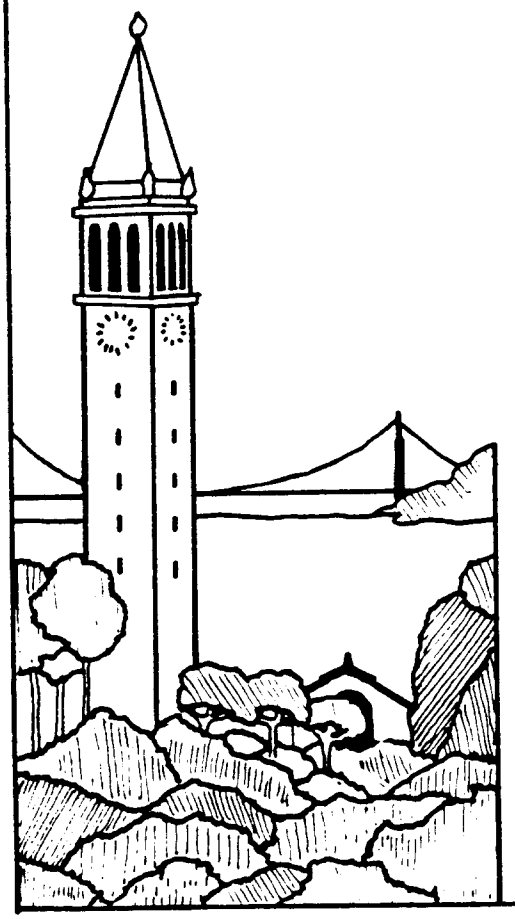
Version Modeling Concepts for Computer-Aided Design Databases

Randy H. Katz, Ellis Chang, and Rajiv Bhateja



Report No. UCB/CSD 86/270

November 1985

Computer Science Division (EECS)
University of California
Berkeley, California 94720

VERSION MODELING CONCEPTS FOR COMPUTER-AIDED DESIGN DATABASES[1]

Randy H. Katz, Ellis Chang, Rajiv Bhateja

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

ABSTRACT: We describe a semantic object-oriented data model for representing how a complex design database evolves over time. Structural relationships, introduced by the data management system, are imposed on the objects created by existing CAD tools. The relationships supported by the model are: (1) version histories, (2) time-varying configurations, and (3) equivalences among objects of different types. We describe mechanisms for (1) identifying current versions, (2) supporting dynamic configuration binding, and (3) verifying equivalence relationships. The data model is being implemented in a *Version Server*, under development at the University of California, Berkeley.

KEY WORDS AND PHRASES: Versions, Configurations, and Equivalences; Design Database; Object-Oriented Database; Time in Databases;

1. Introduction

Computer-aided design databases have emerged as an active area of research. Database systems have been extended with *complex objects* and *design transactions,* to enable them to better store and manipulate design data [HASK82]. Semantic data models have been proposed for describing the special kinds of relationships among design data [KATZ83, MCLE83, BATO85a, AFSA85], and for exploiting them to maintain the integrity of the design [NEUM83].

Meanwhile, there has been a growing awareness of the importance of modeling time in databases. With the exception of some recovery mechanisms, database systems make available only the current values of database objects. It is convenient for some applications if the database is seen as evolving over time: a record is stored as multiple instances. Proposals have been made for augmenting existing data models and manipulation languages with time semantics (e.g., [ARIA83]).

These research areas intersect in version handling for design databases [KATZ84a, BATO85a]. However, there is much confusion over exactly how versions should be represented.

[1]Research supported by the National Science Foundation under Grant ECS-8403004.

Implementation issues, such as data compression, are confused with representation issues. Early proposals merely suggested extending existing records with a timestamp, without any explicit support from the database system. More recent work has introduced modeling constructs for versions, but these do not extend time-variation to relationships defined over versions. In particular, *time-varying configurations* are not adequately supported.

By focusing on design applications, we can identify specific time sensitive relationships worthy of special support in a data model for design. A given portion of the design may have many instances over time *(version histories)*, may be constructed hierarchically from components *(configurations)*, and may correspond to other portions in different representations *(equivalences)*.

In this paper, we concentrate on data modeling issues: how to represent and manipulate a design database as it evolves over time. This is an extension and further elaboration of [KATZ85]. Our modeling decisions have been tempered by a pragmatic desire to avoid modifying existing design tool interfaces. We have carefully chosen our primitives so they can be imposed with little effort on existing collections of design objects, created by existing design tools. In fact, most of our "design objects" could be conventional design files.

In this work, the design database is only loosely coupled with design tools. The database structure is used by the design management system to locate objects relative to their relationships with other objects. Once an object is found, it is accessed by the design tools through a conventional file system interface. We have not yet explored how to provide a tighter coupling, in which design tools make direct calls on a database system to access design objects.

The rest of the paper is organized as follows. In the next section, we present an object model for design data, and describe the issues in specifying its structure to model a database across time. In section 3, the operations for manipulating the data structure are presented. Section 4 briefly describes an operational system in which these ideas are being implemented: we call it a *Version Server*. Section 5 compares our work with previous proposals. Finally, our status and conclusions are given in section 6.

2. Object Model/Data Structures

2.1. Modeling Primitives

We view a design database as a **(large) collection of objects** that together describe an artifact being designed. *Object* is a convenient term for identifying useful aggregates of design data without committing to an implementation in terms of records, tuples, or files. The aggregation size is left unspecified. It could be large (an entire subsystem of the design), or very small (an individual design primitive) at the discretion of the object's definer. Typically a design object will contain up to a few hundred modeling primitives, e.g., lines of code, layout geometries, etc. "Objects" are usually packages of data and manipulation procedures, but we do not assume this in the following discussion.

Design descriptions exist across representations. For example, a VLSI design is specified by layout, transistor, and functional descriptions. Thus, design objects containing representation-specific data are *typed:* a layout object, a transistor object, a functional object.

Design descriptions are composed hierarchically. A given object can be described in terms of component objects. For example, a datapath layout object can be described by a combination of layout primitives and the composition of ALU, register file, and shifter layout objects. Objects built in this way are *composite,* while objects without components are *primitive.* A composite object and its components are *type-homogeneous.*

Existing design synthesis tools are specific to a particular design representation. However, designers must span representations during analysis and verification. A design is described by a collection of composite objects, one for each of its representations. Each is the root of a hierarchy of object compositions, called a *representation hierarchy.* Since it is convenient to identify objects in different representations that describe the same real-world artifact, we introduce *equivalence relationships.* They are *type-heterogeneous,* provide the necessary linkages across representations, and can be exploited in maintaining design consistency (see Section 3.3).

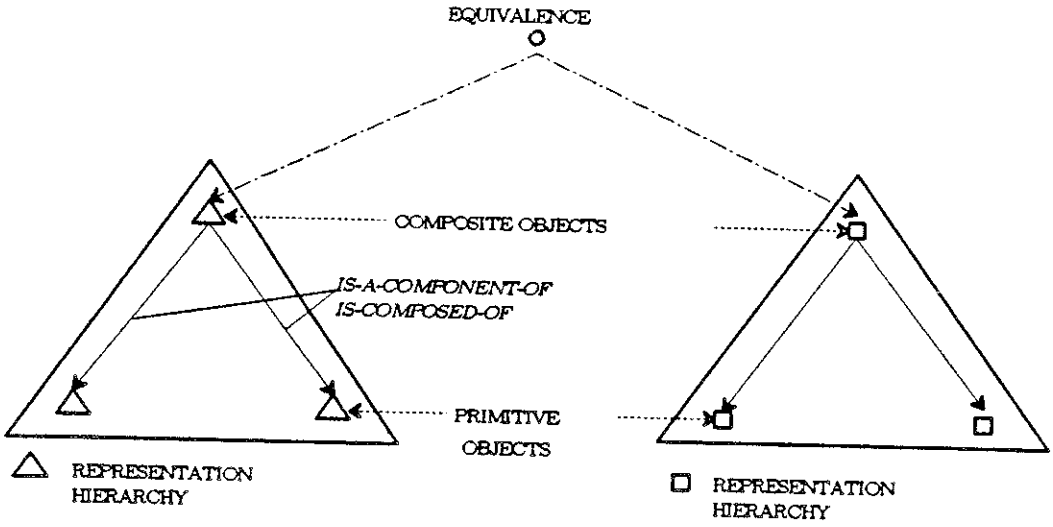


Figure 2.1 – Composite/Primitive Objects; Representation Hierarchies and Equivalences; Representation objects are typed, e.g., triangle or square. They can be primitive, or composite, i.e., hierarchically related to other objects of their type, which are components. Correspondences across representations are indicated by equivalence relationships, explicitly represented by distinct equivalence objects.

These concepts are summarized in Figure 2.1. They can be found in almost every design organization known to the authors. For example, VLSI design files contain "cells", each of which is built from design primitives and compositions of subcells. A similar observation can be made about software systems, which are usually described as hierarchical collections of source, object, and executable modules. All too often the equivalence relationships exist, but are implicit in naming conventions, e.g., "x.c" for source, "x.o" for object, and "x" for executable.

2.2. Adding Time: Versions and Configurations

Consider the representation of a microprocessor datapath layout. It is a composite object with components that describe the register file, ALU, and shifter. Because the design is under continuous revision, many descriptions of the datapath layout may exist at different points in time.

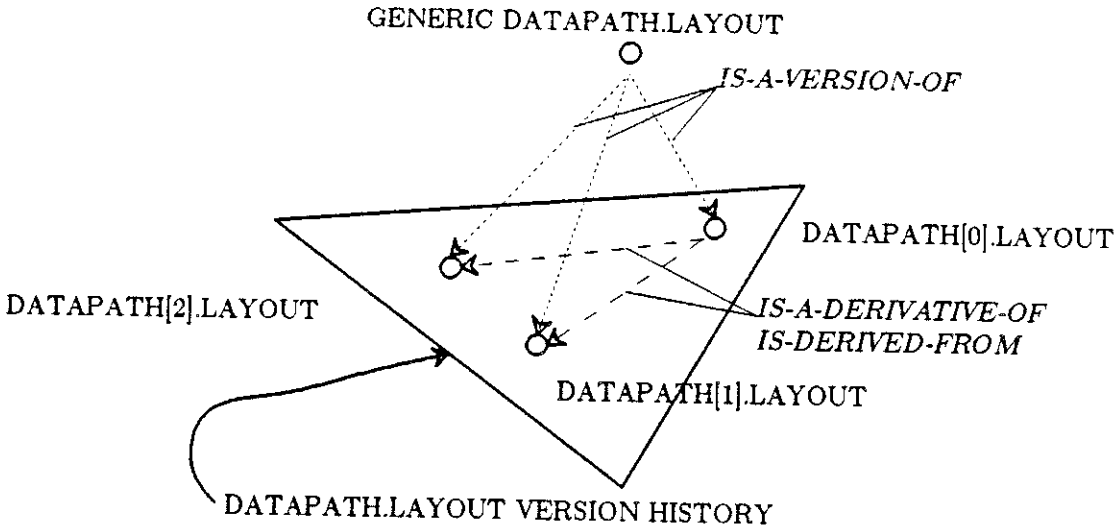


Figure 2.2 -- Datapath Layout Version History

Version objects are organized into a derivation history: e.g., Datapath[1].layout is derived from Datapath[0].layout. This means that version 1 was created by applying a set of changes to version 0. Each of these objects is also related to an associated "generic" object: datapath.layout. The latter relationships are essentially generalizations, supported by most semantic data models.

Individual instances of the datapath layout are full-fledged representation objects, directly manipulated by design tools. No design tool creates the "generic" datapath layout object. It must be added to the database to interrelate the representation objects that are different *versions* of the same *generic object*. Hence, it is not a representation object, but a *structural object*, added by the data management system to organize the objects created by design tools. Associated with each generic object is a *Version Plane*, which is simply a graphical representation of the relationships (1) between the generic datapath layout object and its versions, and (2) among the version objects themselves (see Figure 2.2).

The relationships among versions have not been adequately recognized in other version data models. To maintain a true *version history*, it is not enough to merely group together versions of the same object. The derivation sequence must be represented explicitly. Timestamps are insufficient because the version history is a tree: several versions may be derived from the same parent. These parallel versions are often called *alternatives*.

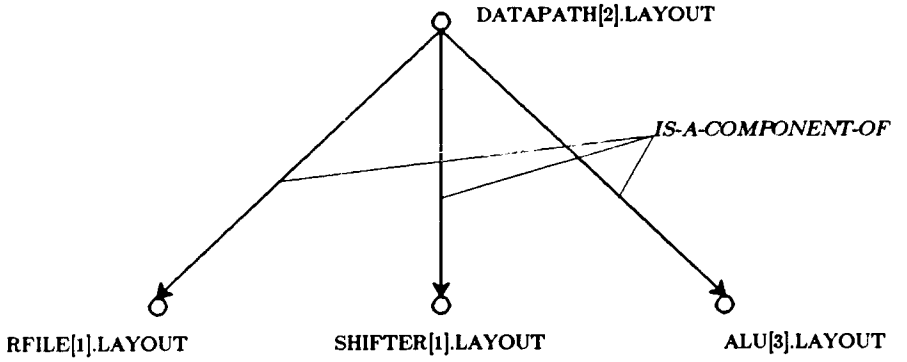


Figure 2.3 -- Datapath Layout Configuration

Configurations differ from simple composite objects in that the composition must bind to some version of the components.

Each datapath layout version is a composite object, composed from register file, ALU, and shifter objects. A datapath layout *configuration* is composed from specific versions of these generic objects. Thus, configurations correlate versions of composite and component objects (see Figure 2.3). Most proposals do not support time-varying configurations.

Sometimes it is useful if the relationship between composite and components can be left unbound. A version of the datapath layout contains some ALU version, but exactly which one is determined dynamically. An operational mechanism for supporting dynamic configurations will be presented in Section 3.

2.3. Constraints on the Database: Equivalence Relationships

Equivalences identify correspondences across representations. For example, a datapath layout version corresponds to some datapath transistor version and some datapath functional version. An equivalence *structural* object, the "datapath," is introduced to interrelate these (see Figure 2.4).

From an operational viewpoint, equivalences are also constraints: the correspondences between objects must be verified by the design team. Sometimes they can be established

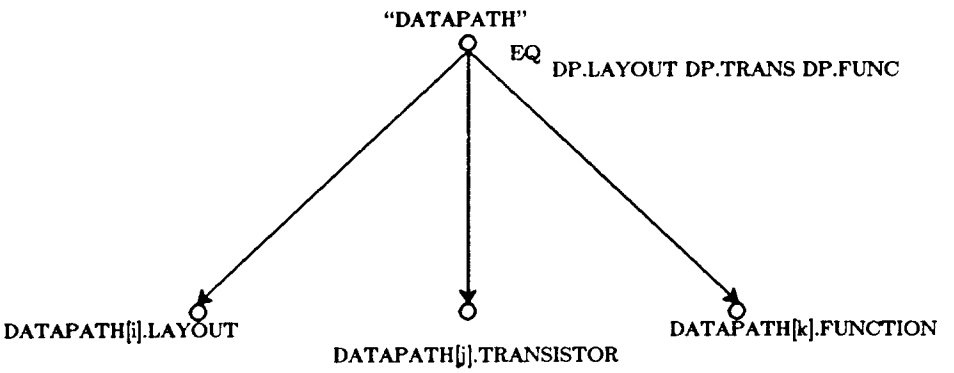


Figure 2.4 -- Equivalence Objects

Equivalence objects tie together objects of different representations that describe the same real world artifact. The equivalence object can be viewed as representing that artifact, while the associated representation objects its description in the appropriate type.

automatically. For example, an "object code" object is equivalent to a "source code" object if it has been compiled from it. Otherwise, the validity of equivalences is determined by the successful execution of analysis tools. Mechanisms to support the verification of equivalences are described in Section 3.

2.4. Further Modeling Issues

So far we have introduced primitive and composite representation objects, and have defined over these version, configuration, and equivalence relationships. In doing so, we have introduced composite structural objects for grouping versions (generic objects) and equivalents (equivalence objects). By recursively applying these relationships, further useful structure can be imposed on design objects. If representation objects are 0^{th} level and the structural objects introduced so far are 1^{st} level, then we will now examine 2^{nd} level relationships.

It is useful we can model the evolution of equivalence objects (see Figure 2.5). Consider two representation objects A and B, related through equivalence object EQ_{AB}. A new version of A,

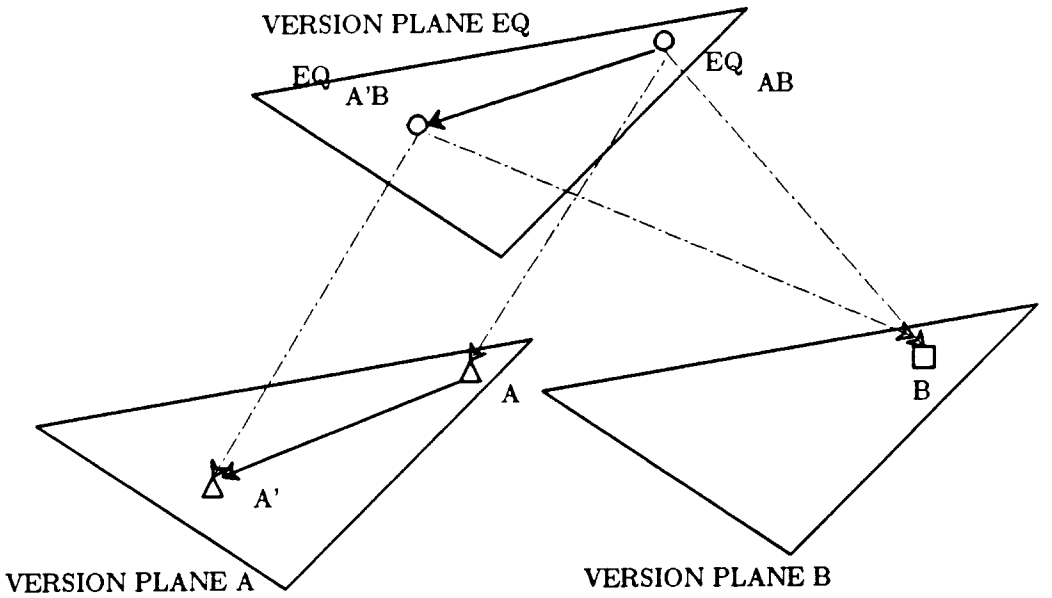


Figure 2.5 – Versions of Equivalences

Equivalence objects evolve in much the same way as representation objects. $EQ_{A'B}$ *is-derived-from* EQ_{AB} to represent the new equivalence between A' and B.

A', is created and shown to be equivalent to B,[2] introducing the equivalence object $EQ_{A'B}$. $EQ_{A'B}$ is actually a derivative of EQ_{AB}. As an example, suppose that the equivalence objects link different representations of the microprocessor datapath. Then the second-level generic object, associated with the equivalence object version plane, represents the version history of the datapath (independent of representation).

There are some advantages if equivalence objects can be composite and can participate in configurations. Their configurations are inferred from the objects they range over (see Figure 2.6a). If an equivalence object interrelates composite representation objects, and if the components of these are also interrelated by equivalence objects, then the equivalence objects can form a component hierarchy as shown in Figure 2.6b.

[2] Equivalences are demonstrated by special analysis tools that are part of the design environment. We will say more about these in Section 3.3.

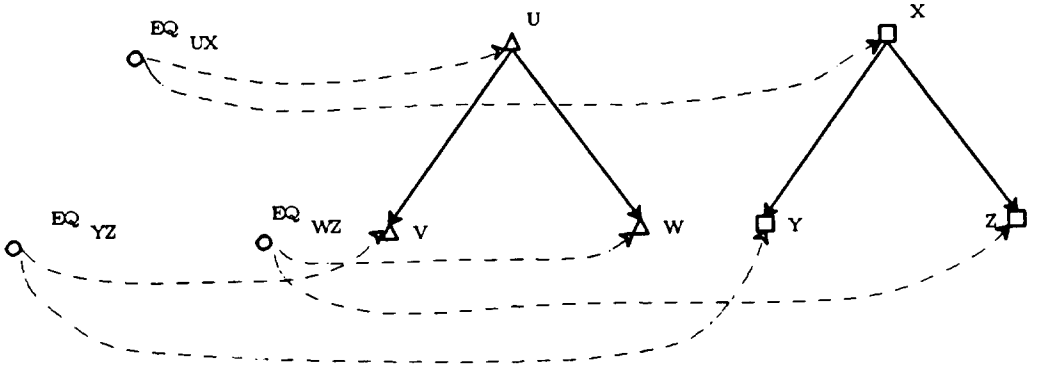


Figure 2.6a -- Separated Representation Hierarchies

The triangle and square objects form representation hierarchies. Their hierarchical structures can be clustered independently of equivalence relationships. Solid lines are compositions, while dashed lines are equivalence relationships.

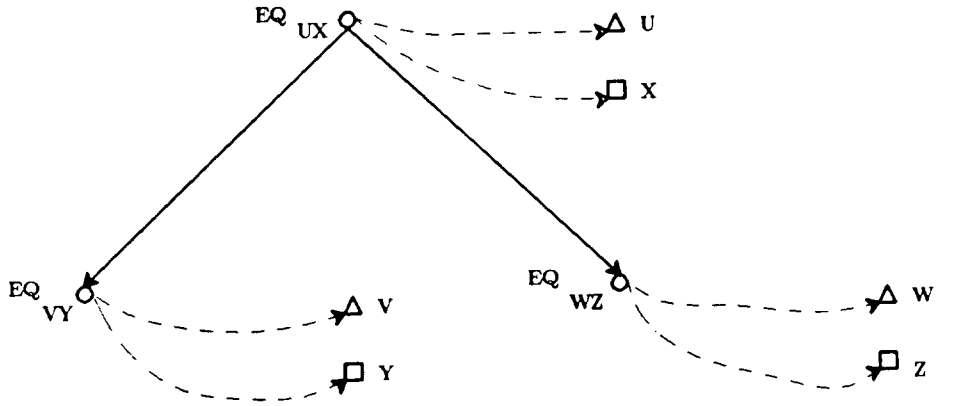


Figure 2.6b -- Clustering by Equivalence

The equivalence objects have inherited their compositions from the objects they range over. The composition relationships among the triangle and square objects still exist, but are not shown in the figure. Objects can be clustered across representations first by their equivalence relationships, and then by the compositions of the equivalence objects.

By explicitly denoting correspondences, we retain flexibility by not forcing every object in one representation to have equivalent objects in all other representations. Thus, representation hierarchies remain independent, and need not be *isomorphic* (i.e., all representation hierachies are forced to have identical compositions). However, when the hierarchies are isomorphic, the hierarchical composition of equivalence objects provides a way to cluster representation objects both across representations and within compositions.

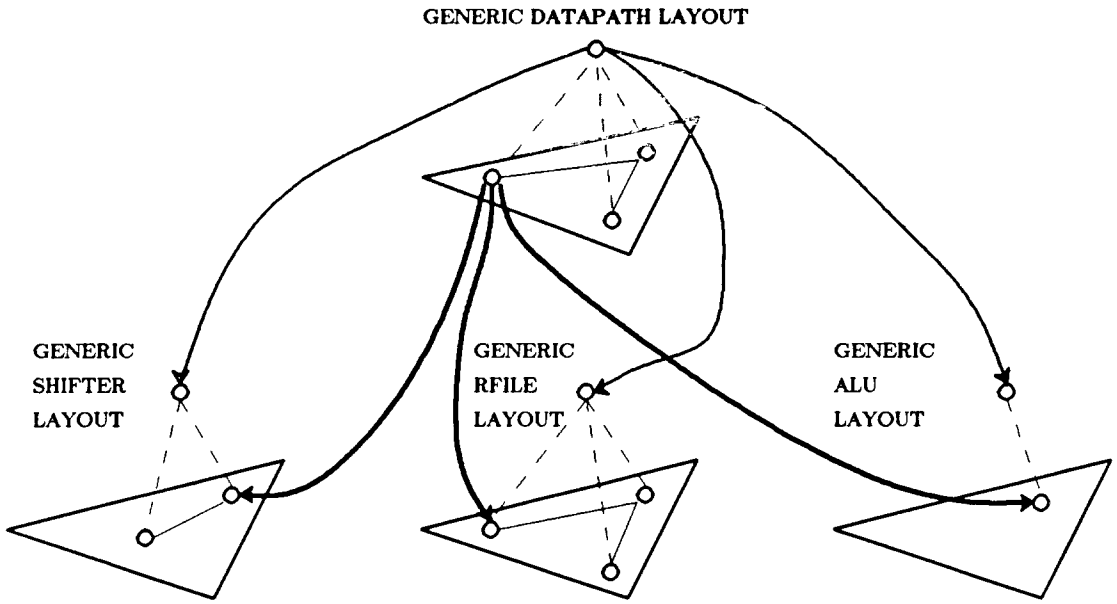


Figure 2.7 -- Configurations of Generic Objects

Individual representation objects identify their components explicitly (heavy solid lines). These can be inherited by the associated generic object (medium solid line) only if every composite it ranges over is composed from a shifter, rfile, and alu instance from the component version planes.

Configurations of generic objects can be constructed as for equivalence objects (see Figure 2.7). A generic object can be configured from component generic objects only if all of its associated versions incorporate some version of the component generic objects. For example, if some datapath layout versions have no shifter, or some make use of a completely different kind of register file, then the generic object ranging over datapath layouts cannot be configured from either the shifter or the register file generic objects.

The composition of generic objects implies that a datapath could be constructed from any combination of shifter, register file, and ALU, but this is usually not the case. Particular versions of these must be compatible before they can be composed into a valid datapath, and such constraints are not obvious from the structure formed in Figure 2.7. A related idea is to permit configurations to span both 0^{th} and 1^{st} level objects. For example, a particular datapath layout object could be configured from a particular register file layout object, and the version histories of

the ALU and shifter. In essence, the choice of which ALU and shifter layout should be incorporated into this datapath layout object is left unbound. Again, this has a problem with compatible compositions of the ALU and shifter. We will describe an operational mechanism in the next section that supports dynamic configuration binding while also supporting the concept of compatible configurations.

The higher order relationships we support are summarized in Figure 2.8. Other possible relationships have been eliminated. It does not appear useful to support versions of versions. The essential point of versions is that updates are not performed in place: a sequence of updates are applied to an object to create a new derivative of that object. However, someplace in the system updates must be performed in place. Thus, we have decided that creating a new version directly updates the associated generic object, rather than creating a new version of it. Similarly, we disallow equivalences to be defined over equivalence objects. While it may be useful to have some notion of corresponding equivalence constraints, it is not clear how this would be established. Neither do we support equivalences among generic objects. Again, it is not clear how such equivalences would be established.

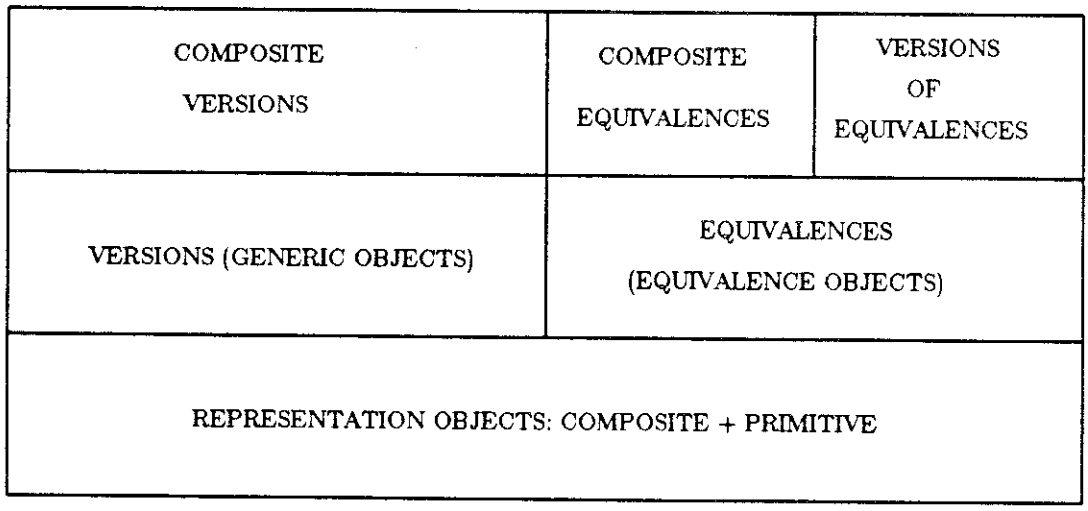


Figure 2.8 – Higher Order Relationships

3. Object Model/Operations

3.1. Currency Within the Version History

Without some control mechanisms, version histories can branch widely. It is useful if a preferred version can be identified from which new derivatives should be created. This is accomplished with a *currency* indicator: new derivatives can be created from previously superceded versions, as long as they are descendents of the current version. Currency can be set explicitly, to allow designers to follow any desired system release policy.

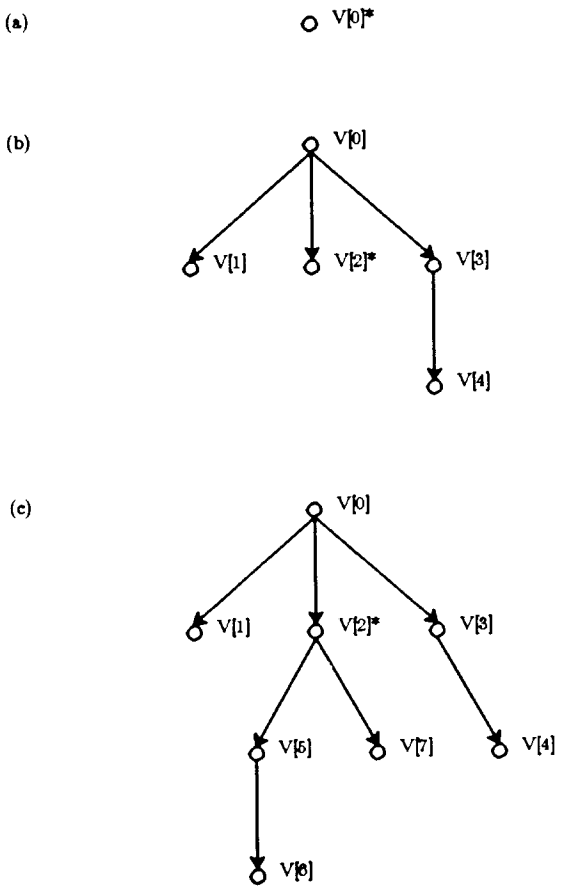


Figure 3.1 – Example Derivation of a Version History

Initially, V[0] is the current version. After currency moves to V[2], no further derivations can be made from V[0], V[1], V[3], or V[4] without repositioning the currency.

Consider the derivation sequence shown in Figure 3.1. V[0] is created and made the current version (see Figure 3.1a). V[1], V[2], and V[3] are created as alternative derivatives of V[0]. V[4] is then derived from V[3]. At this point, V[2] is set to the current position (see Figure 3.1b), and no further derivatives of V[1], V[3], or V[4] can be derived without changing currency. V[5] is derived from V[2], and in turn, V[6] is derived from V[5]. Note that V[2] remains the current version (see Figure 3.1c). Thus, it is possible to create V[7] as a new alternative derived from V[2], even though V[6] has superceded V[2]. If V[6] is now made current, further derivations from V[2] or V[7] would be disallowed.

3.2. Dynamic Configuration Binding

A version of a composite object is formed from versions of its components. Instances can be bound at the time the composite is created, or can be left unspecified until the object is accessed. The latter approach, *dynamic binding*, is most useful during the exploratory phases of design, when alternative new versions are being evaluated. At some point in an object's lifetime, its configurations must be bound to specific versions, usually when it is "released."

Layers, first proposed in [GOLD81], support dynamic configurations. The database is partitioned into *layers* that correlate versions among related objects. The initial layer contains the original versions, the second layer contains newly added objects and new versions of existing objects, etc. A composite object identifies its components by referencing their associated generic objects. At least conceptually, the binding to actual versions takes place by searching through the design layers for the first encountered version of the desired object.[3]

The power of layering is that the designer determines which versions will be bound simply by *specifying the layer search order*. The choice of ordering is an *environment*, and all object accesses are evaluated with respect to one of these. There can be many user-defined environments.

[3]This search can be implemented efficiently as an index structure mapping unique object identifiers into object versions, taking account of the specified order of the layers (e.g., see [KATZ84a]).

As an example, consider the creation of layers as shown in Figure 3.2. By creating environments from different sequences of layers, different instances of the ALU and the Register File can be bound. If the environment is formed from layers 0, 1, 2, and 3, then the ALU is bound to instance 2 and the Register File is bound to instance 2. If the environment is formed from layers 0, 1, and 2, then the instances bound are 2 and 1 respectively. If the layers are sequenced as 0 followed by 1, then the ALU instance is 1 and the Register File instance is also 1. If the environment contains just layer 0, then the ALU[0] and RFile[0] are the instances bound. As a final example, an environment constructed from layer 0 followed by layer 3 would yield ALU[0] and Rfile[2] as the bound objects. Note that it is not possible to create a context that binds ALU[0] and RFile[1], because of the grouping of ALU[1] and Rfile[1] in the same layer.

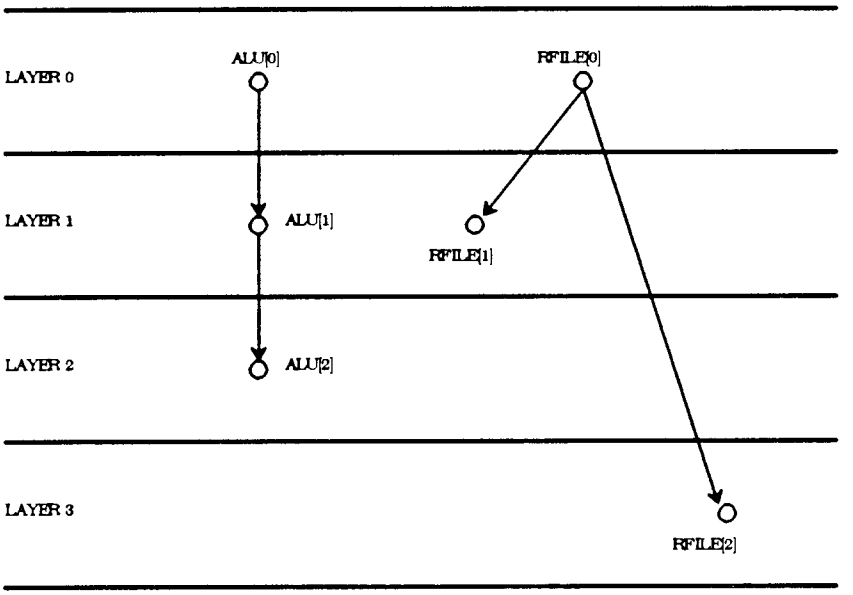


Figure 3.2 – Layers and Environments Example

The Version Histories are partitioned into layers as shown. Layers can be shuffled to make some versions dominate others. For example, if layer 1 dominates layer 2, then a reference to the ALU will be bound to ALU[1] rather than the newer ALU[2].

The mechanism as presented provides a primitive way to constrain allowable configurations to those that are *consistent*, i.e., the interfaces of the components are compatible in how they are interconnected to realize the composite object. A more sophisticated approach would introduce compatibility relationships among objects of the same type, in a manner similar to equivalences. For example, ALU[0] *is-compatible-with* Rfile[2], and thus can participate in the same configuration of the datapath. The approach is still under investigation.

3.3. Validation

Part of the function of any database system is to keep its databases *consistent*. In design databases, consistency enforcement is closely tied to the mechanisms that permit a design to be released to a user community. One such mechanism, based on object check-in to an archive, is described in the next section. Consistency is usually determined by the successful execution of sequences of validation tools. It is beyond the scope of the data management system to automatically invoke such sequences, and to determine whether they are successful. However, it can still assist designers track portions of the design that must be revalidated after a change. The system can log designer activity with their assistance, for example, to record the success or failure of a simulation run. Equivalence constraints are described by validation scripts that must match the actual log of design events to be valid. For example, verifying that a layout and transistor object are equivalent requires the invocation of a circuit extractor and a schematic comparison tool. These tools must be applied to the appropriate versions constrained to be equivalent.

We have based a simple Validation Subsystem on PROLOG. PROLOG provides an elaborate pattern matcher, in which the validation scripts, specified as PROLOG rules, are matched against the event log, stored as time-stamped PROLOG facts, to "prove" that the constraint is in force (see Figure 3.3).

PROLOG can be used to infer unvalidated equivalence relationships from those that have already been validated. It can also inform the designer about the nature of equivalence violations if there are any, by backtracking to the point of failure. Suppose that A and B are equivalent. A

rules:

```
equivalence (Layout, Transistor) :-
        extractor (Layout, T1),
        comparator(Transistor, T1, succeed).
```

facts:

```
extractor (layout_1, transistor_1).
extractor (layout_2, transistor_2).

comparator (transistor_3, transistor_1, succeed).
comparator (transistor_3, transistor_2, fail).
```

query:

```
equivalence (layout_1, transistor_3)?

        ** YES
```

Figure 3.3 -- Check-in Script and Proof of Consistency in PROLOG

Layouts are shown to be equivalent to transistor descriptions by executing a circuit extractor and schematic comparator. This sequence of events is specified in the Prolog rule, where the capitalized parameters are variables. The facts indicate which tool events are associated with which versions (lower case parameters), and whether the invocation succeeded or failed. To check that layout_1 and transistor_3 are equivalent, the rule is matched against the facts, and Prolog's inference mechanism can deduce that the rule is satisfied for the specified objects.

designer checks-out A to create a new version A'. By inheritance, A' must be shown to be equivalent to B before it can be checked back into the Archive. The designer can augment the database with a new equivalence relationship among A and A'. If this constraint is shown to be valid, then the original constraint is satisfied by transitivity: A is equivalent to B and A' is equivalent to A implies that A' is equivalent to B.

4. Version Server

We are incorporating the data model of Section 2 and the operations of Section 3 into a *Version Server* [KATZ86]. It (1) organizes the design into configurations across representations, (2) maintains version histories, (3) supports workspaces, in which designers can make private changes, (4) permits these changes to be shared with other designers through semi-public

workspaces, and (5) implements the careful update of the design archive, by insuring that objects added to it have been successfully validated and that all equivalence relationship are in force. By supporting the structural relationships of Section 2, without imposing constraints on the internal formats of the objects it manages, the Version Server should be able to manage information from many design domains.

The Version Server provides access to objects through check-out/check-in operations, which are nested within design transactions. Versions are checked-out to private workspaces, where a designer can create new derivatives. These can be shared, without first validating them, by checking them into a semi-public workspace associated with the design transaction. Designers checking out versions from these workspaces accept the risk that such objects may be incomplete or invalid. Otherwise, the objects can be checked back into the archive as new versions, but only if all constraints, equivalence and otherwise, can be shown to be valid. These mechanisms are more fully described in [BANC85, KATZ84b, KIM 84].

5. Previous Work

The Source Code Control System of UNIX [ROCH75] is perhaps the most widely known version management system. It uses differential file techniques to encode versioned text files [SEVE76]. Versions are named according to creation order, and it is possible to create parallel versions. A particular version is first extracted from the base file, and later appended to it as a new version.

While being widely used, SCCS does have shortcomings. There is no way to order the versions other than by creation time. Its mechanisms have been selected to support versions of a single file. It is not possible to associate versions of component modules with the module that incorporates them. Further, the system has no support for correspondences across files containing information of different types, for example, documentation and source code.

The UNIX MAKE facility provides a primitive mechanism for enforcing equivalence constraints. Object A is equivalent to (or dependent on) object B if there is an algorithmic

process by which A can be derived from B (e.g., A is the result of compiling B). The constraint is in force as long as A has a newer timestamp than B. If B has a newer timestamp when MAKE is invoked, then A is rederived from B by executing the appropriate script. The approach has been generalized for CAD databases in [NEUM83]. In our model, we explicitly keep track of versions, and do not use timestamps to identify invalid equivalences. Our equivalences are validated with respect to more general scripts, and are enforced passively. The Validation Subsystem flags unvalidated equivalences rather than forcing them to be valid. Many designers prefer a passive approach that they can override as desired.

Several version management schemes have been proposed within the database context [STON81, KATZ84a], but they do not adequately support configurations. Each assumes an underlying record-based system, while the object-oriented model presented in this paper is more general. Relationships over versioned relations could be stored as versioned relations, giving the flavor of configurations, but how to support concepts like dynamic configurations is not obvious.

Data compression is an important consideration for most version schemes. Configurations provide this capability at an object granularity. A component object can be shared among several composite objects, including multiple versions of the same composite object. This is effective if objects are of a reasonable size (hundreds of design primitives, not millions). It is still possible to compress within versions of the same generic object, but to do so requires some constraints on the internal formats of versions. For example, differential file techniques cannot easily be applied to bit-string data files. It also makes it expensive to reference configurations that incorporate old versions, since these must be decompressed on access.

There have been some recent proposals in which versions are modeled by the SmallTalk type/instance model [BATO85a,b]. Generic objects become *types*; version objects become *instances*. Instances can inherit attributes from their type. This is similar to the generalization hierarchies supported by most semantic data models. However, inheritance requires that the

system understand the internal structure of design objects[4], which we have tried to avoid. Further, these models do not make explicit provision for configurations or composite objects. An object is viewed as having several representation-dependent facets, which we represent through configurations of equivalences as in Figure 2.6b. It is not easy to support non-isomorphic representation hierarchies, as important requirement for some design domains such as VLSI, without explicit denotations of equivalences.

In our model, we have observed a close interrelationship among versions and configurations, but are not the first to do so. [MCLE83] introduced an AND-OR notation to represent that objects were configured from components (AND), which in turn were bound to one of several alternatives (OR). This idea was adapted from earlier work in version control of software systems. Our contribution is to separate these concepts into orthogonal, rather than sequential, relationships.

6. Conclusions

Even though version and configuration management systems have existed for some time, how best to structure such information remains an open question. We have introduced (1) Version Histories, (2) Configurations, and (3) Equivalences, as a framework for organizing design databases that evolve over time. Explicit configurations is key: it must be possible to specify an object version in terms of the versions of its components. We have defined these concepts so they can be imposed upon an existing collection of design files, created by existing design tools, and are attempting to do so in a new system called a *Version Server*.

We make no claim that the model proposed here is either optimal or exhaustive. We plan to experiment with real designers to see how they make use of a Version Server that supports this particular model of versions, configurations, and equivalences. If we have done nothing else, we hope to have convinced the reader that the representation issues of versions are subtle and worthy

[4] [AFSA85] is a good example of the incredible richness of modeling concepts with which the internal structure of a design could be described.

of further research.

We gratefully acknowledge the assistance Mohammed Anwarrudin, a visiting Industrial Fellow from Digital Equipment Corporation, in formulating the ideas described in this paper and in assisting with their implementation.

7. References

[AFSA85] Afsarmanesh, H., D. McLeod, D. Knapp, A. Parker, "An Extensible Object-Oriented Approach to Databases for VLSI/CAD," Proc. 11th Very Large Database Conference, Stockholm, Sweden, (August 1985).

[ARIA83] Ariav G., J. Clifford, M. Jarke, "Time and Databases," Proc. ACM SIGMOD Conference, San Jose, CA, (May 1983).

[BANC85] Bancilhon, F., W. Kim, H. Korth, "A Model for CAD Transactions," Proc. 11th Intl. Conf. on Very Large Databases, Stockholm, Sweden, (August 1985).

[BATO85a] Batory, D. S., W. Kim, "Support for Versions of VLSI CAD Objects," M.C.C. Working Paper, (March 1985).

[BATO85b] Batory, D. S., W. Kim, "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems*, V 10, N 3, (September 1985).

[GOLD81] Goldstein, I. P., D. G. Bobrow, "Layered Networks as a Tool for Software Development," Proceedings 7th International Conference on AI, (August 1981).

[HASK82] Haskin, R., R. Lorie, "On Extending the Functions of a Relational Database System," Proc. ACM SIGMOD Conference, Orlando, Fl., (June 1982).

[KATZ83] Katz, R. H., "Managing the Chip Design Database," *I.E.E.E. Computer Magazine,* V 16, N 12, (December 1983).

[KATZ84a] Katz, R. H., T. J. Lehman, "Database Support for Versions and Alternatives of Large Design Files," *I.E.E.E. Transactions on Software Engineering,* V SE-10, N 2, (March 1984).

[KATZ84b] Katz, R. H., S. Weiss, "Design Transaction Management," Proc. A.C.M./I.E.E.E. 21st Design Automation Conference, Alburquerque, N.M., (June 1984).

[KATZ85] Katz, R. H., M. Anwarrudin, E. Chang, "Organizing a Design Database Across Time," Islamorada Workshop on Large Scale Knowledge Bases and Inference Systems, Islamorada, FL, (February 1985).

[KATZ86] Katz, R. H., M. Anwarrudin, E. Chang, "A Version Server for Computer-Aided Design Data," submitted to 23rd ACM/IEEE Design Automation Conference, Las Vegas, NV, (June 1986).

[KIM 84] Kim, W., et. al., "Nested Transactions for Engineering Design Databases," Proc. Very Large Database Conference, Singapore, Malaysia, (August 1984).

[MCLE83] McLeod, D., K. V. Bapa Rao, K. Narayanaswamy, "An Approach to Information Management for CAD/VLSI Applications," Proc. ACM SIGMOD Conference, San Jose, CA, (May 1983).

[NEUM83] Neumann, T., "On Representing the Design Information in a Common Database," Proc. ACM SIGMOD Conference, San Jose, CA, (May 1983).

[ROTH75] Rochkind, M. J., "The Source Code Control System," *IEEE Trans. on Software Engineering,* V SE-1, N 12, (December 1975).

[SEVE76] Severence, D. G., G. M. Lohman, "Differential Files: Their Application to to the Maintenance of Large Databases," *ACM Trans. on Database Systems,* V 1, N 3, (September 1976).

[STON81] Stonebraker, M. R., "Hypothetical Databases As Views," Proc. ACM SIGMOD Conference, (May 1981).