Version Management of Composite Objects in CAD Databases

Rafi Ahmed
Hewlett-Packard Laboratories
Palo Alto, CA 94304
rafi_ahmed@hplabs.hp.com

Shamkant B. Navathe
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

There are essentially two characteristics of design objects, which impact upon the iterative and exploratory nature of the design process. First, they are hierarchically formed assemblies of component objects. Second, they go through phases of design giving rise to multiple versions. This paper addresses a broad spectrum of modeling and operational approaches related to these two aspects in computer-aided design databases.

A classification of composite object attributes into intrinsic interface attributes and nonintrinsic internal assembly attributes provides an insight into their behavior and functionality. The design process is studied with respect to these classes of attributes. A design evolves in discrete states through mutation and derivation and may cause version proliferation. A notion of version equivalence, based on invariant attributes and value-acquisition, is used to prevent version proliferation. States of versions, which determine their updatability, and operations on these states are described. A methodology for version management at the type and instance levels and solutions for the related problems are presented.

1.0 Introduction

In recent years, there has been a tremendous surge of interest in the research and development of computer-aided design (CAD) systems for aiding and controlling the design efforts of a wide variety of engineering products, including VLSI circuits, mechanical parts, software systems, etc.

The iterative and exploratory nature of the design process leads to two aspects of design objects which must be dealt with. First, they are usually complex; that is, they are assemblies of components that themselves may be constructed hierarchically from constituent objects. Second, there can be several alternative descriptions, called versions, of a design object. Management of design data in the presence of alternatives and assembly thus becomes a difficult task. The problem of managing a design is compounded by the fact that composite objects can have multiple alternative configurations, if versions of components are taken into consideration.

Traditional database systems, designed to deal with only regular and structured data with no built-in concepts of versioning, cannot efficiently manage design data. We need a unified system which incorporates these requirements of design data and provides tools for efficiently dealing with composite objects and their version management.

This paper develops a concrete approach to the definition, representation, manipulation and management of versions. This work is a part of an extensive scheme for version control, manipulation, and storage.

1.1 Our Approach

In this section, we briefly describe the general approach we have taken for version management. Our approach is consistent with object-oriented concepts and hence has a wide scope for applications.

We present a classification of composite object attributes into intrinsic interface and nonintrinsic internal assembly attributes. The design process is studied with respect to these classes of attributes.

This distinction among attributes leads to a meaningful versioning scheme for composite objects. Concepts of invariant interface attributes and value acquisition are used for developing a notion of version equivalence.

We discuss a detailed versioning scheme governed by the classification of composite object and versioned object attributes and different states of versions. A mechanism for interrelating disparate types of objects is described through notions of generic objects and value-acquisition. A methodology is proposed to deal with versioning at the instance and template levels. We believe that the proposed versioning scheme can be implemented over any object-oriented platform.

1.2 Related Work

Complex objects and version management are the two aspects of related work that are reviewed below.

Many recent investigators have recognized the need of composite objects in CAD applications [Lor83, Emo83, Sto83, Bat84, Bat85, Kim87, Kim89, Nav90]. [Bat84] develops a framework for modeling complex objects and classifies them into four different types. [Sto83] introduces user-defined ADT's (Abstract Data Types) and ADT indices to relational DBMS INGRES for supporting representation and retrieval of composite objects. The use of ADT simplifies the definition and logical manipulation of composite objects by allowing user defined data types and operations on these types. [Bat85] proposes modeling concepts for VLSI objects. They use entity-relationship diagrams to represent these objects which have interface descriptions and implementation descriptions. The modeling framework provided in this work is very useful in representing composite objects and their versions. [Kim89] discusses the facilities for the composite objects in the ORION data model by introducing the "is-part-of" relationship between objects. Composite objects in ORION are simply an aggregation hierarchy with integrity and existence dependencies.

The deficiencies of current DBMS's with respect to the management of refinements, alternatives and versions of designs have been recognized and discussed in the literature extensively [Kat82, Joh83, McL83, Kat86, Kla86, Lan86, Bee88, Ket88, Nav88, Kim89]. In the scheme proposed by [McL83], both refinements and alternatives are called "revisions." This proposal uses Tich's AND-OR graph model to abstract the state of a system at an arbitrary stage of its design. Representations are considered a kind of versions which are equivalent objects describing the same objects. [Kat86] deals with three aspects of design: version histories, configurations, and equivalence among objects of different types; in this proposed design, descriptions may exist across representations. [Lan86] discusses some important concepts of versioning and propagation of changes. Changes in this scheme can be grouped together that can postpone automatic generation of versions. [Cho86] presents a model based on classification of versions and message-passing protocols for incorporating the distributed nature of CAD environment and the complex configurations of design objects.

The point that emerges from the study of related work is that there is a clear lack of cohesion among different schemes, and that some approaches for versioning have been proposed without a concrete underlying data model. Furthermore, the proposed schemes have not accounted for the fundamental nature of design data, where nonintrinsic and intrinsic attributes play different roles.

The remainder of the paper is organized as follows. In Section 2, some basic modeling constructs for design databases are described. In Section 3, the abstraction developed for a complex object is discussed. Section 4 deals with the basic modeling constructs and concepts for versioning; Section 5 provides a scheme for managing version control. Finally in Section 6, we summarize the contributions of this paper.

2.0 Modeling Constructs for Design Databases

In this section, we present modeling constructs for design databases. The constructs are essentially based on the ideas proposed in existing functional, semantic and object-oriented data models. These constructs will be used as an underlying model for the concepts developed in this paper for the version management in design databases [Ahm89].

The universe of discourse is viewed as a collection of objects. These objects are a convenient aggregation of information describing real world concepts.

The basic constructs of the model are *types (T), functions (F), instance objects* (*IO*) and *class* (*C*), which form a partition of the universe of discourse.

Objects of similar behavior are grouped together and are said to have the same type. Instance objects belong to one or more types.

An essential concept in modeling the properties and interrelationships of objects is that of a function. *Functions* are named objects that take one or more

arguments and return result values; that is, they provide mappings among objects.

A *class* is defined to be a set of objects, which either are enumerated or satisfy some constraints and belong to *exactly one* of the constructs, *T*, *F*, *C*, *IO*. Class is a useful concept needed in design applications; for instance, a class of functions defined on a type can be used to model its behavior. By defining a hierarchy of subtypes of *C*, different constraints can be imposed on functions or types.

The supertype/subtype relations among user-defined objects are represented by a rooted, labeled, directed, acyclic graph, which captures generalization/specialization. An instance of any type is also an instance of all its supertypes, and thus inherits a subset--not necessarily proper--of the operations and functions of its supertypes. The label associated with an edge in this graph is a class of functions which is inherited by the successor node from its predecessor node.

We also need a notion of *value acquisition*. This is closely related to the idea of function (attribute) inheritance. The difference is that value acquisition is defined for object *instances* rather than types. The instance that acquires values will be called *receptor* and the instances from which values are acquired will be called *transmitter*. The type of a receptor either is of the same assigned-type as that of its transmitter or is its subtype. Value acquisition means that a given class of functions on a receptor return the result values which are acquired from its transmitter. A receptor is not allowed to update the result values of these functions. As in case of attribute inheritance, this notion is further refined by allowing partial value acquisition. The use of value-acquisition in design databases will become clear in Section 4.2.

Design databases require a facility for defining and manipulating a hierarchically structured set of objects as a single logical entity. We call such an entity a *composite* type object, which is defined as a directed acyclic graph where the nodes with an in-degree of zero are the types of the highest level composite objects, the nodes with an out-degree of zero are the types of primitive objects. The directed edges in the graph represent the "*is-composed-of*" relationship between a composite object and its constituents. We call this graph the *composition graph*.

The constructs of types, functions, and class, the abstractions of generalization and composition graphs, and the notion of value-acquisition provide a basic framework for version management.

3.0 An Abstract View of Composite Objects

A composite object, as represented by the composition graph described in Section 2, is a recursively defined aggregation of its constituent objects. It is one of the premises of this work that composition, though fundamentally important, is not completely adequate for modeling objects in a complex engineering design. Relationships among a composite object and its constituent objects must also be explicitly represented.

The familiar abstractions of white-box and black-box properties can provide some insight into the nature of design objects. The properties that describe an object without any concern for its internal structure are viewed as black-box properties. The properties that provide a complete description of an object to the external world are looked upon as white-box properties. In the following, we propose a classification of the properties of design objects based on the above abstraction. This classification provides a framework for dealing with a composite object and its version representations.

External features (EF) of a design object are its non-structural attributes that are visible to the external world.

Internal assembly (IA) of a composite design object is its structural attributes that identify its constituent objects and describe their interrelationships.

A *primitive* object does not have any internal assembly properties. It possesses only the external features. In the composition graph, primitive objects form the vertices that have an out-degree of zero.

Thus external features correspond to black-box properties, as the internal details are not seen, whereas both internal assembly and external features correspond to white-box properties, as entire object becomes visible. This classification also provides a paradigm for understanding the concept of versioning as it relates to composite objects.

The external features can be subdivided into *descriptive* (D_S) and *interface* (I_F) attributes, whereas the internal assembly can be subdivided into *composite aggregation* (C_A), *interconnection* (I_C), and *correspondence* (C_R) attributes.

Interface (I_F) properties are involved in the abstraction of a design object. These properties define links through which it interacts with the external world, i.e., with the user and other design objects. Interface properties are crucial in understanding the nature and functionality of a design

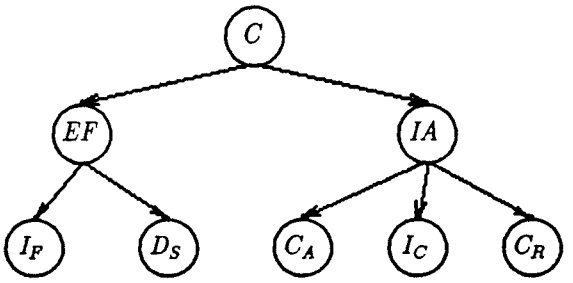

Figure 3-1 Attribute Classification for Composite Objects

object, as they are visible to the external world and present a higher level of abstraction of its internal assembly.

Descriptive (D_S) attributes provide identification and non-interface descriptions of a design object. Properties such as *Name, Designer* etc. are categorized as descriptive attributes.

Composite aggregation (C_A) of a design object identifies its immediate constituent objects. This property is captured by one level of the composition graph defined in Section 2.1.

Interconnection (I_C) attributes of a composite object represent the interconnections among its constituent objects. These interconnection attributes involve only the interface attributes of the constituent objects.

Correspondence (C_R) attributes of a composite object represent the connection between its interface and the interfaces of its constituent objects.

We believe that the abstraction of composite objects presented here is *independent* of any data model. In Section 2, we introduced the system-defined construct C and the notion of class. Other system-defined classes appear as subtypes of C. The classes of functions defined here will have the following type structure. The new system-defined classes, EF and IA, become the subtypes of C. The classes, I_F and D_S, are subtypes of EF, and they represent the sets of attributes that constitute external features. This can be diagrammatically represented as shown in Figure 3-1.

Functions defined on composite objects can be declared as the members of these classes. Enforcement of constraints on the classes of functions is facilitated by the type structure shown in Figure 3-1.

The afore-mentioned subdivision of external features and internal assembly may need some enhancement for applications in structural or mechanical engineering design.

3.1 An Example of A Composite Object

In this section, we present an example of a composite object using the constructs defined in Section 2.0. Figures 3-2 and 3-3 [Bat85] show the interface specification and internal assembly of a circuit representation of a 4-bit adder respectively. A pair of 4-bit numbers (X, Y) are inputs, a 5-bit number representing their sum (Z) is its output. It can have descriptive attributes, *Name, Designer*, etc., which are not shown in the figure. The internal assembly and interface of this adder are shown in Figure 3-3. It contains four adder-slices. The interface properties of adder-slice are inputs X, Y (1-bit numbers) and C_i (the carry from a previous slice) and outputs C_o (carry) and Z (1-bit sum). The following functions can be used to model the object adder and its interface and internal assembly.

function Name (*adder*) → charstring
function Designer (*adder*) → charstring
function Output (*adder*) → *terminal*
function Input (*adder*) → *terminal*
function Contains (*adder*) →
  < *adderslice, adderslice, adderslice, adderslice* >
function Link (*adder, terminal*) →
  <*adderslice, terminal*>
function Connection (*adder, adderslice, adderslice*)
  → <*terminal, terminal*>

In the above example, the user-defined types *adder, adderslice*, and *terminal* belong to the system-defined type T and the functions Name, Designer, Output, Input, etc. belong to the system-defined type F (Section 2.0).

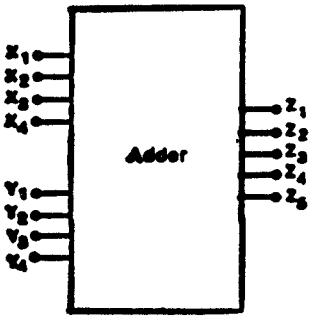

Figure 3-2 Interface of Adder

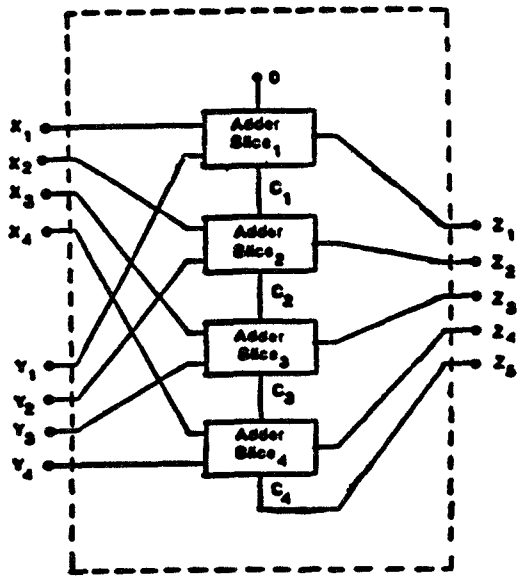

Figure 3-3 Adder

Functions defined on a type can be declared as members of the classes shown in Figure 3-1. For instance, we have the following classification for the functions defined on type, *adder*.

{ Name, Designer } $\in D_S$	(Descriptive)
{ Output, Input } $\in I_F$	(Interface)
{ Contains } $\in C_A$	(Composite Aggregation)
{ Link } $\in C_R$	(Correspondence)
{ Connection } $\in I_C$	(Interconnection)

We believe that this classification allows us to deal with different behavioral and update constraints on the attributes of a composite object. It further provides a basis for dealing with the problem of version control and management.

4.0 Concepts for Version Management

Generally, design of an object starts with a high level description of some aspects of that object. A design object retains its identity while it evolves through various stages of design. *Versions* are distinct snapshots of a design object in different states and share some identifiable common characteristics. The versions of a design object contain qualitatively or quantitatively different information, but they evolve in the direction of an ultimate design goal.

There is considerable debate in the literature as to when two instances of the same type are different objects and when they are merely different versions of the same object. The key to deciding this issue is to view each attribute of a design object as either intrinsic or nonintrinsic. If *intrinsic* properties change, so does the object. *Nonintrinsic* properties, on the other hand, can be modified without changing the object in any significant way. Interface is an example of intrinsic attributes, whereas properties such as "designer", "co-ordinate position", and even internal assembly are examples of nonintrinsic attributes.

4.1 Basic Premise of Our Approach

One of our basic premises is that different versions of a design object share the same interface but may have different internal assemblies. If the interface properties change, this implies that the object itself has changed in some fundamental way. Several arguments can be put forth in support of this hypothesis.

1. Interface defines the functionality of the object and is visible to the external world. Design objects interact -- correspond, interconnect -- with one another through their interface. Therefore, any modification of the interface of an object may disqualify it from being referenced as a component by its composite object.

2. Changes in the internal assembly of an object, if its interface remains invariant, are not visible to the external world and may thus warrant creation of only a different version.

Therefore, we maintain that different versions of an object share the same interface properties and some descriptive attributes but can differ from one another in their internal structure both at the type and instance levels.

4.2 Generic, Versioned and Unversioned Objects

In this section, we propose three system-defined types, as the basic framework for our proposed methodology for version management.

Each design object, in addition to possessing its user-defined types, belongs to one of three system-defined types: *Generic*, *Versioned* or *Unversioned*. Generic objects are instances of type *Generic*. The instances of type *Versioned* are versioned objects. Design objects which are not going through a process

of evolution are instances of type *Unversioned*. These three types are mutually exclusive.

The common characteristics that are used to relate all versions of a design object are called its *invariant* properties. Invariance, however, does not imply that they can never be modified; it means that they are invariant over the version set.

A *generic* design object is characterized by its invariant external features, which subsume the interface properties. Therefore, a generic object identifies the design object, represents its essence, and contains a high level abstraction of its functionality.

In this model, all versions of a design object, which are allowed to have different types, value-acquire the invariant external features from its generic object. The type of a versioned object appears in the generalization graph as a subtype of the generic object type, if they are of different assigned-types. Furthermore, the versioned objects become receptors with the generic object instance as their transmitter. There is only one generic object for each version set. This scheme provides a paradigm that relates all versions of a design object and also ensures that they have identical interface attributes.

4.3 Version Graph

A version graph captures the evolution history of versioned objects. We define a *version graph (VG)* as a disconnected, directed, acyclic graph. The nodes in the graph are the versions that belong to a generic object. The edges of this graph represent the successor/predecessor relationship.

Figure 4-1 shows an example of a version graph. D.g is the generic object from which every version in the version graph value-acquires the interface properties. The roots of the subgraphs, D.v1 and D.v6, are the objects that are initial versions. These versions can be viewed as different alternatives of a design. The non-root versions in a subgraph can be looked upon as the refinements of the alternative design that forms the root of a subgraph. For example, D.v5 has been derived from D.v1, and is thus its refinement. Non-root versions in a subgraph are derived from some other version of the subgraph, and hence have the same user-defined types; for example, the versions D.v1, D.v2 and D.v5 have the same user-defined types. As mentioned earlier, the roots of subgraphs are not necessarily of the same type; that is, D.v1 and D.v6 may have different user-defined types.

All versions in a version graph constitute a *version set*; in Figure 4-1, versions D.v1 through D.v11 form a version set. Versions in the version set are considered equivalent, as they have identical invariant attributes. In the forthcoming discussion, we use the term *equivalent* versions for the members a version set.

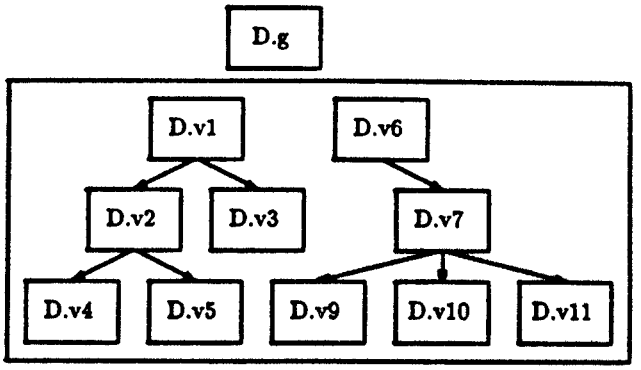

Figure 4-1 An Example of Version Graph

4.4 Versioning at the Type and Instance Levels

In the literature, it has often been felt that versioning should be allowed both at the type and instance levels. Versioning at the instance level implies that different versions of the same object differ only in the *values* of some of their properties. Versioning at the type level, on the other hand, means that different versions of an object can have different assigned types. It has an added implication that modifications of a template--a type and a collection of attributes defined on it--creates another template version. Furthermore, there is an underlying assumption that a template is adequate for providing a complete description of a design object.

Most systems, however, provide versioning at the instance level, since versioning at the type level is not without its problems. First, it is quite a deviation from the traditional notions of data and databases. Second, templates often prove to be insufficient in providing a complete description of a design object, since many attributes still need instantiated values. Third, in many applications, particularly in software design, versioning at the type level has no relevance at all.

The proposed scheme brings about a compromise between the two opposing views. Versioning remains at the instance level in the sense that all information is required to be instantiated. However, versions can have entirely different types and thus different internal assembly properties. These disparate types of versions are related together by the generic object and the mechanism of value-acquisition of invariant attributes.

4.5 Classification of Version Attributes

To model the dynamic behavior of versions, the system should be able to control and record significant changes in the evolution history of design. This requires categorizing the functions of versioned objects into different classes. It should be emphasized that a design object itself is a complex structure, which has different classes of attributes even in the unversioned state.

We present a classification of attributes of versioned design objects. The notion of class, defined in Section 2, is used to model the behavior of a design object on the updatability of various groups of functions. This classification of versioned object attributes is quite independent of the classification of composite object attributes presented in Section 3. The present classification determines the degree of updatability of a versioned object.

The three classes of attributes, invariant, *version significant* and *non version-significant*, provide a mechanism for controlling update propagation in versions.

Version-significant attributes can be updated only in a non-destructive manner. Conceptually, modifications in one of these attributes create a new derived versioned object bearing the change. Such a creation of a new version is called *mutation*. There is an important exception to this rule. The replacement of equivalent versions by one another in the internal assembly functions, though defined to be version-significant, does *not* cause mutation. This rule provides a mechanism for effectively controlling unnecessary proliferation of versions. In the forthcoming discussion, we refer to this update as *equivalence modification*.

Nonversion-significant attributes of an object can always be modified without causing any mutation.

The invariant attributes of a versioned object cannot be modified at the version level, since they are value-acquired from the generic object.

The three classes, invariant, version-significant and nonversion-significant, can be defined as subtypes of C, and different functions can accordingly be declared as their members.

Table 4-1 shows correspondence between the classes of composite and versioned object attributes. For instance, I_F is classified as invariant and IA as version-significant. $D_S{}^1$ and $D_S{}^2$ form a partition of D_S. $D_S{}^1$ is invariant, whereas $D_S{}^2$ is nonversion-significant. The classifications of composite object attributes is illustrated in Figures 3-1.

	Invariant	Version siginificant	Nonversion significant
$D_S{}^1$	Yes	No	No
$D_S{}^2$	No	No	Yes
I_F	Yes	No	No
IA	No	Yes	No

Table 4-1 Attributes of Composite Object Vs. Version Object

5.0 Version Management

In systems that support version management, a feature that provides "automatic" generation of a new version on the modification of version-significant attributes would be quite convenient. However, in design applications, modifications are rarely atomic; normally, they are batched updates. For instance, if the component aggregation properties are modified, this can require modifications in the interconnection and correspondence attributes.

5.1 Version States

In design databases, a nonatomic update (an update where more than one property is simultaneously updated) is a useful facility. In order to provide nonatomic updates and to maintain the constraints of version-significant and nonversion-significant properties, we stipulate that a versioned object be in any of the following three states: *validated*, *stable* and *transient*. The characteristics of these three version states are described below

Validated versions have the following characteristics:

1. No modification is allowed on validated versions.
2. All the constituent references in a validated version must be bound either to unversioned or other validated versions.
3. New versions can be derived from them.

Stable versions have the following characteristics:

1. Their version-significant attributes cannot be modified subject to the exception of equivalence modification.
2. Nonversion-significant attributes can be modified.
3. New versions can be derived from stable versions.

Transient versions have the following characteristics:

1. All newly created and derived versions begin in the transient state.
2. All noninvariant attributes of a transient version can be modified.
3. New versions cannot be derived from them.

The following table summarizes the updatability of versions in the three states.

	Invariant	Version siginificant	Nonversion significant
Transient	No	Yes	Yes
Stable	No	No *	Yes
Validated	No	No	No

* Subject to the exception of equivalence modification

Table 4-2 Updatability of Versioned Object Attributes

5.2 Version Operations

Since in this model we do not allow automatic generation of versions, modifications in the version significant attributes of an object can be achieved by deriving a new version, which is a copy of the given version object, in transient state and performing modifications on this version. Four operations and their detailed protocols are discussed below.

Promote is an operation defined on transient or stable versioned objects. When it is applied to transient or stable version, its state changes to stable or validated respectively.

Create is an operation defined on a generic object for creating a new transient versioned object from the generic object. It entails the following steps:

1. This operation requires a mandatory specification of a user-defined type along with the identifier of the generic object.
2. An object of the given type as well as of type *Versioned* is created, and it is assigned a version number.
3. The versioned object starts in the transient state.
4. It is made to value-acquire the invariant attributes from the generic object. No values are assigned to its other attributes.

Derive, an operation defined on stable or validated versions, creates a copy of the operand version. It entails the following steps:

1. A new object of the same type as the given versioned object is created and a version number is assigned to it.
2. The derived version starts in the transient state.
3. It is made to value-acquire all the invariant attributes from the generic object of the given version. All other attribute values, with the exception of the version specific functions defined on type *Versioned*, are copied from the given version.

Convert is an operation that converts an existing unversioned object into a generic object and creates its first version. It entails the following steps:

1. The type *Generic* is added to this object.
2. A new object as its first version is created with the same user-defined type.
3. It is made to inherit all invariant attributes from the generic object. All other attributes of the generic object are copied to this versioned object.
4. The internal assembly properties of the generic object, if any, are deleted.

In Figure 5-1, the transition diagram shows the effect of applying these operations on unversioned, generic and versioned objects. The dashed arrows signify that the object itself undergoes change, whereas the solid arrows mean that the object remains unchanged, and that these operations give rise to a new object.

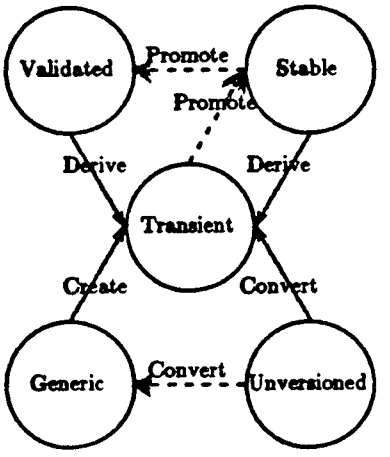

Figure 5-1 Transition of Version States

5.3 Prevention of Version Proliferation

Since versioned objects may themselves contain other versioned objects as constituents, an uncontrolled propagation of updates can proliferate to the top of composite hierarchy. The composite aggregation is defined to be version-significant, and thus its modification must cause mutation. However, version equivalence and interchangeability provide a capability for exploring and experimenting with different versions of a constituent object without having to create a new version of its composite object. The designer, however, has the option of deriving a new version and incorporating the desired changes there.

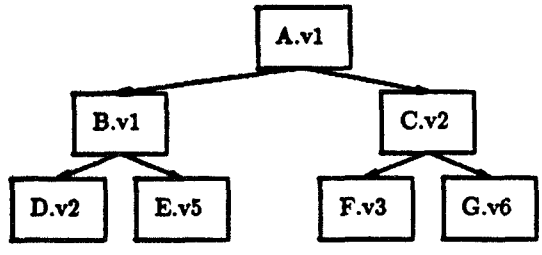

Figure 5-2

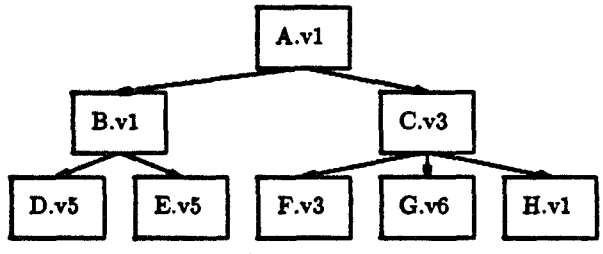

Figure 5-3

An example should illustrate the point. In Figure 5-2, a versioned object A.v1 (read as version v1 of object instance A) has two constituent objects B.v1 and C.v2; B.v1, in turn, contains D.v2 and E.v5 and C.v2 contains F.v3 and G.v6. Suppose a new version D.v5 is derived from D.v2. (Its version graph is shown in Figure 4-1.) B.v1 changes its reference from D.v2 to D.v5, which is shown in Figure 5-3. Since B.v1 has undergone equivalence modification, no new version of B.v1 need be created. However, it is possible to derive a new version B.v2 (not shown in the figures) from B.v1 and to replace D.v2 by D.v5 in B.v2. Again, either B.v2 can simply replace B.v1 in A.v1's assembly or a new version, A.v2, can be derived from A.v1 and the change can be incorporated in A.v2. Thus, the designer has the option, at every level of composition hierarchy, to choose from one of the two strategies.

Let us consider another scenario. The internal assembly of C.v2 need to be changed by adding a new object H.v1 to its composite aggregation and updating its interconnections and correspondence. (We assume that it has no effect on the interface of C.v2). This cannot be done without mutating C.v2. Thus, a new version C.v3 is derived from C.v2 and necessary modification are made in the internal assembly of C.v3, as shown in Figure 5-3. Now there are two options: either C.v2 is replaced by C.v3 in the internal assembly of A.v1 or A.v1 is mutated by deriving another version. Figure 5-3 represents the situation where a replacement is made in the composition aggregation of A.v1 without causing any mutation.

6.0 Conclusion

In this paper, a concrete approach has been proposed for defining a unifying framework for the effective management of versions of composite objects. In our view, the major results and contributions of this work are the following.

Generalized object-oriented concepts were presented to deal with the modeling of composite objects and their version management. The proposed classification of composite objects into intrinsic interface and nonintrinsic internal assembly attributes facilitates the representation of their functionality and behavior.

The maintenance of invariant attributes is provided by the notions of class and value-acquisition. The mechanism for interrelating disparate types of objects as members of a version set was described. A methodology was proposed for dealing with versioning at the instance and template levels.

Versions were classified under different states based on their updatability. Operation were defined to provide transition between their different states. A protocol was provided for controlling update propagation and preventing version proliferation.

The scheme presented here is a part of a detailed work on the overall treatment of version representation, creation, storage, and management [Ahm90]. The proposed scheme may be validated with an object-oriented system and a real-life design environment.

7.0 References

[Ahm89] Ahmed, R., and Navathe, S. B., "Version Control and Management in Computer-Aided Design Databases", Technical Report CIS-TR-89-9, University of Florida, Gainesville, FL (1989).

[Bat84] Batory, D. S., and Buchmann, A. P., "Molecular Objects, Abstract Data Types, and Data Models - A Framework," *Proc. VLDB Conf.*, Singapore (1984).

[Bat85] Batory, D. S., and Kim, W.,"Modeling Concepts for VLSI CAD Objects," *ACM TODS*, vol. 10, no. 3, p. 322-346 (1985).

[Bee88] Beech, D. and Mahbod, B., "Generalized Version Control in an Object-Oriented Database," *IEEE Conf. on Data Engineering,*, Los Angeles, CA (1988).

[Cho86] Chou, H-T. and Kim, W., "A Unifying Framework for Version Control in a CAD Environment," *Proc. VLDB Conf.*, Kyoto, Japan (1986).

[Cor90] Cornelio, A., Navathe, S. B., and Doty, K. L., "Extending Object-Oriented Concepts to Support Engineering Applications", *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, CA (1990).

[Emo83] Emond, J. C., and Marechal, G., "A computer Aided-Design System Based On A Relational DBMS," *ACM SIGMOD/IEEE Engineering Design Applications*, Atlanta, GA (1983).

[Fis87] Fishman, D. H., "Iris: An Object-Oriented Database Management System," *ACM TOOIS*, vol. 5, no. 1, p. 48-69 (1987).

[Joh83] Johnson, H. R., and Schweitzer, J. E., "A DBMS facilities for Handling Structured Enginnering Entities," *ACM/IEEE Engineering Design Aplications*, Atlanta, GA (1983).

[Kat82] Katz, R. H., "A Database Approach for Managing VLSI Design Data," *Proc. of 19th Design Automation Conf.*, Las Vegas (1982).

[Kat86] Katz, R., Chang, E., and Bhateja, R., "Version Modeling Concepts for Computer-Aided Design Databases," *Proc. ACM SIGMOD Conf.*, Washington, D. C. (1986).

[Ket88] Ketabchi, M. A., and Berzins, V., "An Object-Oriented Semantic Data Model for CAD Applications," *Information Sciences*, vol. 46, no. 1, p. 109-139 (1988)

[Kim87] Kim, W., Chou, H-T., and Banerjee, J., "Operations and Implementation of Complex Objects," *Proc. of Data Engineering Conf.*, Los Angeles, CA (1987).

[Kim89] Kim, W., Bertino, E., and Garza, J. F., "Composite Object Support Revisited," *Proc. ACM-SIGMOD Conf.*, Portland, OR (1989).

[Kla86] Klahold, P. and Schlageter, G., "A General Model for Version Management in Databases,"*Proc. of VLDB Conf.*, Kyoto, Japan (1986).

[Lan86] Landis, G. S., "Design Evolution and History in an Object-Oriented CAD/CAM Database," *IEEE COMPCON*, San Francisco, CA (1986).

[Lor83] Lorie, R. A., and Plouffe, W., "Complex Objects and Their Use in Design Transactions," *ACM SIGMOD/IEEE Engineering Design Applications*, San Jose, CA (1983).

[McL83] McLeod, D. and Rao, K. V., "An Approach to Information Management for CAD/VLSI Applications," *ACM SIGMOD/IEEE, Engineering Design Applications*, San Jose, CA (1983).

[Nav88] Navathe, S. B., and Ahmed, R., "Temporal Aspects of Version Management," *IEEE Data Engineering Special Issue*, vol. 11, no. 4, p. 34-37 (1988).

[Nav90] Navathe, S. B., and Cornelio, A., "Modeling Engineering Data by Complex Structural Objects and Complex Functional Objects", *Proceedings of the International Conference on Extending Database Technology*, Venice, Italy, (1990).

[Sto83] Stonebraker, M., Rubenstein, B., and Guttman, A., "Application of Abstract Data Types and Abstract Indices to CAD Databases," *ACM SIGMOD/IEEE Engineering Design Applications*, San Jose, CA (1983).

[Su90] Su, S. Y. W., Krishnamurthy, V., and Lam, H., "An Object-Oriented Semantic Association Model (OSAM*)," *AI in Industrial and Manufacturing: Theoretical Issues and Applications*, S. Kumara, R. L. Kashyap, and A. L. Soyster (Eds.), American Institute of Industrial Engineers, New York, NY (1990).