **VersionLevel**(version_Id):

Return an indicator identifying whether the version is a leaf or a node.

• **VersionCount**():

Return the number of versions in the version tree.

• **VersionLeafCount**():

Return the number of leaf versions in the version tree.

• **VersionNodeCount**():

Return the number of node versions in the version tree.

• **VersionLevelsCount**():

Return the number of version levels in the version tree (i.e., depth of the version tree).

• **VersionChildrenCount**(version_Id):

Return the number of immediate child versions of the given version.

• **VersionAncestorCount**(version_Id):

Return the number of ancestor versions of the given version.

• **VersionDescendentCount**(version_Id):

Return the number of ancestor versions of the given version.

• **VersionSiblingCount**(version_Id):

Return the number of sibling versions of the given version.

• **VersionRelationship**(version1_Id,version2_Id):

Return an indicator showing the relationship of version1 to version2 (i.e., Child-of, Parent-of, Ancestor-of, Descendent-of, Sibling-of).

7. Creation and Management of Master Objects and Versions

The preceding section described the types of operation which might be performed by a Version Manager once a Master Object has been created. However, it did not address issues related to creation and management of objects. Some of these issues are: (a) creating a new Master Object and its associated Version Manager; (b) creating new version objects; (c) making a Master Object and all of its versions persistent by saving them in a data base; and (d) activating these objects in a design application, so that design and version management operations can be performed [1]. These issues are considered briefly in this section.

The overall control procedure which is envisaged is as follows. First, a new Master Object is created, by sending a *new* (or *construct*) command to the appropriate design object class. The Master Object then creates a Version Manager object, by sending a *new* command to the version manager class. The Master Object "owns" the Version Manager object. Version management operations are performed by sending commands to the Master Object, which routes them to the Version Manager. The Version Manager then performs the necessary operations. These operations include creating new objects and initializing their inherited (frozen) attribute values.

It is also envisaged that design objects are either *active* or *inactive*. An active object is directly available to a Design Application Program, and is an instance of a class which mainly emphasizes design operations. An inactive object is a persistent copy, and may be of a different class, which mainly emphasizes permanent storage in a project data base. Object management operations involve activating and saving Master Objects and their versions. It is important that a Master Object, its Version Manager and all of its version objects be treated as a unit for these operations. That is, a Design Application Program seeking to activate or save a set of object versions should direct the

activate or save command to the Master Object, and all of its related objects should automatically be activated or saved. This is why the Version Manager is created and owned by the Master Object, since it means that only the Master Object needs to be visible to a Design Application Program.

This approach means that Design Application Programs need not be concerned with any of the details of object management. However, it also means that all design objects must have the capability to act as Master Objects. This can be achieved by requiring that all design object classes be subclasses of a base class with the required capabilities. Among other things, the design object base class must have the ability to create Version Manager objects, to forward commands to these objects, and to interact with a data base management system to activate and save a Master Object and all of its related objects.

8. Some Details for the Implementation of the Version Manager

Each Version Manger object is an instance of the Version Manager class. The Version Manager class is a subclass of a more general class, namely, Manager class as shown in Figure 12a. The Version Manager define attributes and methods needed for managing versions of the master object. Therefore, the Version Manager object (instance of the Version Manager class) must store information on the tree structure. It must also store information on each version, such as its time of creation, its status, its owner, etc. There are many data structures that could be used to implement the Version Manager class, but since the number of versions in any version tree are likely to be small, its probably reasonable to use a table structure with one row for each version in the version tree. The table can be implemented as a linked list or array. Figure 12b shows a possible implementation of the Version Manager class. Some of the data items to be stored for each version might be as follows:

- Version Identifier.
- Identifier of parent version.
- Time and date of creation.
- Version status (i.e., whether versionable or non-versionable).
- A version owner.
- A version address in memory or location in disk.
- Other attributes to simplify the query operations, such as version
- siblings count, version immediate children count, version ancestors count, and so forth.

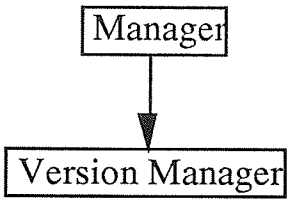


Figure 12a Version Manager Class

Version id	Parent id	Time	Status	Level	Sibling #	Child #	Owner	Version Pointer
V1	V0							
V2	V0							
V3	V1							
V4	V1							
V5	V1							
V6	V2							
V7	V2							
V8	V4							
V9	V4							
V10	V3							
V11	V7							
V12	V7							
V13	V7							

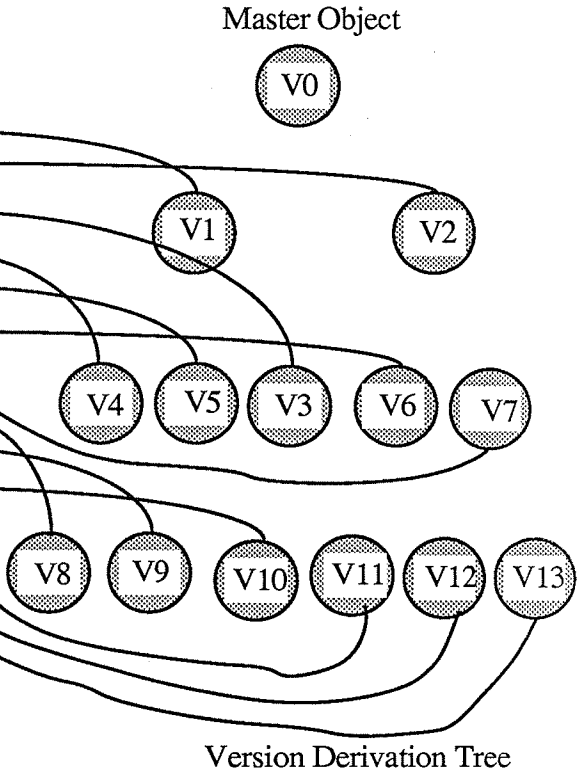


Figure 12b Version Manager Details

9. Conclusion

A scheme for defining and organizing design object versions has been proposed. The features required for a Version Manager which managers design object versions have been identified, and several specific operations for managing versions have been defined. Some broader issues related to object management have also been noted.

Operations at three levels must be performed, namely (1) design operations, performed by an application program on specific design object versions, (2) version management operations,

performed by the version management system under the control of an application program, and (3) object management operations, involving interaction with a data base, also performed by the version management system under the control of an application program. An important requirement is that an application program should be able to treat the set of versions for any design object as a unit, without worrying about the details of version management.

This paper has described the overall needs for version management, including the need to consider not only derived versions but also alternative design objects and decompositions of design objects. The proposed version management system considers only version management, and the features and operations for this system have been described in detail. Additional features would be needed to manage alternatives and decompositions. Several issues associated with higher level object management have been identified, but these issues have not been addressed in any detail.

Some features of the proposed Version Manager are as follows.

1. The versions for any design object are organized in a version tree, with a Master Object at the root. The Master Object owns a Version Manager object, which performs version management operations such as adding new version objects to the tree and reorganizing the tree structure.

2. Each version in the version tree is derived from the Master Object, with strict rules for inheritance of attribute values and for operations on the tree structure.

3. Commands defining version management and object management operations are sent to the Master Object. Commands defining design operations are directed to specific versions, but are also routed through the Master Object. Hence, the details of version and object management are hidden from application programs.

4. All design object classes must be subclasses of a root class which has capabilities to create Version Manager objects, route commands to the Version Manager, and interact with a data base management system.

The implementation of a version management system is clearly a major task, but is necessary if truly integrated design software is to be created. The study described is part of a broader effort to develop concepts for the design of integrated software for structural engineering. Other parts of the effort involve investigations of data base needs, with particular emphasis on object-oriented data bases [3,20], studies on the effective application of object-oriented concepts in structural engineering design [1,21,22,23], development of a model for the structural design process [24,25,26,27], automated modelling for structural analysis [28,29,30], and new geometrical modelling concepts to support design [31].

10. Acknowledgement

The research described in this paper has been supported in part by a grant from University of Khartoum, Sudan, and in part by National Science Foundation Grant No. DMC-8517338. The Center for Integrated Facility Engineering at Stanford University has facilitated the facilities for producing this report. The support of these organizations is gratefully acknowledged.

11. References

[1] Abdalla, J. (1989) Object-Oriented Principles and Techniques for Computer Integrated Design.*Unpublished PhD Dissertation*, University of California at Berkeley.

[2] Sause Jr, R. (1989) A Model of the Design Process for Computer Integrated Structural Engineering. *Unpublished PhD Dissertation*, University of California at Berkeley.

[3] Bhateja, R. (1989) A Database Design for Structural Engineering. *Unpublished PhD Dissertation*, University of California, Berkeley.

[4] Klahold, P., Schlageter, G., and Wilkes, W. (1986) A General Model for Version Management in Databases. *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, pp. 319-327.

[5] Gladney, H. M., Lorch, D. J., and Mattson, R. I. (1987) A Version Management Method for Distributed Information. *IEEE 1987 International Conference on Data Engineering*, Los Angeles, pp. 570-574.

[6] Bjornerstedt, A., and Britts, S. (1988) AVANCE: An Object Management System. *OOPSLA '88 Proceedings*, San Diego, California, USA, pp. 206-221.

[7] Dittrich, K. R., and Lorie, R. A. (1985) Object-Oriented Database Concepts for Engineering Applications. *IEEE 1985 Compint Computer-Aided Design Technologies*, pp. 321-325.

[8] Dittrich, K. R., Gotthard, W., and Lockemann, P. C. (1986) Complex Entities for Engineering Applications. *Proceedings of the Fifth International Conference on Entity-Relationship Approach*, pp. 421-440, Dijon, France.

[9] Chou, H-T., and Kim, W. (1986) A Unifying Framework for Version Control in a CAD Environment. *Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 336, Kyoto, Japan.

[10] Dittrich, K. R., and Lorie, R. A. (1988) Version Support for Engineering Database Systems. *IEEE Transactions on Software Engineering*, Vol. 14, No. 4, pp. 429-437.

[11] Katz, R. H., Chang, E., and Bhateja, R. (1985) Version Modeling Concepts for Computer-Aided Design Databases. *Report No. UCB/CSD 86/270*, EECS, University of California, Berkeley.

[12] Rieu, D. and Nguyen, G. T. (1986) Semantics of CAD Objects for Generalized Databases. *Proceedings of the 23rd Design Automation Conference*,Las Vegas, pp. 34-40.

[13] Zdonik, S. B. (1986) Version Management in an Object-Oriented Database. *Proceedings of the International Workshop on Advanced Programming Environments*, Trondheim, Norway, pp. 139-200.

[14] Davison, J. W., and Zdonik, S. B. (1986) A Visual Interface for Database with Version Management. *ACM Transactions on Office Information Systems*, Vol. 4, No. 3, pp. 226.

[15] Ecklund, D. J., Ecklund, Jr., E. F., Eifrig, R. O., and Tonge, F. M. (1987) DVSS: A Distributed Version Storage Server for CAD Applications. *Proceedings of the 13th VLDB Conference*, Brighton, England, pp. 443-454.

[16] Katz, R. H., Anwarrudin, M., and Chang, E. (1986) A Version Server for Computer-Aided Design Data. *23rd Design Automation Conference*, Las Vegas, pp. 27-33.

[17] Beech, D., and Mahbod, B. (1988) Generalized Version Control in an Object-Oriented Database. *Proceedings of the 4th International Conference on Data Engineering*, pp. 14-22, Los Angeles.

[18] Batory, D. S., and Buchmann, A. (1976) Molecular Objects, Abstract Data Types and Data Models: A Framework. *Proceedings of Very Large Data Bases*, pp. 9-36, Vol. 1, No. 1.

[19] Wiess, S. et. al. (1986) DOSS: Design Object Storage System. *Proceedings of the 23rd Design Automation Conference*, Las Vegas, 41-47.

[20] Powell, G. and Bhateja, R. (1988) Data Base Design for Computer-Integrated Structural Engineering. *Engineering with Computers*, Vol. 4, 135-143.

[21] Powell, G. and Abdalla, J. (1989) Object Management in an Integrated Design System. *Proceedings, ASCE Sixth Conference on Computing in Civil Engineering*, Atlanta, pp. 556-563.

[22] G. Powell, R. Bhateja, J. Abdalla, H. An-Nashif, K. Martini and R. Sause (1988) A Database Concept for Computer Integrated Structural Engineering Design. *Proceedings, 5th Conference on Computing in Civil Engineering*, Alexandria, VA, pp. 521-529.

[23] G. Powell, J. Abdalla and F. Filippou (1988) An Object-Oriented Approach to Computer Aided Reinforced Concrete Design. *Proceedings, 3rd International Conference on Computing in Civil Engineering*, Vancouver, Canada, pp. 119-126.

[24] Sause, R. and Powell, G. (1990) Design Process Model for Computer Integrated Structural Engineering. *Engineering with Computers*, Vol. 6, 129-143.

[25] Sause, R. and Powell, G. (1989) A Model for Knowledge-Based Structural Design. *Proceedings, ASCE Sixth Conference on Computing in Civil Engineering*, Atlanta, pp. 170-177.

[26] Sause, R. and Powell, G. (1988) Knowledge Representation and Processing for Computer Integrated Structural Design. *Proceedings, 5th Conference on Computing in Civil Engineering*, Alexandria, VA, pp. 1-10.

[27] Powell, G.; Abdalla, J. and Sause, R. (1989) Object-Oriented Knowledge Representation: Cutes Things and Caveats. *Proceedings, ASCE Sixth Conference on Computing in Civil Engineering*, Atlanta, pp. 556-563.

[28] An-Nashif, H. N. (1989) "Automated Structural Analysis for Computer Integrated Design", *Unpublished PhD Dissertation*,, University of California, Berkeley.

[29] Powell, G. and An-Nashif, H. (1988) Automated Modelling for Structural Analysis. *Engineering with Computers*, Vol. 4, 173-183.

[30] Powell, G. and An-Nashif, H. (1988) Automated Structural Analysis in a Computer Integrated Design System. *Proceedings, 3rd International Conference on Computing in Civil Engineering*, Vancouver, Canada, pp. 29-36.

[31] Martini, K. and Powell, G. (1990) Geometric Modeling Requirements for Structural Design. *Engineering with Computers*, Vol. 6, 93-102.