TOWARDS A UNIFIED FRAMEWORK FOR VERSION MODELING[1]

Randy H. Katz

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT: Support for computer-aided design data has been of increasing interest to database system architects. In this survey, we concentrate on one aspect of such support, namely *version modeling*, i.e., how best to describe the structure of a complex design artifact as it evolves across its multiple representations and over time. An operational model is also needed to describe how artifact descriptions are created and modified. While there has been much work in proposing new models and mechanisms for supporting version concepts in a database, it is our purpose not merely to describe such proposals but to attempt to unify them. It is not our goal to propose yet another model, but rather to provide a common terminology and a common collection of mechanisms that should underlie all version models. The key remaining challenge is to construct a single framework, based on these mechanisms, which can be tailored for the needs of a given version environment.

KEY WORDS AND PHRASES: version models, version frameworks, computer-aided design databases, design data management

1. Introduction

The database research community has recently shown substantial interest in extending database technology to better support non-traditional non-commercial applications. These new applications, in areas as diverse as computer-aided design, office information systems, and artificial intelligence, do not fit well within the confines of the usual database models based on tables, set-oriented non-procedural query languages, and the transaction models based on serializability. For these applications, data requires more complex organizations, with a high degree of connectivity that leads more naturally to retrieval via navigation, and support for interactive, long duration access.

Computer-aided design (CAD) has emerged over the last decade as an applications area of enormous economic importance, and one with significant demands as a consumer of database services. In broad strokes, its requirements are not very different from its cousins *computer-aided software engineering* (CASE) and *computer-integrated manufacturing* (CIM), which have begun to enter the database literature more recently. While Very Large Scale Integrated (VLSI) circuit design is usually chosen as the applications domain, the database requirements are not significantly different whether the object being designed is a chip, a software system, or an integrated circuit fabrication process. Each application must manage vast quantities of complex, highly interrelated

1. Research supported under NSF Grant # MIP 8706002.

data that changes over time, and must be prepared to react to certain kinds of data changes (e.g., alerters, change notification, etc.).

In this paper, we shall concentrate on computer-aided design, as it is the domain best understood by the author. The computer-aided design environment is particularly interesting for a number of important reasons. First, it is representative of the kind of non-traditional application domains that are driving the extension of conventional database systems. Second, computer-aided design software represents a multi-billion dollar software industry, with intense motivation for improving performance and ease of use. Third, because of the iterative and tentative nature of design activities, the CAD environment demands unconventional solutions (at least in the sense of traditional database management systems) for concurrency control, consistency, and crash recovery. Fourth, the CAD environment is history oriented: current data is not overwritten with new data, but rather maintained as historical data. And finally, the data organization supported by the system must be able to handle multi-faceted, multi-representational, complex (i.e., hierarchical) data aggregates.

A data model for computer-aided design must consist of two components: a *version model*, describing primitives for organizing data across time, and a *design transaction model*, specifying how such data can undergo change in a consistent manner. In this paper, we will concentrate on the former, but will touch on some aspects of the latter, since the version model is closely tied to the manner in which design objects are permitted to evolve.

Version-oriented databases are related to, but distinct from, time-oriented databases. Versions represent a significant, semantically meaningful change. Thus, it is implicit in a version database that not every update need result in the construction of a new version. Similarly, it does not make sense to perform a time series analysis of versioned data: there is no meaningful correlation between the time of the change and the value of the change.

The first papers within the database literature to address database support for CAD appeared in the early 1980s (actually the CAD literature is full of attempts to use database technology long before this). In the intervening years, there have been many models for version management proposed, but comprehensive frameworks for understanding version semantics are still absent. To be a true version framework, it must be possible to tailor the semantics of the version model to the needs of the particular user's environment. One of the key goals of this paper is to review the existing models in an effort to produce the catalog of features every version model must support, as well as the mechanisms proposed to implement these features. Only once the appropriate "knobs" are made visible and available to the user, with mechanisms clearly distinguished from policies, will it be possible to achieve a true version framework.

Throughout this paper, we will concentrate on *modeling* and *operational* issues in version management. Version modeling deals with such data structuring issues as organizing individual versions into version histories (i.e., which version is derived from

which), composing composite objects from versions of their components (i.e., configurations), and tracking equivalent versions across representations. Some of the operational issues include inheritance (i.e., a new version looks like its ancestors), change notification and propagation, and workspace space models with check-in/out mechanisms.

The rest of this paper is organized as follows. Section 2 defines some basic terms of relevance to version modeling, and describes the requirements that must be met by any model claiming to support computer-aided design data. Section 3 presents a detailed version model, developed by the author over the last few years, to highlight the basic modeling and operational issues. An in-depth survey of the evolution of version models is given in Section 4. We claim to be representative in our choice of models if not exhaustive. In particular, we will highlight recent developments in the commercial arena, such as the version models supported by Sun Microsystems' Network Software Environment and Apollo's Distributed Software Engineering Environment (both of which are targeted for software development, but actually have wider application). Section 5 extracts the basic mechanisms of version modeling from the systems reviewed in Section 4, and presents our proposal towards a unified framework. Our summary and conclusions are given in Section 6, followed by our references.

2. Definition of Terms and Basic Requirements

As stated in the introduction, the field of design databases is too new to have yet evolved a consistent terminology. In this section, we try to remedy this problem by defining some basic terms related to design management. In addition, we briefly review the requirements that any serious approach to version and design management must satisfy.

2.1. Terms

A *design system* is the collection of software for creating or synthesizing the design, analyzing it for design correctness, managing the storage and organization of the design data itself, and managing the process of design flow.

We make a distinction between *design applications*, i.e., the CAD programs that consume and produce design data, and CAD *design management*, i.e., the programs that structure the data and provide an operational interface through which design applications can access their data on demand. A *design database system* is that portion of the design management software that deals with (1) the storage of design data on disk, and (2) its consistent update. The design management software interprets how the data in the database represents a design, i.e., how the collection of design files is interrelated to form an organization for the design. In addition, the design manager provides certain operational supports, such as performing the more difficult task of interpreting design consistency rather than database consistency, that are beyond the capabilities of the database system.

Two key elements of design management are *workspaces* and *design objects*. Workspaces are named repositories of design objects, from which users and design applications can access them. They are also a unit of sharing among users of the design management system, as well as a point of access control. Design objects are aggregates of design primitives, such as layout geometries or lines of source code. They tend to be related to other design objects in rather complex ways. While a design object could be as primitive as a single geometry or a single line of code, it is better to think of them as units of the design that can be given a name, such as "Arithmetic Logic Unit" or "String Handler". Simply stated, a design object is a useful module of the design.

A *version* is a semantically meaningful snapshot of a design object at a point in time. As such, it is a descendent of some existing versions (if not the first version) and can serve as an ancestor of additional versions. While every change could result in a new version, this is usually not the case. New versions are created as part of the design process: changes are made, the changed object is verified, and if these changes are deemed acceptable, a new version is created or "released". Since some design objects are themselves an hierarchical collection of component objects, we define a *configuration* as a binding between a version of a composite object and the versions of its components.

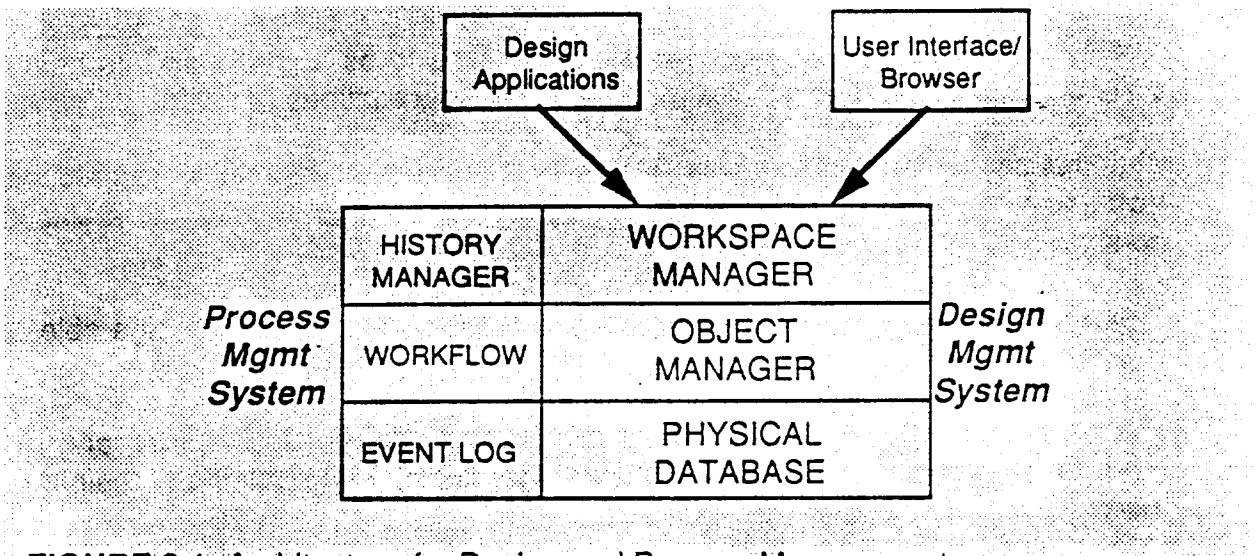

FIGURE 2.1: Architecture for Design and Process Management

The figure shows a leveled architecture for design and process management [KATZ 85]. *Design applications and the User Interface sit on top of the Workspace Manager, which controls access to objects* [GEDY 88]. *The Object Manager implements the semantics of design objects on top of the primitives supplied by the Physical Database component, which stores data on disk. In parallel with these components, the design process is controlled by a History Manager, which captures design events, a Workflow component, which interpretes these to determine what to do next, and a stored event log.*

Some of these concepts are illustrated in a sample system architecture for design management shown in Figure 2.1. The physical database component stores data on disk and ensures that the disk representation is updated atomically, i.e., is not left in a broken state in the event of a system crash in the middle of the update. It corresponds to what we have called the design database system. Implemented on top of this is an Object Manager that implements the highly interrelated network of design objects from the storage and access primitives provided by the physical database manager. Built on top of this is the Workspace Manager. It provides shared workareas to hold design objects. Special utility programs, such as a design browser, and the Workspace and Object Managers, represent aspects of design management not normally supported by a conventional database management system.

Figure 2.1 also shows some components for design process management, which we view as an independent subsystem distinct from design management. The History Manager tracks all accesses of data from design applications, and records these "events" in a log managed by the physical database component. The Workflow component reacts to these events, perhaps by scheduling the next logical event (e.g., tool invocation) in the design process flow.

2.2. Basic Requirements

Here we list the basic set of features every design data model should support. This "feature summary" can be further broken down into modeling requirements and design transaction requirements. A basic list follows.

(1) The design model should permit design primitives to be aggregated into named units called *design objects*. Design objects are aggregations of design primitives.

(2) The design model should allow design objects to be hierarchically composed from more primitive design objects. Design objects are composite, hierarchically constructed from components.

(3) The design model should associate *contents* and *interfaces* with design objects. Interface data is an important aspect of design representation.

(4) The design model should provide mechanisms to *represent*, *construct*, and *manipulate* configurations of design objects in terms of the interactions of their interfaces.

(5) The design model should provide mechanisms to specify and manipulate multi-version design objects and configurations of such design objects.

(6) The design model should include primitives to permit the selection of a particular version of a design object ("last created", "currently released", etc.) to participate in construction of a configuration, with a default mechanism.

(7) It should be possible to reuse previously defined cells, drawing a distinction between *instances* and *definitions*.

(8) The design model should incorporate a history mechanism to record design decisions for future review.

These general requirements can be made more specific, by separating them into the key requirements for representing the design data structure ("modeling") and for mediating the access and consistent update of design data ("design transaction"). The modeling requirements follow.

(1) Design data is organized hierarchically across all design representations.

(2) The model must support design evolution, i.e., the modeling of versions (also known as *alternatives*, *revisions*, *engineering change orders*) and configurations. Configurations should be able to be constructed both statically (all component references bound at creation time) and dynamically (references bound at the time of access). Dynamic configurations are also known as *parametrized versions*.

(3) The model must provide a way to identify equivalent or corresponding design objects across representations.

(4) The model should support object-oriented concepts, such as design objects as

instances of abstract data types and inheritance mechanisms.

Design transaction requirements describe how designers and design applications interact with design data. A list of these requirements follow:

(1) The design transaction model should support both loose and tight application coupling. In the former, the applications and the design management system work at arms length. The mechanisms provided are *check-in/check-out* and *workspaces*. In the latter, the design manager is part of the run-time environment of the application, providing it with a persistent object store.

(2) The design transactions should be conversational, i.e., a design transaction is a unit of concurrency (only one designer can access an object for update at a time), but not consistency or recovery. Thus a design transaction holds locks through crashes, but need not recover an in-progress object to its last consistent state.

(3) The design transaction model should provide mechanisms for supporting cooperative work and design teams. Such mechanisms as change notification and propagation, as well as support for workflow concepts address this requirement.

3. Detailed Presentation of a Version Model

3.1. Introduction and Motivation

In this section, we will present a detailed description of a particular version model, implemented in a Version Server for computer-aided design data [KATZ 86a; KATZ 86b; [KATZ 87]. We make no claim that the Version Server data model is the ultimate or most comprehensive, but it is representative of the approaches that have appeared in the literature. The model serves to highlight the requirements presented in the previous section and one set of mechanisms designed to meet those requirements.

The most important aspects of CAD data are their evolutionary, multirepresentational, and complex structure. The model has evolved to meet these representational demands of CAD data. Further, because of the way we update CAD data, the model supports workspaces and check-in/check-out operations for moving design objects among them. The greatest challenge is maintaining the correspondences across representations of the design, and we shall discuss mechanisms for enforcing such "equivalences" [BHAT 87].

In developing the model, we were guided by a single pragmatic consideration: that it be possible to implement the model on top of existing design files and environments to avoid rewriting any existing design applications.

3.2. Modeling Primitives

3.2.1. Philosophy

The underlying primitives of the Version Server data model are design objects and the relationships among them. In a sense, the model is like the popular Entity-Relationship model for commercial databases, except that certain relationships (introduced in the following subsections) are specially distinguished within the model. These include **is-a-kind-of** and **is-a-part-of** relationships, but others as well. We will often resort to figures to explain concepts within the model, and will use a nodes and arcs representation that naturally follows from objects and relationships.

A useful analogy can be drawn between a hierarchical file system and the space of design objects. Consider the former. The actual space of files is flat. Directories reference other directories and simple files, imposing hierarchical names on the flat space. Directories are merely files whose internal structure has special meaning for the operating system.

Now consider the space of design objects, which roughly correspond to files of design data produced and consumed by design applications. It is also flat. The purpose of the design model is to impose useful hierarchical organizations on this space. This is accomplished by introducing special organizational objects and relationships, in much

the same way that directories are added to a file space. We call these additional objects *structural objects* to distinguish them from the *representational cbjects* accessed by design tools.

It is a key aspect of our organization that structural information be separated from representational information. This is absolutely necessary if the same model is to be applicable to a wide range of design domains. *Structure* is clearly the responsibility of design management, while *representation* is determined by the design.tools.

Our model distinguishes three structural relationships upon which to organize hierarchical groupings of design objects. These are component hierarchies, version histories, and equivalences, and each is introduced in the following subsections.

3.2.2. Component Hierarchies/IS-A-PART-OF

Design objects are either *primitive* or *composite*, and are of a particular representation type. Associated with each design object are primitives of its type, such as layout geometries, logic schematics, or functional descriptions. A primitive design object resides at the leaves of a component hierarchy, while a composite object is the root of a subgraph. A particular representation of a complete design is represented by a component hierarchy of representation objects rooted at the object that describes the top-level of the design. Figure 3.1 shows a portion of a component hierarchy rooted at the datapath layout object.

The relationship between a component object and the composite object that "contains" or "uses" it has been called **is-a-part-of** in the artificial intelligence literature, and **aggregation** in the semantic data modeling literature. Note that although Figure 3.1 shows a tree structured composition, it is more general to permit composition hierarchies to be directed acyclic graphs (DAGs). Thus, a given component may be used by multiple composite objects simultaneously, yielding a DAG structure. Also note that although the figure shows a type homogeneous composition (all objects are of type layout), it is possible to mix objects of different types within the same hierarchy if desired.

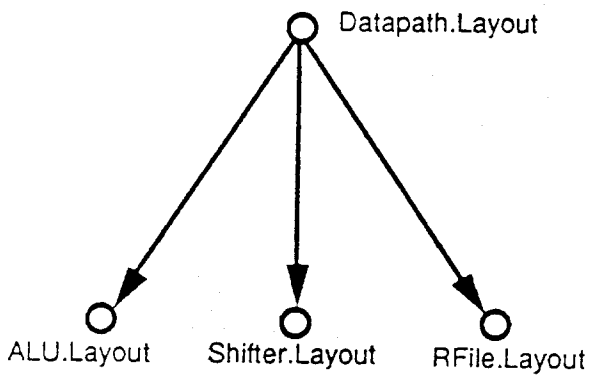

FIGURE 3.1: Component Hierarchies
Datapath.Layout is a composite object. Rfile.Layout ***IS-A-PART-OF*** *Datapath.Layout. This kind of interobject relationship is often called aggregation. Note that the component hierarchy can be a general directed acyclic graph, and is not constrained to be a tree as shown here.*

3.2.3. Version Histories/IS-A-KIND-OF; IS-DERIVED-FROM

We now incorporate the time dimension into the representation of design objects, by adding a version number to the name and type associated with each object (we denote objects by the triple *name[version #].type*). Yet this is not sufficient to capture all of the interesting semantics of versions. In particular, it is not possible to determine from this naming scheme which version is descended from which. Thus, we introduce the concept of a *version history* and **is-derived-from** as a new relationship distinguished by the model. This is shown in Figure 3.2.

In the figure, ALU[2].Layout **is-derived-from** ALU[0].Layout, where the latter is the ancestor of the former. We also say that ALU[2] is a *derivative* of ALU[0] since they are on the same path to the root, and that ALU[1], ALU[2], and ALU[3] are *alternatives* because they are NOT on the same path to the root (i.e., they are parallel versions). Note that in the figure, the version history is a tree, that is, every version has a single ancestor. From a data structure viewpoint, there is no reason why this cannot be generalized to a directed acyclic graph, although one with a single root. However, there are operational reasons why this may be difficult to support in practice, and we shall describe these in Section 3.3.

As can be seen in the figure, we have introduced a new object called *ALU.Layout*. Each of the objects ALU[i].Layout corresponds to some existing design file, but there is no such correspondence for the "generic" ALU Layout. This is our first example of a *structural object* introduced by the model to organize the underlying design objects. Each ALU[i].Layout **is-a-kind-of** ALU.Layout in much the same way as instances are related to types in object-oriented languages such as Smalltalk. Other researchers have proposed that versions be viewed as instances of a type (i.e., the type-version generalization of Batory and Kim [BATO 85]).

3.2.4. Configurations

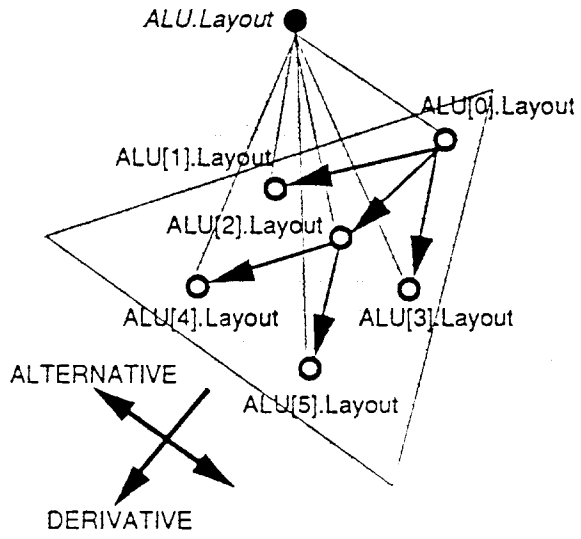

FIGURE 3.2: Version History

Version histories explicitly record the ancestor/descendent interrelationships among versions through distinguished ***is-derived-from*** *relationships. Alternatives are parallel versions while derivatives are versions on the same path from the root. Note that ALU[i].layout corresponds to some existing design file, but there is no such corresponding file for the generic ALU.Layout object. This is an example of a structural object added by the model to the space of design objects to assist in organizing them.*

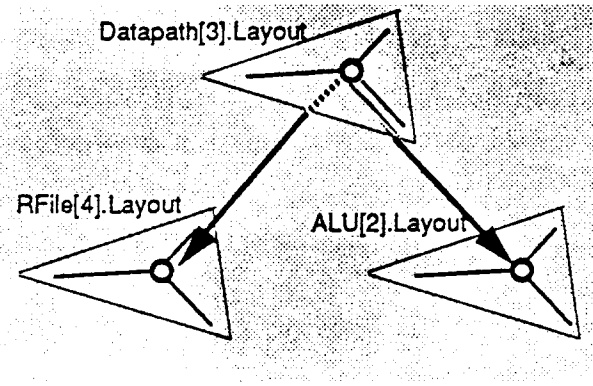

FIGURE 3.3: Configurations
The model supports a concept of distributed configurations. A version of a composite object is composed from specific versions of its components. Thus, Datapath Layout Version 3 is configured from Versions 4 and 2 of the Register File and ALU respectively. If the Register File and the ALU are not primitive, then they in turn are configured from versions of their components.

When the concepts of version history and component hierarchies are combined, the result is a *configuration*. This follows from the observation that a version of a composite object is composed from specific versions of its component objects. This is shown in Figure 3.3.

It should be noted that the model supports a kind of "hierarchical" configuration concept. Rather than associate a mapping between generic objects and specific versions of those objects for an entire component hierarchy, we associate such mapping information at each internal node of the hierarchy. In other words, a composite object version carries with it the version numbers of its components. This leads to the important concept of *static* versus *dynamic configurations*, which will be explored in greater depth in Section 3.3.2. There we will describe a mechanism that allows the binding between component generic object and one of its specific versions to be deferred until the **is-a-part-of** relationships is actually traversed.

3.2.5. Equivalences

Design data has often been called "multifaceted" because a variety of representations are needed in order to describe a design artifact. Each one of these facets or views (e.g., a layout view, a transistor view, a functional view) is a design object in its own right, yet the model should provide some primitive to tie these independent objects together. In the Version Server Data Model, this is accomplished through *equivalences*, another kind of structural object added to the design space to impose organization on

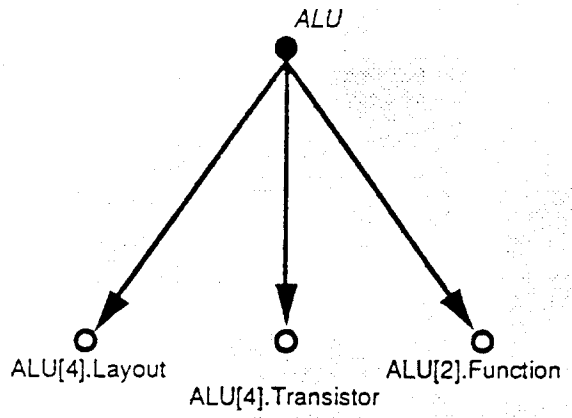

FIGURE 3.4: Equivalences
The structural/equivalence object ALU ties together the independent representations of the ALU, i.e., its layout, transistor, and functional facets. The arcs represent **is-equivalent-to** *relationships. Although we have shown them as having similar names, it is not a requirement that every object participating in the equivalence relation need have the same name or the same version number. Equivalences are also constraints on the database that can be exploited at design verification time.*

design objects. This is illustrated in Figure 3.4.

Equivalence objects stand in for design objects that are independent of representation type. Contrast the structural/version object *ALU.Layout* with the structural/equivalence object *ALU*. The former represents the version history and is type homogeneous, while the latter spans heterogeneous types.

Note that equivalence objects define equivalence relations on the space of design objects: they collect together design objects across types that describe the same real-world thing. It is necessary to initially make equivalences known to the system. As the design database evolves, equivalences can be inherited by new versions and later enforced during design verification. This will be discussed in Section 3.3.6.

3.3. Operations

Of course, every design object must be read or written. However, in this subsection we focus on those operations of design management that go beyond what one might expect to find in a conventional database system. These include operations for: identifying the current version within a version history, describing dynamic configurations, managing the movement of objects among workspaces, supporting change propagation, and inheriting attributes from related design objects.

3.3.1. Currency

It is frequently the case that the default or *current* version is not the most recently created version. Thus it is useful to separate "current" from "last", and to make currency update explicit. The currency concept can also be used to limit the branchiness of the version history. In the Version Server data model, we restrict new versions to be (transitive) descendants of the current version. That is, it is not possible to create a new derivative of an alternative of the current version. This is illustrated in Figure 3.5.

Operations are provided to explicitly position currency anywhere within the version history, although not every user should have access to such an operation. Multiple currencies can be made available, and references to current can be resolved with respect

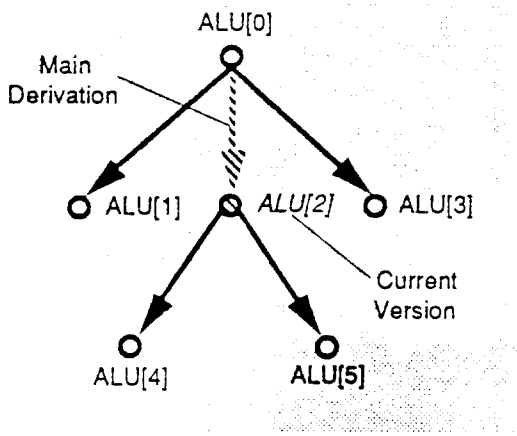

FIGURE 3.5: Currency Mechanism
ALU[2] is identified as the current version even though ALU[4] and ALU[5] were derived from it. The main derivation is the path from the root of the version history to the current version. Note that under the rules of currency, no new derivative of ALU[0], ALU[1], or ALU[3] can be made unless the currency is reset. However, new derivatives of ALU[4] and ALU[5] (and ALU[2] of course) can be added to the version history without changing currency.

to a chosen currency setting. For example, ALU[0] may be current in the "May Release" while ALU[2] is current in the "June Release". Obviously, the advantage of currency to constrain the proliferation of versions will be squandered if too many parallel currencies can be created.

Currency is our first example of an operational *mechanism* supported by the model. Associated with it is a policy that determines how the mechanism should be applied within a particular project context. Examples of policy specifications include: (1) which users are permitted to reset currency, (2) how many currencies may be simultaneously active within a single version history, (3) whether currencies can move backwards to earlier versions or must they always advance towards later versions.

3.3.2. Dynamic Configurations

It is useful to be able to create dynamic configurations, i.e., configurations in which references to components are not resolved until the **is-a-part-of** relationships are actually traversed. There are a variety of methods for implementing such a mechanism. We will describe the implementation used in the Version Server. An example is given in Figure 3.6.

The Version Server data model's mechanism is based on the layers and context approach used in the PIE System developed at Xerox PARC [GOLD 81]. The basic idea is to place portions of the version histories for multiple related objects into *layers*, and to provide *contexts* which order the layers. If two versions of different objects must be used together, then they are placed in the same layer. The context defines the search order over the layers. For example, if the current context defines the layer search order to be L3/L2/L1/L0, then a reference to ALU will resolve to version 2 and a reference to RFile will resolve to version 2. If the layers are shuffled so that L0 dominates L3, then the resolution will be to version 0 of each object.

A weakness of this method is that it is possible to combine the layers in ways that do not make sense. There are no explicit constraints that state, for example, which versions of the ALU can "interface with" which versions of the Register File. It may not be possible to combine ALU version 2 with the the first version of the Register File, although a context with layer 2 dominating layer 1 would lead to this resolution.

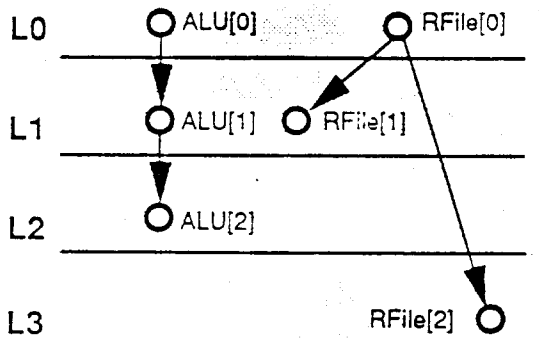

FIGURE 3.6: Dynamic Configurations
The version histories of related objects are layered and "compatible" versions of objects are placed within the same layer. Thus, ALU[0] and RFile[0] are compatible, as are ALU[1] and RFile[1]. A context is a particular ordering of layers. Object resolution proceeds by searching for a version of a named object through the layers in the order defined by the current context. In a way, contexts are like directory search paths in file systems.

3.3.3. Workspaces

Workspaces are named repositories for design objects. Workspaces can be *archive*, *group*, or *private*. Each kind of workspace is implemented in the same way, the only difference being who has the right to access the contents of workspace for the purposes of reading or writing. An archive workspace is readable by all and anyone may append to it, except that special protocols are enforced to ensure that only fully verified (i.e., "released") objects are placed into it. The Validation Subsystem is responsible for performing the necessary verification before permitting the archive to be updated (this is described in more detail in Bhateja and Katz [BHAT 87]). In a sense, an archive is meant to contain fully verified and finished design objects.

Private workspaces belong to specific designers, and only the owner of the workspace may read or append to its contents. The idea is that private workspaces can contain incomplete work in progress, so incomplete that other designers should be protected from looking at them.

It turns out that two kinds of workspaces are not sufficient for the CAD environment. Sometimes it is necessary to combine the in-progress work of two or more designers before it can be determined that the assembly works as required. Group workspaces are meant to support this kind of activity: any member of a specific group may access the contents of a group workspace or append to it.

Figure 3.7 shows a typical way in which a new version might evolve as it moves among the different kinds of workspaces. The object is checked-out from the archive into a private workspace, where changes are made. This is then made available to other designers in the group by checking the new version out of the private workspace and into the appropriate group workspace. Finally, after the required verification suite has been successfully run against the object, it can be checked back into the archive as a new version, with all the necessary version relationships automatically established.

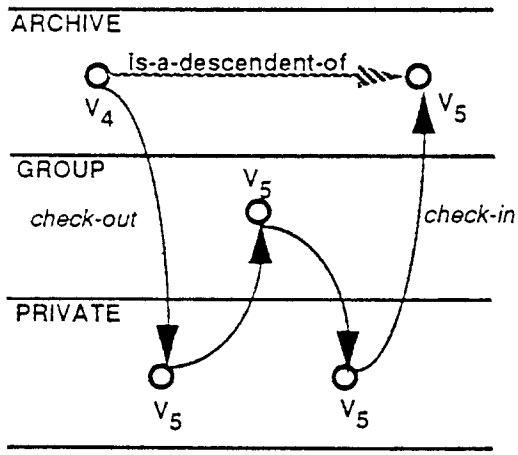

FIGURE 3.7: Workspace Model

The figure shows the evolution of V[4] as it moves through the different kinds of workspaces. A designer checks-out V[4] into his private workspace as V[5], where he proceeds to make some changes. The version is then checked back to a group workspace for integration testing with other design objects. Perhaps bugs are uncovered, and the version has to be re-checked into the private workspace to correct the errors. Finally, the new version is verified and returned to the archive for public use. At this point, its version history relationships are established.

3.3.4. Logical vs. Physical Representation

To make the discussion of the next two subsections a bit easier to understand, it will be necessary to reveal some of the details of how the Version Server physically implements the design model presented above. The basic idea is to represent the web of interrelationships by linked list structures on disk. Representation objects are implemented by records that point to the UNIX files containing actual design primitives in the format expected by the design tools.

Figure 3.8 shows the contrast between the logical view (what the designer sees) of the design and its internal representation. Internally, design objects are represented by **con**figuration objects that contain references to design files, and pointers to associated configuration objects of descendants (via version links) and components (via component links). The structural objects for **ver**sion histories and **eq**uivalences are present, with pointers to the configuration objects of the versions or equivalent objects. The version database is actually the set of configuration, version, and equivalence nodes, as well as the version, component, and equivalence links. The design files are not managed by the Version Server's database software, although new file system files are created as a side effect of check-out. Consequently, the storage overhead for representing versions can be kept relatively small.

3.3.5. Change/Constraint Propagation

We define *change propagation* as the process that automatically incorporates new

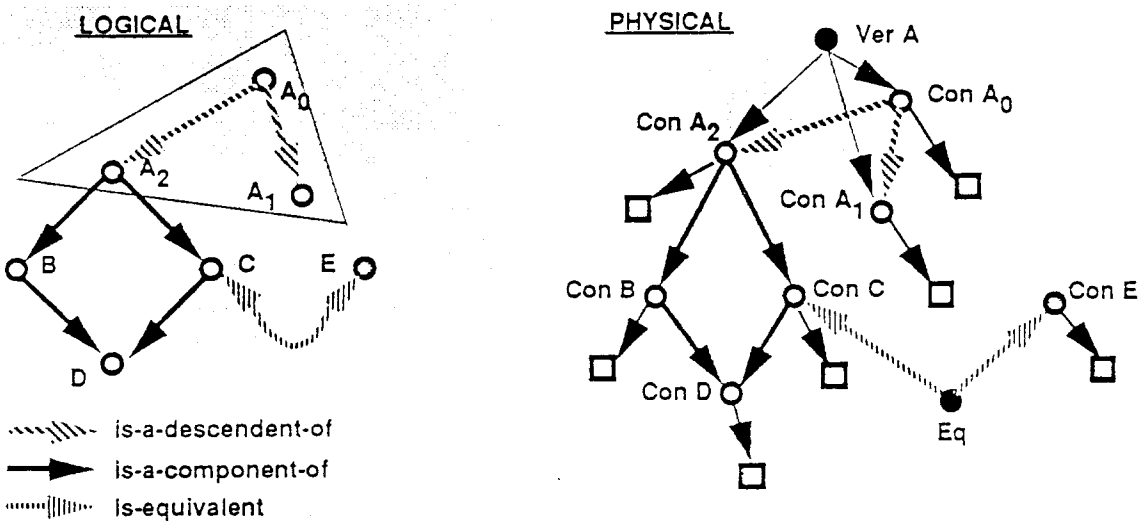

FIGURE 3.7: Implementation of Version Model Relationships

The logical view shows a three element version history, a component hiearchy rooted at A[2], and an equivalence between objects C and E. The physical view shows how these are implemented as linked structures on disk. The nodes represent records, the arcs pointers, and the squares UNIX files. A ***configuration*** *(CON) object points to: (1) the associated design file, (2) configuration objects of its descendents, and (3) configuration objects of its components. An* ***equivalence*** *(EQ) points to the configuration objects which are constrained to be equivalent. A* ***version*** *(VER) object points at all objects which are versions of the same generic object.*

versions into configurations. *Constraint propagation* refers to the enforcement of equivalence constraints by procedurally regenerating new versions [KATZ 88]. For constraint propagation to be supported, equivalences must be "directional", i.e., objects in one representation can be derived from objects in the other equivalent representation, such as between source and object code.

The two key issues in developing a mechanism for change propagation are to provide ways to (1) limit the scope of propagation -- rarely does the designer want to create a new configuration all the way up to the root of the component hierarchy, and (2) disambiguate the path of changes -- the resulting configuration can be quite different depending on the sequence in which the propagation takes place. This latter point is illustrated in Figure 3.8. While it is usually the case that a new version should supersede all uses of the version it replaces (case (ii) in Figure 3.8), there are situations in which it is useful to have more than one version of the same object in a given configuration. For example, a newer version of a register cell may be used in the area sensitive instruction cache of a microprocessor, while an older version is sufficient for the register file.

Figure 3.9 shows a sample sequence of steps followed in applying change and constraint propagation for an example configuration. In (a), the initial logical and physical configurations are given. Step (b) shows the result of executing a check-out followed by some update activity: a new version of B is created, pointing to a file with updated contents. The next step shows how the new version of B is embedded in a new configuration rooted at A. Even though A's contents have not changed, it now points to a new component, and is thus a new version in its own right. In step (e), the change to B triggers the invocation of a script to generate a new version of E. Note that the equiva-

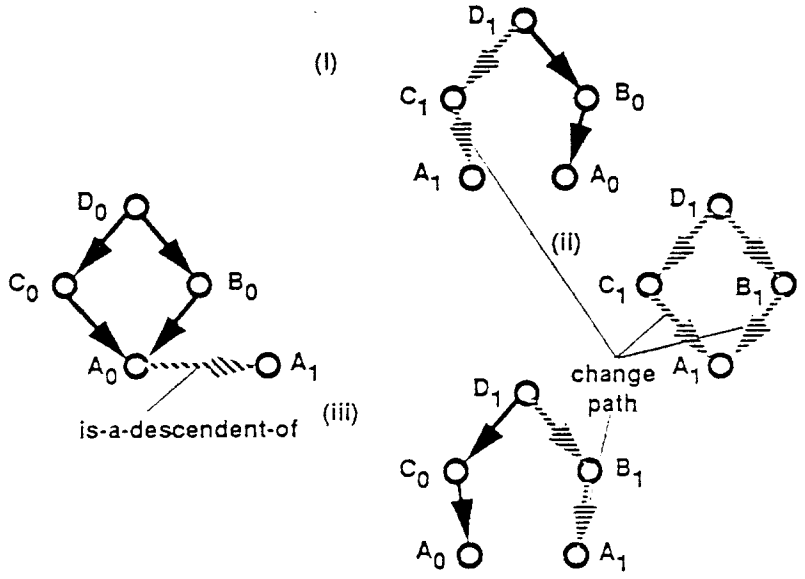

FIGURE 3.8: Ambiguous Change Propagation

The figure illustrates three possible outcomes for creating a new configuration as the result of embedding A[1] in the configuration at the left. Choice (i) shows the change being propagated along the path from A to C to D. Choice (iii) is its analog from A to B to D. Choice (ii) has propagated the change along both paths.

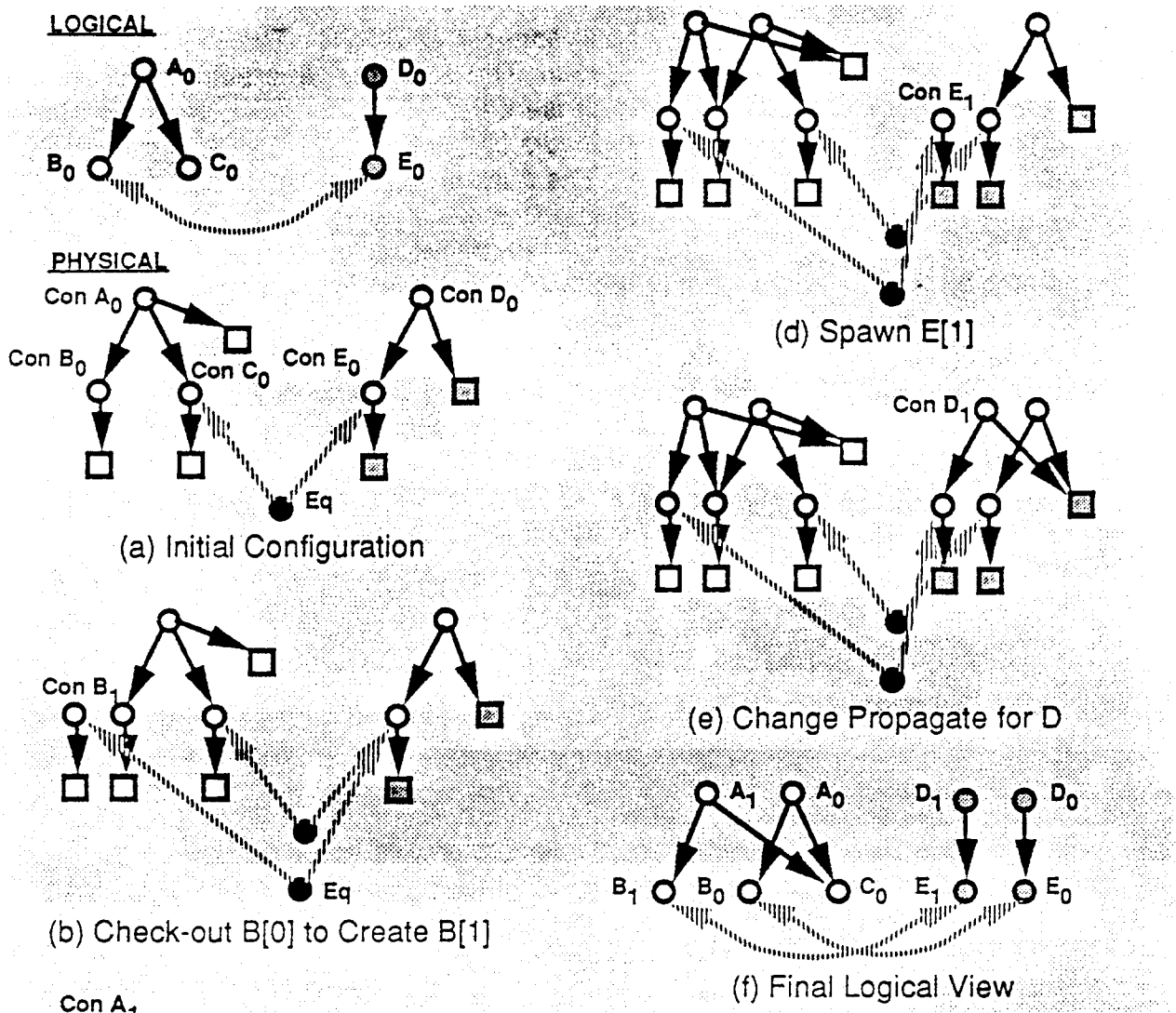

FIGURE 3.8: Change Propagation
The figure shows the steps from an initial configuration (a), as a new version of B is created (b), embedded in a new configuration rooted at A (c), spawning a new version of E through constraint propagation (d), and embedding this new version in a new configuration rooted at D (e). The final logical view appears in (f). Note that although D[1] and D[0] point to the same Unix file in (e), they have different components and thus are different versions in (f).

lence relationship is updated to point at the newly generated version. This is then embedded in a new configuration rooted at D, again shown as a new version because its components have changed, in step (e). The configuration as seen by the user is shown in (f).

There are several choices for how to deal with the ambiguity that is inherent in at-

tempting to propagate changes through DAG structured configurations. The first is to simply disallow all change propagation in non-tree situations. This is likely to lead to infrequent application of change propagation. The second is to create the cross-product of all resulting configurations, but to represent these as alternative versions of the root of the configuration. This proliferation of versions, many of which are likely to be of little interest to users, is also undesirable. The third is to propagate whenever the choices are unambiguous, but to abort propagation as soon as an ambiguous situation is encountered. The final approach, taken in our model, is to define operational mechanisms through which designers can make their intentions unambiguous.

The key mechanisms we use are *group check-in* and *group check-out.* The result of a group check-in is to guarantee the unambiguous creation of a single new configuration. Figure 3.10 shows the difference between checking in two new versions as a group versus checking them in independently. When an object is checked back in individually, the change paths are determined from the composition relationships associated with the object's ancestor (i.e., the object originally checked-out). These paths are followed creating new configuration nodes along their routes. A configuration node spawns a new version at most once no matter how many times a change path passes through that node. It is easy to show that the resulting configuration is independent of the check-in order and that only one final configuration is possible. Obviously, if multiple objects are checked back individually, a new resulting configuration will be created for each. On the other hand, when objects are checked back as a group, a single new configuration is created.

Group check-out clarifies which equivalences should be inherited at check-out time. Figure 3.11 shows where the ambiguity can arise. A checked-out object inherits the same equivalences as its immediate ancestor (unless overridden at check-out time). This can cause a problem if two equivalent objects are checked-out for simultaneous update with the intention of keeping them equivalent. The solution is to provide group check-out, to limit inherited equivalence to those objects within the group.

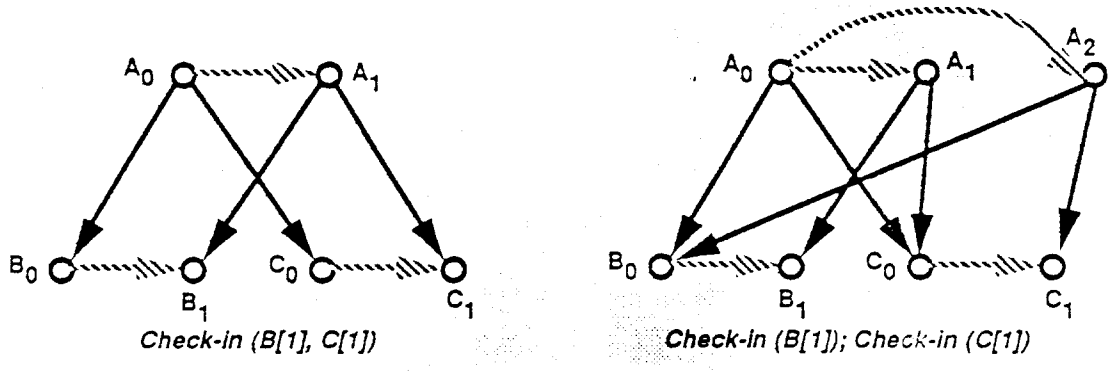

FIGURE 3.10: Group Check-in vs. Individual Check-in

In group check-ins (on the left), configuration nodes are updated only once, no matter how many times a change propagation path passes through them. Thus all new objects participate in a single resulting configuration. The right half of the figure shows what happens during individual check-ins: each check-in results in a new configuration.

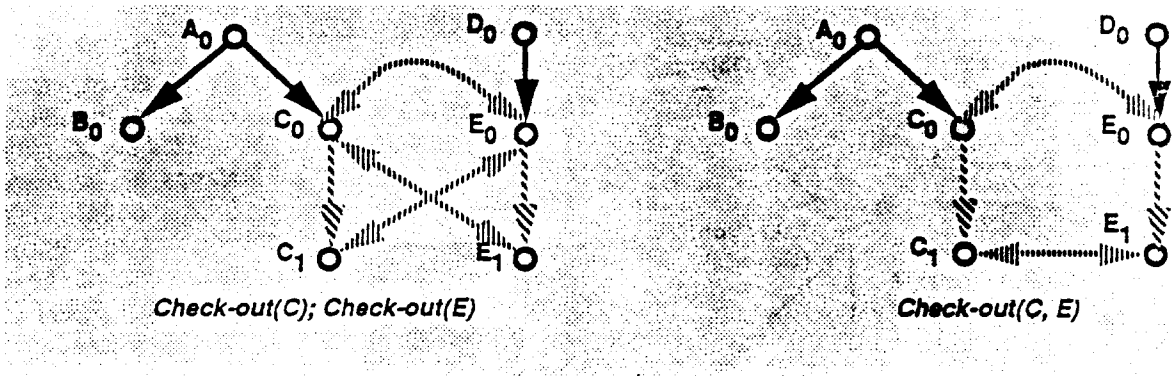

FIGURE 3.11: Group Check-Out

The figure illustrates the differences between individual (on the left) and group (on the right) check-out. When C[0] and E[0] are checked out individually to form C[1] and E[1], the latter inherit the equivalences of the former. When they are checked out together as a group, the new versions inherit equivalences to other new versions within the group, not to any ancestor objects of the group.

The second major issue in change propagation is limiting the scope of the change. There are a number of different mechanisms that can be employed, and we only touch on these briefly here. The first is to place constraints on configuration nodes so that change propagation will halt if creating a new version of the configuration would cause the constraint to be violated [RUMB 88] (see Section 4.2.11 for more details). Most of the approaches based on timestamps can be supported in this fashion. The mechanism can also support compatible interfaces, as well as proper containment and partitioning. A second possibility is to use a graphical presentation of the configuration [GEDY 88] and to hold a dialogue with the user to establish how far to propagate. This mechanism can also assist in disambiguating change propagation if the operational approaches described above prove difficult to use.

3.3.6. Inheritance

The concepts of object-oriented system are having a profound impact on all fields within computer science, and CAD databases are no exception. The most important aspects of the approach are its concepts of packaging operations with data (*abstract data types*) and inheritance. Abstract data types make it possible to apply certain operations to layout objects but not to transistor objects and vice versa. Inheritance is usually thought of as a mechanism to provide scoping for data and operation references. In addition, inheritance can be used to define how to propagate default operations and values to new versions. Some models have adopted inheritance as the primary mechanism through which to model versions. These are viewed as instances of a type, and inheriting certain aspects of that type, such as their interface description, at the time of their creation (see Figure 3.12). We have already seen the use of inheritance to describe how new versions participate in equivalence relationships/constraints.

Most models limit their support to *type-instance inheritance* , i.e., the only path of definition propagation is from types to instances. However, in a structural model such as ours there are other avenues for inheritance. For example, it is not particularly con-

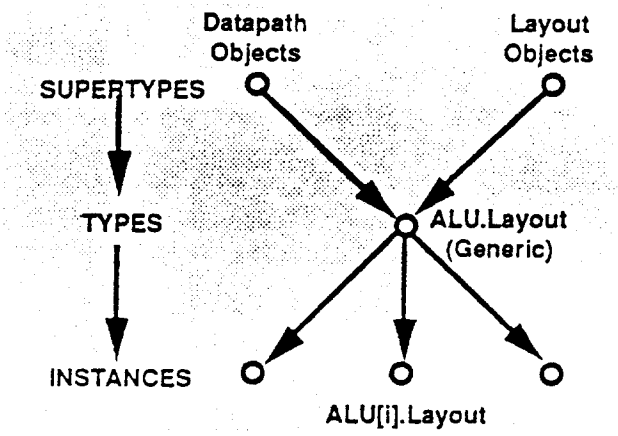

FIGURE 3.12: Inheritance

The figure illustrates inheritance among supertypes, types, and instances as it may be used in a version database. The ALU.Layout generic/structural object is an instance of the supertypes Datapath Object and Layout Object. It is a type in turn, with each of its "instances" versions of the ALU Layout. Any operation that can be applied to a datapath object or a layout object can be applied to a version of the ALU Layout. Further, the ALU Layout versions can inherit common attributes and operations from the ALU Layout, or override them on a version by version basis.

venient to model the equivalence inheritance of the version model solely in terms of inheritance from types to instances. See Figure 3.13. The desired semantics are that a new version inherits constraints from its ancestor in the version history, which is *another instance*. Since equivalence constraints are defined over specific instances, the only way to model this with conventional type-instance inheritance is by introducing new subtypes for each group of instances that share a common constraint. This proliferation of subtypes is obviously undesirable.

A better mechanism can be based on direct *instance-to-instance inheritance*, i.e., it is possible to inherit directly among instances, and there is no real difference between types and instances [KATZ 89]. This is most appropriate in a CAD database, since related objects are connected via structural relationships, and these are the most appropriate paths for inheritance. For example, a new version is highly likely to inherit most of its initial description from its ancestor in the version history. Alternatively, values may

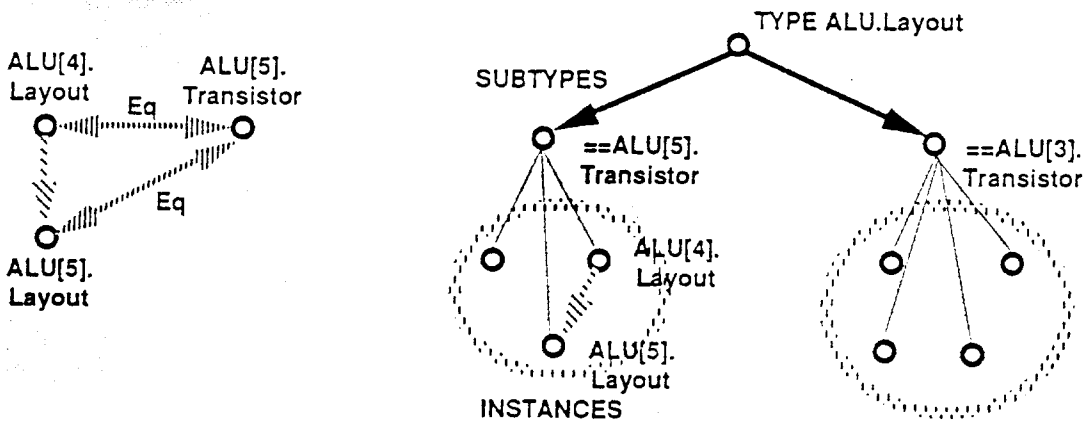

FIGURE 3.13: Problems with Type-Instance Inheritance

The figure illustrates the problem of support instance specific equivalence constraint inheritance. To do this with conventional type-instance inheritance requires that each group of instances that participate in the same constraint should be members of the same subtype. Then, when a new version is created, it can inherit the constraint from the subtype of its ancestor. Unfortunately, this proliferation of subtypes confuses the issue, and is not as desirable as supporting direct instance-to-instance inheritance.

be determined via context, with inheritance directed along configuration relationships. These may be such things as I/O port types, delays computed from loads, hints for placement and routing, and the change propagation constraints mentioned in the previous subsection. Finally, it may be useful to inherit along equivalence relationships as well, such as interface specifications from the functional description.

3.4. Summary

To summarize this section, we have presented a version model based on three orthogonal relationships for organizing the design space: version histories, configurations, and equivalences. We have also examined some of the operational aspects of the model, such as change propagation and instance-to-instance inheritance. In the former, the key issues were how to disambiguate changes and limiting propagation scope, and in the latter, it is clear that embedding versions in an object-oriented type system is a complex undertaking.

Now we turn to how the model described in this section meets the requirements of the previous section. The Version Server supports design objects, compositions, configurations, version histories, and equivalences. It also provides mechanisms to select the current version within a version history, and a primitive form of dynamic configurations through a layer/context mechanism. Where the model is weak is its lack of support for separate interface and contents portions of design objects, and an unclear distinction between instances and definitions.

4. Survey of Version Data Models

In this section, we review the various proposed version models and examine how they have evolved over the last eight years. In Section 5, we will put the different mechanisms into context by showing their similarities and differences across the proposed models.

4.1. General Trends

The beginning of the 1980's saw the emergence of the relational model as the data organization of choice for commercial data. Almost as soon as the model began to be more widely adopted by users outside the research community, attention became focused on the search for new applications of the relational approach. CAD data was one of the earliest to be examined, in part because of the rapid growth of the CAD industry during this time, as well as the growing interest in VLSI circuit design within the universities.

Over the last few years, three basic approaches have been pursued in developing new data models. The first is to extend the relational model, the second is to extend the entity-relationship model (the model of Section 3 falls into this class), and the third is to develop new models based on object-oriented concepts. We will examine each of these approaches in turn.

The relational model is based on tables, rows and columns of information, and powerful set-oriented query languages. However, it is difficult to model hierarchy with flat tables, and the concept of transaction, at least as implemented in most relational systems, is not well matched to the needs of the design environment. Thus, the extensions have focused on these issues. *Complex objects*, i.e., hierarchical collections of tuples, have been added to the relational model, and the transaction model has been extended with *conversational* and *nested* transactions. In addition, to overcome some implementation restrictions, the model has been extended with long fields, allowing tuple columns to contain data of almost unlimited length.

The entity-relationship model divides the world into objects of the real world, called *entities*, and their inter*relationships*. The way in which the E-R model is extended for CAD data is to select distinguished relationships to embed into the model, such as the **is-derived-from**, **is-a-part-of**, and **is-equivalent-to** relationships of the Version Server data model. In addition, the model must be extended in the operational direction, along the lines of Section 3.3.

Object-oriented models are the newest development. These fall into two broad classes: *operational models*, with support for packaging operations with data and mechanisms such as inheritance, and *structural models*, with distinguished inter-object relationships that can be used to model design objects as complex objects. The key difference in the object-oriented approach is to extend the type system to cover CAD data

through such mechanisms as inheritance. Obviously. models that combine operational and structural objects are the best suited for CAD data.

In tracing the evolution of version ideas over the last several years, we can see a growing awareness of the complexity of the issues and a proliferation of ever more complex mechanisms to deal with versions. Back in 1980-1981, many researchers thought that it was sufficient to model versions in a database with records or tuples extended with timestamps. 1982 saw the introduction of the important concept of the complex object, i.e., a way to represent hierarchical data within a relational database. This led to increased interest during 1983-84 on how best to structure CAD data, through such concepts as molecular objects and composite objects. During this time there was much controversy over the terminology associated with versions (which unfortunately, still exists to this day). Distinctions were made between versions as revisions and alternatives as alternate implementations. Further, the first mechanisms to support dynamic configurations were proposed. During 1985-86, the importance of derivation/version histories, configurations, and equivalence/correspondences were beginning to become recognized, while object-oriented type systems were beginning to influence the structure of models for design data. Change notification and propagation became key operational issues in 1986-87. In 1988, the field has moved towards extending the version concepts to model schema evolution (e.g., [BANE 87]), i.e., how the structure of the database changes over time, not just how its contents change. An increased awareness of the need for comprehensive version frameworks is also becoming widely recognized.

4.2. Proposed Models and Concepts

In this subsection, we will review key proposals from the literature, describing the version models and pointing out the basic concepts and what new ideas have emerged. Our presentation is largely chronological. A good way of tracking the evolution of the ideas is shown in Figure 4.1, which shows a "version graph" of the some of the key papers in the field, most of which we will discuss in the following subsections.

4.2.1. Haskin and Lorie [HASK 82]

One of the most fundamental papers to describe extensions to relational systems to handle CAD data was written by Haskin and Lorie of the IBM Research Laboratory at San Jose in 1982. This paper described a variety of extensions to the System-R Relational Database System to improve its ability to store and manipulate CAD data, including support for hierarchical aggregates of tuples across relations (*complex objects*) and mechanisms for long duration transactions, such as non-blocking locks and an on-disk lock table.

Despite ignoring the issue of versions, the system made an important contribution to the modeling of complex objects within the relational model. The schema definition language was extended to make it possible to declare explicit key attributes and to reference these from other relations as foreign keys. The SQL language was extended to make it possible to retrieve a heterogeneous, hierarchical collection of tuples based on

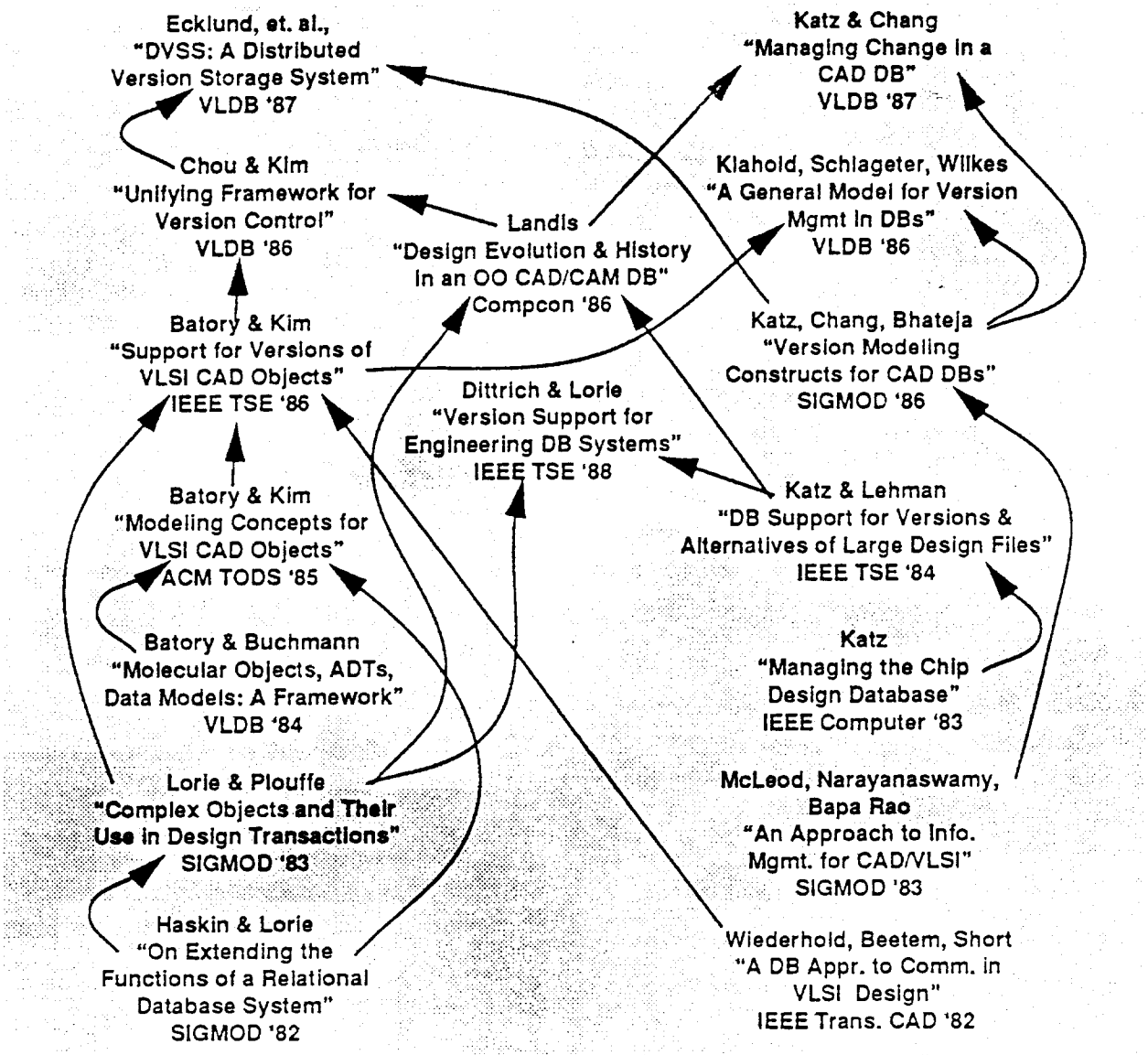

FIGURE 4.1: A "Version Graph" of Some Papers on Versions
Note that [WIED 82; KATZ 83; LORI 83; BATO-84; KATZ 84] *are early papers not mentioned in the text.*

key-foreign key matching given just the key value of the root tuple.

Figure 4.2 shows how inter-relation hierarchical relationships are specified through COMP-OF and REF attributes. An SQL query can be phrased to bring into memory the root tuple of a complex object (there is one complex object per tuple in MODULES), followed by a related PART tuple, FUNCTION tuple, and set of PINS tuples, next related FUNCTION tuple, and so on. So the query language can support a kind of hierarchical traversal of the underlying relationships. This works well as long as the relationships are tree-structured, and if they are not, the schema definer must identify the preferred retrieval path since only one attribute of a tuple can be COMP-OF. It is possible to define the interface so that an appropriately linked record structure is formed in memory as the result of a complex object retrieval.

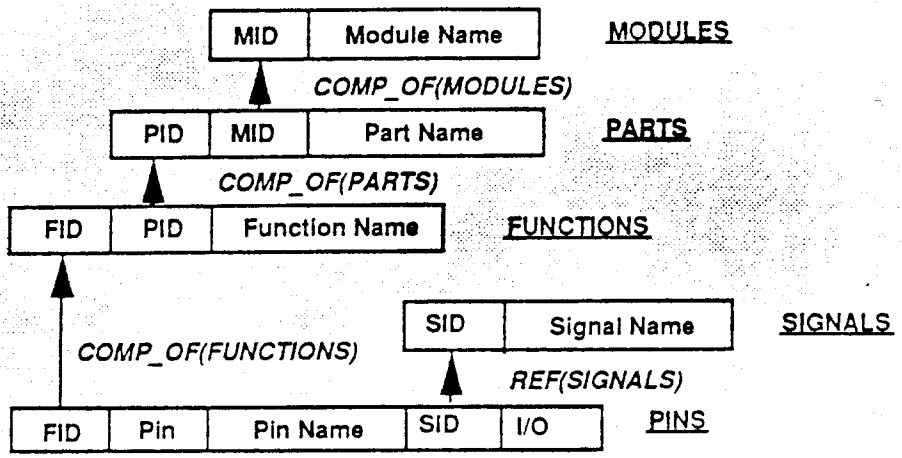

FIGURE 4.2: An Example CAD Complex Object

Attribute MID of the MODULE relation is declared as a key. Attribute MID of relation PARTS is declared as a foreign key of MODULES by using the notation COMP_OF(MODULES). Similarly for the attributes PID in PARTS and FUNCTIONS, and FID in FUNCTIONS and PINS. An SQL query can retrieve a set of tuples rooted at a particular MODULE tuple, with many PARTS tuples, each PARTS tuple with many FUNCTIONS tuples, each with a nested set of PIN tuples. If the relationships are not tree structured, as with PINS, FUNCTIONS, and SIGNALS, the tuples are retrieved but not in nested hierarchical order.

4.2.2. McLeod, Narayanaswamy, and Bapa Rao [MCLE 83]

In their 1983 paper, McLeod, Narayanaswamy, and Bapa Rao were among the first to realize the importance of data model support for versions and configurations, borrowing on some earlier work from the software engineering literature. Unlike Haskin and Lorie, who concentrated on extending the relational model, their work was developed within the framework of semantic data models. However, the paper does describe a procedure for realizing the semantic design schema as a collection of relational tables.

The model introduces the explicit concepts of compositions and versions. Although called configuration in their paper, their notion of object composition is not the same as what we have been using so far. The model distinguishes among versions representing alternative implementations, revisions, and different representations. Their model makes an explicit distinction between instances and definitions, and associates an interface description with objects.

These notions are combined and visualized in terms of an AND-OR graph with alternating AND and OR nodes, in which AND nodes represent object compositions while OR nodes represent alternative versions (see Figure 4.3). Thus, a version is described in terms of a composition of its subcomponents, each of which has its own versions. To promote reusability, composite objects are actually represented as *instantiations*, sort of a template with slots for instance specific attributes and a reference to the object's actual definition. In addition, versions are viewed as instances of the same type, sharing common attributes with their siblings. Note, however, that the model does not explicitly support relationships among versions, such as **is-derived-from**.

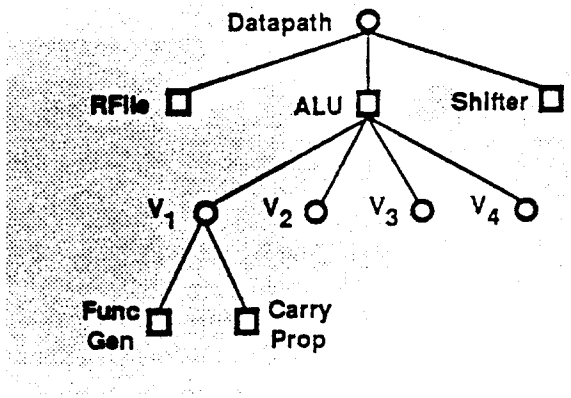

FIGURE 4.3: AND-OR Graph
AND nodes are circles and OR nodes are squares. A particular Datapath version is configured from RFile, ALU, and Shifter components. The ALU has four versions in turn. Version 1 of the ALU has the components Function Generator and Carry Propagate, which in turn have versions. Each version can be viewed as an instance of a type, sharing attributes with other instances, such as a common interface.

4.2.3. Dittrich and Lorie [DITT 88]

Dittrich and Lorie present a model of versions based on the concepts of design objects, generic references, and logical version groupings. In their view, a design object is a set of versions with a single distinguished *current version*. They provide no explicit support for derivation information. Design objects can reference other design objects to form hierarchical aggregations. These references may be bound to a specific version of a design object (including such qualifiers as "last frozen version") or they may be *generic*, which is to say that they refer to a particular design object, but not to any of its specific versions. Generic references are one way to provide the *dynamic configurations* discussed in Section 3. Similar to the proposal in Section 3, generic references can be dereferenced through an environment mechanism. Environments bind design objects to specific versions, or may reference other environments which define the binding. Environments may be ordered so that a given environment's bindings may dominate the bindings of other environments. The basic concepts are illustrated in Figure 4.4.

In addition, their model supports the concept of logical version cluster, which allows the user to impose more structure on the design versions by aggregating them into arbitrary groups. For example, it is possible to impose on the space of versions a grouping structure that clusters together versions that are revisions of the same alternative. Thus, under the design object node are a group of nodes representing version clusters for individual alternatives. Under these are additional clusters representing revisions of each alternative. Finally, associated with each revision cluster are those versions that participated in the revision. Thus, arbitrary hierarchies of version clusters can be formed. Note that the clustering structure need not partition the versions; a version may appear in more than one cluster.

4.2.4. Klahold, Schlageter, and Wilkes [KLAH 86]

Klahold, Schlageter, and Wilkes proposed additional methods for imposing structure on the version space. Their model is founded on the concept of *Version Graph*, which is similar to the Version History of Section 3, but more general since derivation history is only one of many possible threadings through the versions to form a graph. In an effort to combine concepts of object consistency with version information, they layer *partitions* on top of the Version Graph. Partitions are groupings of versions based on their level of

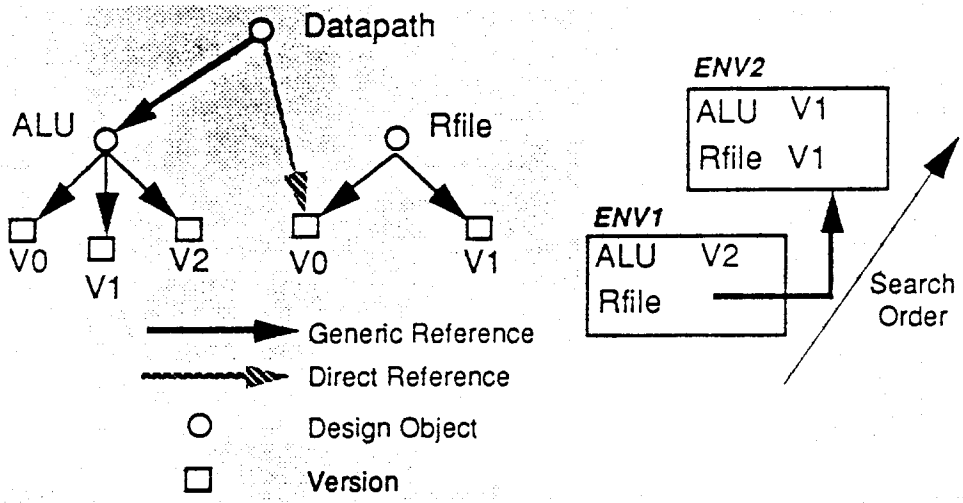

FIGURE 4.4: Dereference Procedure for Generic References

A datapath object is formed from Rfile version 0 and a generic reference to the ALU. The generic ALU reference is dereferenced with respect to Envronment 1, thus binding to version 2. Note a generic reference to Rfile would be redirected to Environment 2, where the binding would be to version 1. If Environment 2 is given priority over Environment 1, either generic reference would bind to version 1 of the respective design objects.

consistency, the idea being that older versions have passed more consistency checks than new versions. They also provide support for *subset views* of the Version Graph, which are subsets of the graph that satisfy specified criterion. Linkages among objects are inferred if intermediate nodes are left out of the subset. Figure 4.5 gives an example of a Version Graph and one of its subset views.

4.2.5. Batory and Kim [BATO 85]

Batory and Kim published an important paper addressed at how to model VLSI CAD objects and their versions. Their approach was an Entity-Relationship model extended

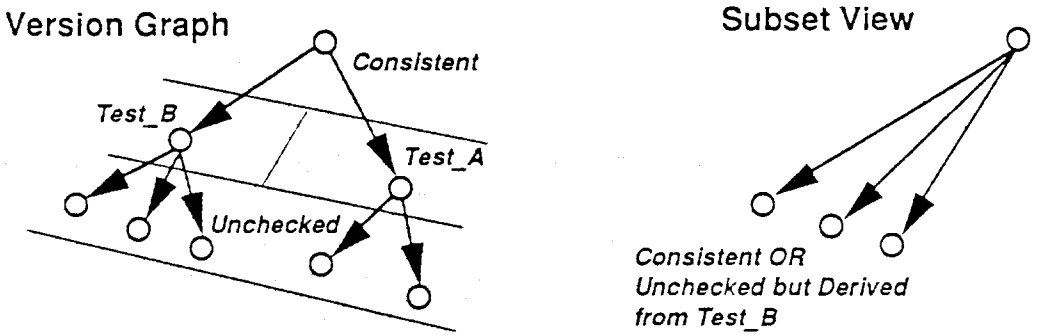

FIGURE 4.5: Version Graphs, Partitions, and Subset Views

On the left is a Version Graph partitioned into the four groups "consistent", "passed test_A", "passed test_B", and "unchecked". At the right is a subset view of this version history that shows all versions that are either consistent or are as yet unchecked but have been derived from some version which has passed test_B.

with object-oriented inheritance. The paper introduced four new modeling concepts: *molecular objects*, *type-version generalization*, *instantiation*, and *parametrized versions*. We will discuss each of these in turn.

Molecular objects are a way to aggregate more primitive objects and their relationships. For example, a 4-input NAND circuit could be aggregated from three interrelated 2-input NAND gates. The 2-input NANDs are primitive objects, while their aggregation into the 4-input NAND constitutes a molecular object. A key aspect of molecular objects which makes them attractive for modeling VLSI components is their explicit separation of *interface* from *implementation*. In essence, the molecular object model describes the construction of higher level component assemblies in terms of the interconnection of component interfaces.

If molecular objects form the component hierarchy, then type-version generalization provides the mechanism through which a form of version history can be supported. In the Batory and Kim model, interfaces are used as the specification of an object *type*, and versions are represented as *instances* of that type. In other words, all versions of an object are constrained to implement the same interface. Versions are viewed as alternative implementations or revisions to previous versions, with no explicit support for derivation history. One important advantage of using the type mechanism as a way to aggregate versions of the same object is that the type system's inheritance mechanisms can be leveraged to good advantage: properties of the type (i.e., the interface) can be inherited by the versions (i.e., the implementations).

The model's instantiation concept separates use of design data from its definition. Instances are distinct from versions; the former represents a use and is a copy of a version, whereas the version corresponds to a definition. Once again, property inheritance can be used to propagate attribute values from definitions (versions) to uses (instantiations).

The fourth mechanism they introduced was parametrized versions, basically a way to support dynamic configurations. A molecular object reference can be bound to a specific version of its component or it can reference the component's type, leaving the binding to a specific version unresolved. The semantics of parametrized versions is that any version of the component can be plugged into the molecular object.

Batory and Kim also address the operational issues of change notification in their design data model. They point out that there are many directions in which change information can spread: from one representation to another, from a component to its container, or from an ancestor to a descendent. Because of this potentially vast complexity, they propose a mechanism that uses timestamp information to limit the range of notification messages that must be sent.

The mechanism works as follows. Associated with each object are two timestamps: T_{CN}, the time when the object was last changed, and T_{CA}, the time when changes

were last acknowledged. An object is *implementation consistent* if its T_{CA} exceeds T_{CN}. In other words, its designer has acknowledged the changes that have been made to the object. An object is *reference consistent* if its T_{CA} exceeds the T_{CN} of all of its components. Thus the designer of a composite must acknowledge the changes of the components s/he has used. Finally, if object L is derived from object H, they are *representation consistent* if L's T_{CA} exceeds H's T_{CN}. L's designer must acknowledge the changes made to the object it depends on, namely H. Change notification is complete once all objects are implementation, reference, and representation consistent.

4.2.6. Landis [LAND 86]

Landis described an early version of Ontologic's data model for applications like computer-aided design. The model is based on four concepts: *non-linear version histories*, *version references*, *change propagation*, and *limiting of change scope*.

The model supports branching version histories, where each branch is meant to represent a new *alternative*. It has explicit support for identifying the *current version* and the *default branch*, in a manner very similar to the model of Section 3.

Version references are a mechanism to deal with dynamic configurations. Historical references are always bound, that is, once a version is superseded, its references to other versions cannot be altered. References from current versions which are not explicitly bound always reference the current version of the target. This is illustrated in Figure 4.6.

The model also provides some operational support for change propagation. New versions of related objects are created as a result of a change made to the version they reference. Under user specification, change propagation is invoked whenever (1) a new version is created, (2) the schema is changed (e.g., a new property is added to a version), or (3) a property value is changed.

As mentioned in Section 3, such a change propagation strategy could lead to a proliferation of new versions, many of which may not be of relevant interest to the design

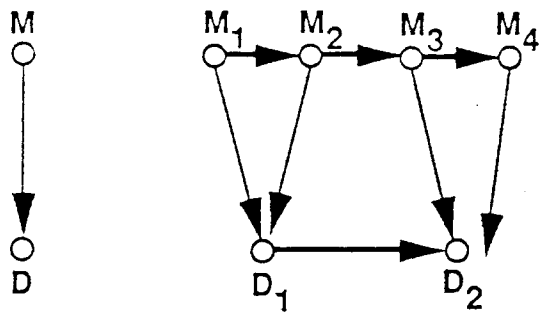

FIGURE 4.6: Version References
Object M references object D. The objects evolve as follows. D1 is current, and is referenced from M1 when it is created. M2 then supercedes M1, forcing M1's reference to be permanently bound to D1. Next, M3 supercedes M2, forcing M2 to be permanently bound to D1 as well. Now D2 supercedes D1, and M3 references it, since M3's reference always binds to the current D until it is superceded. This is shown as M3 is superceded by M4.

team, but merely artifacts of the change propagation process. To help control changes, the Ontologic model provides two additional mechanisms: *delta sets* and *pended version creation*. Delta sets are much like the log of changes maintained by a transaction-based system. These are sets of changes to related objects that should be considered the unit of undo or redo. Pended version creation is similar in spirit to the group check-in mechanism of Section 3.3.5: in propagating changes through the lattice of relationships, no change should cause an object to evolve through more than one new version.

4.2.7. Chou and Kim [CHOU 86]

Chou and Kim collaborated to produce a major revision of the earlier model proposed by Batory and Kim. Their new model borrowed much from the model discussed in Section 3, such as support for configurations, versions, derivations, and equivalences, and an operational model based on workspaces with check-in/check-out operations and dynamic configuration binding via a context mechanism. They also distinguished among three different types of versions: transient, working, and released.

Their new contribution was a further exploration of the change notification problem within the context of their data model. They distinguish between message-based notification mechanisms, which may be either immediate or deferred, and flag-based mechanisms. In the latter, the designer is made aware of a change that may affect his/her object only when it is next accessed. Because of the difficulty of successfully limiting the scope of changes, their approach to change propagation is to limit changes to a single level. That is, only those object which directly reference the changed object undergo change themselves, and the process does not recurse.

4.2.8. Ketabchi and Berzins [KETA 87]

Ketabchi and Berzins propose another variation on database models for versions based on their concept of *refinement*. They describe three different kinds of refinements: (1) *template refinement*, in which a design object is described in terms of its multiple "aspects", (2) *explosion refinement*, in which a version of a composite object is described in terms of versions of its components, and (3) *instance refinements*, which describe the revisions and alternatives of a design object. These three concepts are variations on the notions of equivalences, configurations, and version histories introduced in Section 3.

An interesting aspect of their model is their support for incremental, independent, and alternative refinements in describing the evolution of a design object. Version histories are actually represented by two related graphs. The *refinement graph* (RG) describes which versions are derived from which earlier versions. This is exactly like a version history graph, except that it can be a DAG rather than a tree. Associated with this is the *incremental refinement graph* (IRG), which records the changes or *deltas* that differentiate the new version from its ancestor. These may represent *dependent refinements*, in which changes to the ancestor directly affect the descendent, *independent refinements*, in which deltas on parallel paths do not change the same data, and *alterna-*

tive refinements, in which changes on parallel paths do affect the same information. The three different kinds of incremental refinement are shown in Figure 4.7.

The authors also describe a check-in/check-out model based on *Parts*, *Projects*, and *Private* databases. The Parts database is a global, essentially read-only repository of design components common to multiple design projects. Project databases contain the refinements, alternatives, and versions of a given project. The addition of new objects to these kinds of databases requires a careful validation before update. Private database are the purview of the individual designer, and have few restrictions. Note the general similarity to the Archive, Group, and Private databases of Section 3.

4.2.9. Eckland, et. al. [ECKL 87]

Eckland, et. al. describe a system which generalizes the simple server/client models presented so far. Their system, called DVSS (for Distributed Version Storage Server), views access to design objects in terms of a federation scheme, in which users have access to objects which are allocated to sites. A user who is an associate member of a federation has read-only access to its assigned objects, while participating members have read/write access. A federation is similar to the public workspace in the terminology we have already presented. Federations correspond to major repositories of design data, such as specific part libraries or project design data.

Design objects are represented by version sets organized not as trees but as rooted DAGs. A distinguished path through the DAG is called the *principal path.* The current version of the object is defined to be the most recently created version on the principal path.

The DVSS model distinguishes among three different ancestor/descendent relationships within the version DAG. A new version created along the principal path is called

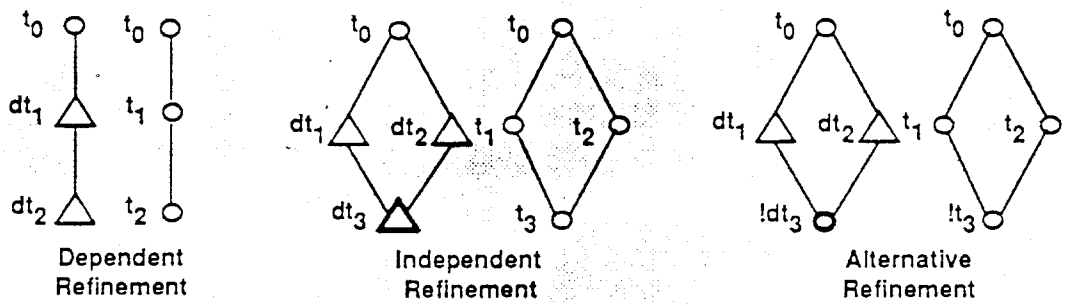

FIGURE 4.7: Dependent, Independent, and Alternative Refinements

Each of the three portions of the figure shows an Incremental Refinement Graph of deltas on the left and a Refinement Graph of versions on the right. For dependent refinement, the versions form a path. T2 is derived from T0 by applying both deltas in sequence. For independent refinement, the versions form a tree with each associated delta changing a different portion of T0. In this case it is possible to combine the changes on multiple paths to create a new version, as is shown for T3. Delta T3 is the combination of Deltas T1 and T2. Alternative refinement also produces a tree, but the associated deltas overlap. An attempt to combine them to form a new version is flagged as an error, as is indicated in !T3.

the *principal successor*, and there is at most one such successor. Its relationship with its ancestor is called *refinement*. If multiple users are making changes to the same version at the same time, the first to finish and check-back the changes as a new version creates the principal successor. Other users create in-parallel descendants, whose relationship with their common ancestor is called *derivation*. When versions along multiple paths are reconciled to form a new common path, the relationships between the newly created version and its multiple ancestors are called *consolidation*.

References among objects can be *fixed* to a specific version or left *floating*. Floating references can have an optional time associated with them. At check-out time, floating references are resolved to the current version (at the optionally specified time) along the specified version path. The version path used for evaluation can be altered via *assign* commands issued to the DVSS.

4.2.10. Beech and Mahbod [BEEC 88]

Beech and Mahbod describe a version model that has been embedded within the object-oriented type system of the IRIS database manager developed at Hewlett-Packard Research Laboratories. In their model, a version is an object in its own right, with its own unique *Object Identifier*. Version instances are organized into version sets associated with a single generic instance. Generic and version are built-in types, and can be mixed with conventional design types, such as layout and netlist. The version graph is realized via functions associated with the generic and version instances. These functions define the first, last, successor, predecessor versions, etc.

New versions can be automatically generated via two methods: *by mutation* and *by propagation*. In the former, a change to a prespecified "significant" property triggers a copy on write: the current version is frozen and a new version is created before the change is made. The second method is very similar to the change propagation of Section 3.3.5, i.e., the creation of a new version of a subcomponent triggers the creation of a new version of its containing object.

Interobject references may be either *specific* or *generic*. Specific references identify a particular related version while generic references identify a generic instance and its associated version set, but no specific version. A *context* mechanism is used to specify when and how generic references are to be resolved. It is a very general mechanism, based on database triggers, which can take place either before or after an operation invocation, and user specified rules, which declare how resolution is to take place.

4.2.11. Rumbaugh [RUMB 88]

Rumbaugh has proposed a simple mechanism for controlling operation propagation across relationships. It is based on associating *propagation attributes* for particular operations with the relationships. These attributes can take on one of four values: *none*, i.e., the particular operation does not propagate; *propagate*, i.e., the operation should be applied to both the relationship instance and the related object; *shallow*, i.e., the opera-

tion should be applied to the relationship instance but not the related object; and *inhibit*, which suppresses propagation as long as an instance of the relationship exists. The latter is not quite the same as *none*. It allows propagation to take place once the last instance of a relationship associated with a changed object is deleted, and is useful for implementing propagated destroys to objects that are no longer referenced. While the operations Rumbaugh had in mind are things like copy, destroy, print, and save, the mechanism has general application for controlling change propagation as discussed in Section 3.3.5.

4.2.12. Vines, Vines, and King [VINE 88]

These authors describe the version and change control model of GAIA, an object-oriented framework for an ADA programming environment being developed by Honeywell. Their approach for change control is based on four related concepts. First, versions are represented by timestamps rather than version numbers, to more clearly correlate events with the versions that they spawn.

Second, explicit relationships between objects are created to define the impacts of change. These relationships may be *version-sensitive*, in which an object is notified of a change to a related object, or *change-sensitive*, in which an object may not itself be sensitive to the change but may be related to another object that could be affected by the change. Change-sensitive relationships allow an object to redirect a change notification to another related object which is not directly related to the changed object.

Third, their model supports explicit objects for managing change. A *change request object* is created in response to a human or machine generated change request. It becomes associated with the object that is to be changed, tracks the change as it evolves, and provides the anchor for an audit trail. A *change notification object* is spawned when a change request is propagated along version or change sensitive relationships. These can spawn new change request objects if the target object must be changed in response to the change being made to the source object. While these change requests can be generated automatically, they are more likely to be created at the discretion of the responsible designer.

Fourth and finally, configuration objects are introduced to group changes together. A configuration is a tabulation of objects and their specific versions. New configuration objects can be generated in response to requests spawned from changes to component objects.

4.2.13. SUN NSE [SUN 88]

The SUN NSE system is a design environment for software development, but incorporates many of the concepts discussed above. It views the design environment in terms of objects that are to be manipulated by type-specific tools and type-generic commands. The kinds of objects known to the system include the following:

File which can be *source*, *derived*, or *generic*. Files can be versioned.

Target a package of a derived file, the objects it depends on, and its reconstruction recipe.

Component a building block for constructing hierarchical structures which can be type heterogeneous. Components are very much like file system directories, except that they are organized by time sensitive snapshots called *revisions*. In essence, components provide an hierarchical name space for objects.

Linkdb supports arbitrary connections among related objects. Linkdbs can be versioned.

Versions are stored as interleaved deltas within the object.

The access paradigm is based on the operations of *acquire*, *modify*, and *reconcile*. The effect of an acquire operation is to copy (i.e., check-out) an object from one "environment" (i.e., workspace) to another. Reconcile copies back (i.e., check-in) the object as a new revision. The system provides special type-dependent tools to assist in merging together parallel revisions.

The environments, which are essentially workspaces, are arranged hierarchically. While not limited to three levels, the documentation suggests organizing the environments into levels representing at least release, integration, and development activities.

The system provides some operational support for change notification. Developers register notification requests with the system. When a particular "interesting" event happens in an "interesting" environment, such as a recompiling a particular file, the system will send email to the requester.

An interesting aspect of NSE is its ability to be configured for a variety of different hardware architectures. The concept of *variant* allows the design team to specify the appropriate compilers, libraries, etc. that are to be used for a particular target architecture.

4.2.14. Apollo DSEE [LEBL 84]

The Apollo DSEE system is a comprehensive environment for software development. It consists of five main components, of which we will concentrate on the first two. These are: (1) the *History Manager*, which provides SCCS-like version control, (2) the *Configuration Manager*, which through a system model supports the construction of a complex system in terms of its more primitive components, (3) the *Task Manager*, which guarantees proper tool invocation and sequencing, (4) the *Monitor Manager*, which supports change notification, and (5) the *Advice Manager*, which provides on-line assistance for users when executing tasks.

The History Manager provides *Reserve* and *Replace* operations that are comparable to check-out and check-in respectively. Similar to the NSE, versions of a program

module are maintained via interleaved deltas within a single file. Variants can evolve in one of three different ways: as a new independent line of descent (with support for merged versions), as alternative radically different implementations, and as conditional or parametrized descendants. The system identifies the most recent version within a distinguished main derivation as the default version, although this can be changed through a version map mechanism very similar to the layers and contexts described in Section 3.3.2.

The Configuration Manager uses a system model to describe (1) how a module is composed of its components, (2) the sequence of tasks necessary to reconstruct these components in the event of a change, and (3) build dependencies that may exist among the components and other objects known to the system. If a module depends on an object that undergoes change, then it must be reconstructed according to the specified tasks.

The Configuration Manager depends on the concept of *configuration threads* to define the composition of modules. These are similar to the configurations of Section 3.2.4, implemented as described in Section 3.3.4. A configuration thread is a tabulation of component names and versions of those components to be used. Some references can be bound to specific versions while others default to the most recent version of a given component. However, this binding can be modified by a mechanism called *version maps*, which is very similar to the environments of Section 4.2.3. A *bound configuration thread* is one in which all dynamic bindings have been resolved so that all references are to specific versions. Configuration threads can be arranged hierarchically, so the configuration thread for a module can reference a configuration thread for each of its major subsystem components. A *release* is nothing more than a built system, its bound configuration thread, and a keyword to identify it.

5. Version Modeling Mechanisms and Policies

In this section, we take a different cut through the mechanisms underlying version management. Rather than present them from the viewpoint of a given model, as we did in Sections 3 and 4, we shall group the various proposals by the basic aspect of version management they are meant to support in the next subsection. It is no great surprise that much of the variation across models is due not to so much different concepts, but to different terminology for the same concept. Where possible, we present a unified terminology in Section 5.2.

5.1. Basic Mechanisms

While at first glance it may appear that an enormous number of new concepts have been introduced by the papers we have reviewed, a closer inspection reveals that there are really only a small number of notions underlying the various proposals. Broadly stated, these consist of seven classes of mechanisms: (1) organization of the space of versions, (2) dynamic configurations and dereferencing, (3) hierarchical compositions across versions (configurations), (4) alternative groupings of versions, (5) distinctions of instances versus definitions, (6) change notification and propagation, and (7) workspace organization. We shall examine how the models of Section 4 provide alternative mechanisms for each of these.

5.1.1. Organizing the Version Set

All models geared towards supporting versions have some conception of a *version set*. For example, [DITT 88] defines a design object as a set of versions. [BATO 85] use *type-version generalization* to associate a type specification with what amounts to a set of versions. They support the inheritance of common attributes, such as interface information, from types to version instances. [KATZ 86a] introduces the *version history*, which arranges the versions according to ancestor/descendent relationships. Versions are uniquely identified by system assigned version numbers, although some models use timestamps [VINE 88] and others use object IDs [BEEC 88]. Branches are called *alternatives*, with a distinguished *current* version and a *main derivation* path from the root to the current. Associated with the version history is a distinguished generic object which loosely corresponds to Dittrich and Lorie's design object and Batory and Kim's type. [LAND 86] presents a similar model, but current is indistinguishable from last created. [KLAH 86] generalizes version histories by introducing *version graphs*, in which ancestor/descendent relationships are only one of the possible threads that can link versions together. [KETA 87] call their version of the version history a *refinement graph*, and they make the incremental changes explicit in an associated *incremental refinement graph*. Their mechanism supports a true directed graph. The version model of [ECKL 87] is not very different from the version history/graph or refinement graph, and like Landis' model, current is the same as "last created" on a distinguished *principal path*. The same is true of the model of the DSEE [LEBL 84]. The contribution of the model of [BEEC 88] is to phrase generic instances and version sets in terms of a functional data

model. Thus arbitrary relationships among versions, such as first, last, successor, predecessor, etc. are easy to define and modify.

5.1.2. Dynamic Configuration Mechanisms

Most of the models we have reviewed have some mechanisms to support dynamic dereferencing of relationships between versions, usually used to support dynamic construction of configurations. [KATZ 85] distinguishes between *static* and *dynamic configuration binding*. A static reference binds to a specific version of the component, while a dynamic reference is dereferenced at run-time to some version of the component's version history. [ECKL 87] calls these *fixed* and *floating* references respectively. [BEEC 88] call them *specific* and *generic* references. [DITT 88] also calls the latter *generic* references, while [LAND 86] calls then *dynamic version references*. [BATO 85] use *parametrized versions* with type rather than instance specific references to achieve the same effect.

The models provide a variety of different ways to perform the dereferencing of dynamic references to specific versions. One set of mechanisms is based on placing versions into *layers*, and ordering the collection of layers into a *context* [CHOU 86; KATZ 85]. [DITT 88] proposes a similar method based on hierarchical *environments*, which can be ordered in manner similar to layers within contexts. The *version maps* of [LEBL 84] are essentially the same thing as Dittrich and Lorie's environments.

A second dereferencing method resolves to the current version on the the distinguished path within the version history. Both [LAND 86; ECKL 87] use this method, while [LEBL 84] defaults to this method if no version maps are specified. Actually, [ECKL 87] is somewhat more general, in that a reference can be resolved to the current with respect to a given time stamp.

[BEEC 88] provides the most general mechanism of all: the user must supply rules to drive the resolution method.

5.1.3. Hierarchical Compositions (Configurations)

There are really just two ways to represent configurations in the models we have surveyed: *flat configurations* and *hierarchical configurations*. [VINE 88] is an example of the former. A configuration is viewed as a flat table that maps objects onto their specific versions.

A more flexible approach is adopted by most of the other models. Basically, a version of a design object references specific versions of its component objects, as amended by the discussion of dynamic references in the previous subsection. [KATZ 87] introduces *composite objects*; similar mechanisms are used in [DITT 88; LAND 86]. [KETA 87] calls these *explosion refinements*. [LEBL 84] describes *configuration threads*, which are like composite objects, except that they are separated from the underlying modules rather than being stored within the module versions. [BATO 85] use *molecular objects*, but the way in which they map versions of molecular objects onto versions of their com-

ponents is not clearly stated in their model.

5.1.4. Version Grouping Mechanisms

[DITT 88] introduce logical *version clusters* as a way to group versions in arbitrary ways. [KLAH 86] provide *version partitions*, which provide a single level partitioning of the version space.

5.1.5. Instances versus Definitions

Only the models of [MCLE 83] and [BATO 85] make a clear distinction between instances and definitions. In the former, versions are viewed as instances of a type which share common attributes. Composite objects are templates with slots for instance specific attributes. When a composite object is instanced, this template is copied and the instance specific attributes are filled in with the appropriate values. A very similar approach is used in Batory and Kim's model. Instances are distinguished from definitions, and references are maintained between definitions and their associated uses.

5.1.6. Change Notification and Propagation

Several models concentrate on change notification, with limited support for change propagation. [BATO 85] proposes a timestamp based mechanism for change notification that distinguishes between the time that a designer is notified of a change and the time that he or she acknowledges the change. A variety of consistency conditions are defined based on timestamps of components being appropriately ordered with respect to the time stamps of containers. [CHOU 86] proposes a variation of the scheme, with support for both message-based (immediate) and flag-based (deferred) notifications as well as a single level of change propagation. [SUN 88] sends messages when triggered by certain "interesting" events, such as compilations.

Other operational models concentrate on supporting change propagation. [LAND 86] and [KATZ 88] view change propagation as the process of constructing a new configuration to incorporate a newly created version of one of its components. [BEEC 88] takes a similar view. As pointed out in Section 3, change propagation mechanisms must deal with the dual problems of unambiguous change paths and limited change scope to control the effects of changes. [LAND 86] and [KATZ 88] propose *pended version creation* and *group check-in* respectively to unambiguous define the propagation effects. [VINE 88] uses *version sensitive* and *change sensitive* relationships to control the path of changes. Finally, [KATZ 88] and [RUMB 88] propose using *propagation attributes* to limit the range of propagation.

5.1.7. Object Sharing Mechanisms

Most of the models we have described provide some flavor of workspace model for supporting the sharing of objects among multiple designers. Objects move between workspaces as the result of *check-in/check-out* operations. [SUN 88] refers to these op-

erations as *acquire*, *modify*, and *reconcile*, the latter being a special operation for merging together derivation paths in the version history. [LEBL 84] calls them *reserve* and *replace*.

A second variation among the models is how the shared repositories are named and how they are arranged hierarchically. Usually they are called *workspaces*, as in [KATZ 87] and [CHOU 86], but sometimes the terms *database* [KETA 87], *federation* [ECKL 87], and *environment* [SUN 88] have been used. Although the models support the construction of multiple levels of repositories, most agree that at least three levels are necessary. These have been called *archive*, *group*, and *private* in [KATZ 87; CHOU 86]; *parts*, *projects*, and *private* in [KETA 87]; and *release*, *integration*, and *development* in [SUN 88].

5.2. The Grand Unification

Much of the above discussion has pointed out the similarity in the proposed models. In this subsection, we shall propose a unified terminology where a conceptual consensus appears to exist, based wherever possible on the model of Section 3.

5.2.1. Organizing the Version Set

The key concepts we introduce here are *version history*, *generic object*, *ancestor/descendent relationships*, *main derivation*, *branches*, and *current*. Version instances are objects in their own right, and are uniquely identified to the system. They may be identified by the user through a unique version number or time stamp, implemented as distinguished attributes. Version instances are related to a common generic instance, from which attributes and default values can be inherited. They are also interrelated to each other by system maintained ancestor/descendent relationships, as well as any other relationships created by the users. These may be implemented by links as in [KLAH 86] or functional relationships as in [BEEC 88].

Ancestor/descendent relationships are used for placing an ordering on the version history. This has also been called the version set [DITT 88; BEEC 88], version graph [KLAH 86], and refinement graph [KETA 87]. A single version instance is identified as the current version. We feel it is important to decouple the notion of current from last created. There is no such decoupling in the models of [LEBL 84; LAND 86; ECKL 87]. The path from the root to the current is called the main derivation [LEBL 84]. This is the same as the principal path in [ECKL 87] and the default branch in [LAND 86]. Alternatives are represented by branches in the version history. This has been called variants in [LEBL 84] and derivations in [ECKL 87]. Alternative paths can be merged to form a single subsequent path, thus the history is a directed graph rather than a tree. This has been called a merged line of descent in [LEBL 84], a reconciliation in [SUN 88], and a consolidation in [ECKL 87]. These terms are summarized in Figure 5.1.

5.2.2. Dynamic Configuration Mechanisms

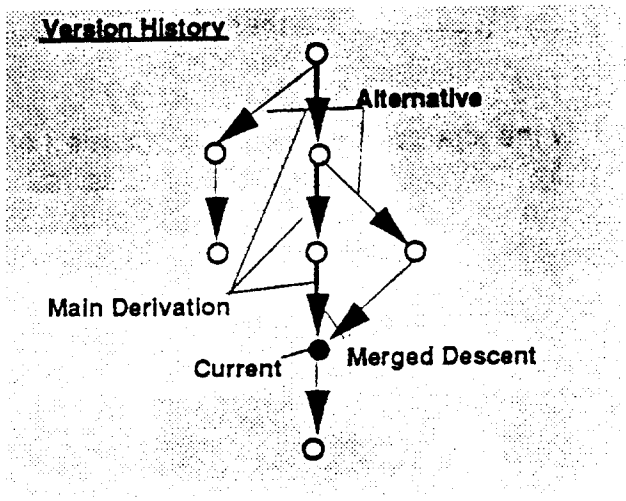

FIGURE 5.1: Version Terminology
A version history is shown with versions related by anscestor/descendent links. Alternatives are branches. The main derivation traces a path from the root version to a distinguished current version, shown as a black dot. A merged descent is when two or more alternative paths are merged together in the creation of a new version.

We adopt the terminology presented in Section 3: static references bind to specific versions, while dynamic references refer to generic objects and must be dereferenced to a specific version for certain operations, such as check-in/check-out. While the rule-based method of [BEEC 88] is clearly the most general, it is not clear exactly how to formulate rules that would allow the resolution order to be permuted in a flexible and easy to understand manner. Therefore, we prefer the concept of hierarchical environments as proposed in [DITT 88]. The mechanism has the advantage of being hierarchical, i.e., the internal bindings of a submodule can be defined within an environment associated with that particular submodule. The order is easily permuted to create "what-if?" configurations. Finally, both the "layers and contexts" approach and the "current on a path" kinds of mechanisms can be supported within the environment model.

5.2.3. Hierarchical Compositions (Configurations)

The *composite object* terminology of Section 3 is quite adequate to describe configurations in terms of versions of composite objects constructed from versions of their components. Note that Figure 3.7 shows how configuration mappings are separated from the versioned data, so the implementation of configurations is essentially identical to DSEE's configuration threads. An additional mechanism to describe *flattened configurations*, constructed from walking the object hierarchy to create a list of objects and their associated versions is also of use, especially for archived configurations, and should be supported in a version model.

5.2.4. Version Grouping Mechanisms

The logical version clusters of [DITT 88] are certainly more general than the version partitions of [KLAH 86]. The creation of hierarchical version clusters should be orthogonal to the descendent and ancestor relationships already supported by the version history. Figure 5.2 shows how version clusters can be grafted on top of the version history.

5.2.5. Instances versus Definitions

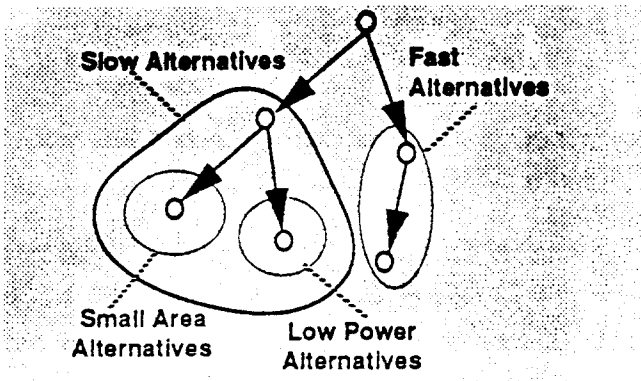

FIGURE 5.2: Grouping Mechanisms
*The figure shows a version history overlaid with logic version clusters. The **small area** and **low power** alternatives are subclusters of the **slow** alternatives. Orthogonal to these are the **fast** alternatives. Clusters are permitted to overlap in any desired manner. Note how version clusters naturally allow names to be associated with branches (i.e., alternatives) in the version history.*

Because of the need to maintain instance specific attributes, it is not possible to describe the component hierarchy simply as a directed acyclic graph with arcs representing uses and nodes representing definitions. A distinction needs to be made between a definition hierarchy and an instance hierarchy. Figure 5.3 shows the relationship between the two for a simple example of a register file consisting of sixteen instances of a register. The instance hierarchy makes reference back to the definition hierarchy to associate instances (uses) with their definitions.

5.2.6. Change Notification and Propagation

Change notification and propagation is a relatively new area, and one in which there is certainly more work to be done. Change notification is fairly straightforward, and any of the message or flag-based approaches seem sufficient. Change propagation, on the other hand, is a more difficult and challenging problem. *Change propagation* as the incorporation of new versions within the configuration hierarchy appears to be a widely accepted terminology.

Mechanisms to disambiguate the change effects and to limit its scope are needed in any propagation mechanism. To disambiguate the change path, we have described some mechanisms based on group check-in. To limit change scope, the propagation attribute approach of [RUMB 88] can be made to cover the timestamp methods of [BATO 85] and [CHOU 86], as well as the change sensitive relationships of [VINE 88], and is more general. Such a mechanism should be incorporated within change propagation.

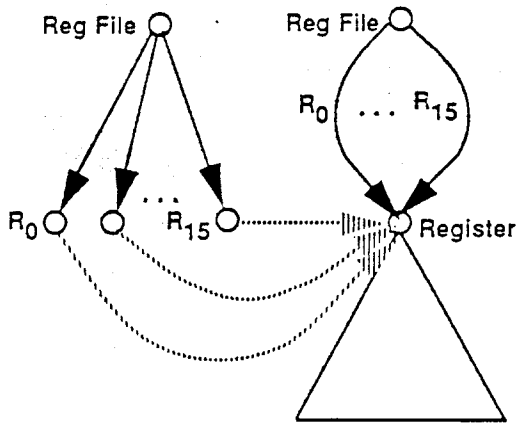

FIGURE 5.3: Instances vs. Definitions
The definition hierarchy on the right while the instance hierarchy is on the left. The register file is defined in terms of sixteen instances of the register. If instance specific attributes are to be associated with the individual registers, then sixteen separate instances must be created, eaching referencing back to the original register definition. The instance and definition hierarchies can be comingled in the sense that separate instance nodes are needed in the case that an object can have instance specific attributes.

5.2.7. Object Sharing Mechanisms

Despite the varying terminology, the basic concepts reduce to *private*, *group/project*, and *public/archive workspaces*, with objects moved among the workspaces through *check-in* and *check-out* operations. The *federation* concept of [ECKL 87] is a natural extension from single site workspaces to workspaces that are spread across multiple sites. It also clarifies who has what kind of access to which workspaces. It is straightforward to combine the federation notion with multi-layer workspaces.

6. Summary and Conclusions

In this paper, we have described the general requirements for version management systems, presented one model in some detail to highlight these requirements, and have reviewed a collection of representative proposals for version management mechanisms from the literature. Despite the large number of proposals, each with its own idiosyncratic terminology, we were able to see that most proposals were really minor variations on a small number of themes.

A major goal of this paper has been to attempt to unify the terminology of these disparate proposals. In certain areas, we have been successful. There is really little that separates the mechanisms for organizing version histories, for example. Most of the apparent differences are really due to variations in terms rather than fundamental differences in underlying concepts. This was also true of support for dynamic configurations and workspaces. In other cases, we were able to identify mechanisms that generalized on previously proposals. For example, propagation attributes for limiting change propagation can also be used to implement the variety of timestamp methods that have been proposed, and thus represent a more fundamental mechanism upon which a variety of different approaches can be implemented.

While we would have liked to have been able to present the ultimate version management model, we do not feel that such a model is likely to exist for some time. Our preferred model would be based on the proposal of Section 3, extended with liberal sprinklings of ideas from the literature we have reviewed: the logical version clusters and hierarchical environments of [DITT 88], the federation concept of [ECKL 87], the propagation attributes of [RUMB 88], and the merged path version history of a number of proposals.

The final challenge, that of creating a single framework, based on the mechanisms we have identified, that can be tailored for the needs of a given version environment, is something that is still left to be accomplished. This paper represents a first step in that direction, but much work remains before we can fully understand how best to support versions in design environments.

7. Bibliography

BANE 87 Banerjee, J., W. Kim, H-J Kim, H. F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proc. 1987 SIGMOD Conference, San Francisco, CA, (May 1987).

BATO 84 Batory, D., A. Buchmann, "Molecular Objects, ADTs, and Data Models: A Framework," Proc. Conference on Very Large Databases, Mexico City, Mexico, (August 1984).

BATO 85 Batory, D., W. Kim, "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems*, V. 10, N. 3, (September 1985), pp. 322-346.

BEEC 88 Beech, D., B. Mahbod, "Generalized Version Control in an Object Oriented Database," Proc. IEEE Data Engineering Conference, Los Angeles, CA, (February 1988), pp. 14-22.

BHAT 87 Bhateja, R., R. H. Katz, "A Validation Subsystem of a Version Server for Computer-Aided Design Data," Proc. 24th ACM/IEEE Design Automation Conference, Miami, FL, (June 1987).

CHOU 86 Chou, H., W. Kim, "A Unifying Framework for Version Control in a CAD Environment," Proceedings 12 VLDB Conference, Kyoto, Japan, (August 1986).

DITT 88 Dittrich, K., R. Lorie, "Version Support for Engineering Database Systems," *IEEE Trans. on Software Engineering*, V. 14, N. 4, (April 1988), pp. 429-437.

ECKL 87 Eckland, D. J., E. F. Eckland, R. O. Eifrig, F. M. Tonge, "DVSS: A Distributed Version Storage Server for CAD Applications," Proc. 13th VLDB Conference, Brighton, England, (September 1987), pp. 443-454.

GEDY 88 Gedye, D. M., R. H. Katz, "Browsing the Chip Design Database," Proc. 25th ACM/IEEE Design Automation Conference, Anaheim, CA, (June 1988).

GOLD 81 Goldstein, I., D. Bobrow, "Layered Networks as a Tool for Software Development," Proc. 7th Intl. Conference on Articial Intelligence, (August 1981).

HASK 82 Haskin, R. L., R. A. Lorie, "On Extending the Functions of a Relational Database System," Proc. ACM SIGMOD Conference, (May 1982), pp. 207-212.

KATZ 83 Katz, R. H., "Managing the Chip Design Database," IEEE Computer, V, N 12, (December 1983).

KATZ 84 Katz, R. H., T. J. Lehman, "Database Support for Versions and Alternatives of Large Design Files," IEEE Transactions on Software Engineering, V SE-10, N 3, (March 1984).

KATZ 85 Katz, R. H., *Information Management for Engineering Design*, Springer-Verlag Computer Science Survey Series, Heidelberg, West Germany, 1985.

KATZ 86a Katz, R. H., E. Chang, R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases," Proc. ACM SIGMOD Conference, Washington, DC, (May 1986).

KATZ 86b Katz, R. H., M. Anwarrudin, E. Chang, "A Version Server for Computer-Aided Design Data," Proc. 23rd ACM/IEEE Design Automation Conference, Las Vegas, NV, (June 1986).

KATZ 87 Katz, R. H., R. Bhateja, E. Chang, D. Gedye, V. Trijanto, "Design Version Management," *IEEE Design and Test*, V 4, N 1, (February 1987).

KATZ 88 Katz, R. H., E. Chang, "Managing Change in a Computer-Aided Design Database," Proc. 13th Conference on Very Large Databases, Brighton, Engiand, (September 1987). Also in *Readings in Object-Oriented Databases*, S. Zdonik, D. Maier, eds., Morgan-Kaufman Publishers, San Mateo, CA, 1988.

KATZ 89 Katz, R. H., E. Chang, "Inheritance Issues in Computer-Aided Design Databases," in *Object-*

Oriented Database Systems, K. Dittrich, U. Dayal, eds., Springer-Verlag, Berlin, West Germany, 1989.

KETA 87 Ketabchi, M. A., V. Berzins, "Modeling and Managing CAD Databases," *I.E.E.E. Computer Magazine*, (February 1987), pp. 93-102.

KLAH 86 Klahold, P., G. Schlageter, W. Wilkes, "A General Model for Version Management in Databases," Proc. VLDB Conference, Kyoto, Japan, (August 1986), pp. 319-327.

LAND 86 Landis, G. S., "Design Evolution and History in an Object-Oriented CAD/CAM Database," Proceedings 31st COMPCON Conference, San Francisco, CA, (March 1986).

LEBL 84 Leblang, D. B., R. P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment," Proceedings ACM SIGPLAN/SIGSOFT Conference on Practical Software Development Environments, (April 1984).

LORI 83 Lorie, R. L., W. Plouffe, "Complex Objects and Their Use in Design Transactions," ACM SIGMOD Conference, San Jose, CA, (June 1983).

MCLE 83 McLeod, D., K. Narayanaswamy, K. Bapa Rao, "An Approach to Information Management for CAD/VLSI Applications," Proceedings SIGMOD Conference on Databases for Engineering Applications, San Jose, CA, (May 1983), pp. 39-50.

RUMB 88 Rumbaugh, J., "Controlling Propagation of Operations Using Attributes on Relations," Proc. OOPSLA '88 Conference, (September 1988), pp. 285-296.

SUN 88 SUN Microsystems, "Introduction to the NSE," Part No. 800-2362-1300, (March 7, 1988).

VINE 88 Vines, P., D. Vines, T. King, "Configuration and Change Control in GAIA," Proc. OOPSLA '88 Conference, (September 1988).

WIED 82 Wiederhold, G., A. Beetem, G. Short, "A Database Approach to Communication in VLSI Design", *IEEE Transactions on Computer-Aided Design*, V 1, N, (1982).