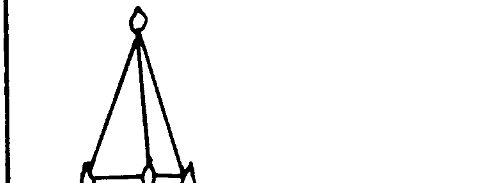
A Version Server for Computer-Aided Design Data

R. H. Katz, M. Anwaruddin and, E. Chang



Report No. UCB/CSD 86/266

November 1985

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

A VERSION SERVER FOR COMPUTER-AIDED DESIGN DATA[1]

R. H. Katz, M. Anwarrudin[2], *E. Chang*
Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT: A design database organizes the description of an artifact, by arranging it as a hierarchical composition of components across multiple representations. It is particularly difficult to manage this complex structure *as it evolves over time*. In this paper, we present a logical organization for describing designs across time. We also present an operational model, based on workspaces and transactions, that describes how these structures can be manipulated while controlling the sharing and integrity of the design database. These concepts are being implemented in a *Version Server* under development at the University of California, Berkeley.

Key Words and Phrases: Version and Configuration Control, Design and Object-Oriented Databases;

1. Introduction

VLSI design is becoming dominated by data management concerns. Correlating a design's physical implementation with its many representations used for simulation is one of the primary sources of complexity. The design is implemented by a set of process masks, but its behavior and performance are verified with register transfer, logic, switch, and transistor level descriptions. It is important to be able to find, across representations, equivalent descriptions of the same portion of the design. Maintaining these correlations is made even more difficult because they change over time.

The organization of the design is itself very complicated. First, it is viewed from several perspectives.[3] For example, the description of a processor as seen by the layout artist, circuit designer, and computer architect are all different, yet they must be correlated: the layout must correctly implement the circuit; the circuit must satisfy the functional specifications.

[1] Research supported by the National Science Foundation under Grant ECS-8403004 and a grant from the National Semiconductor Corporation.

[2] Work done while a Visiting Industrial Fellow with the U. C. Berkeley CAD/CAM Consortium. Current Address: Digital Equipment Corporation, 75 Reed Rd., Hudson, MA.

[3] Perspectives are sometimes called *views* in the CAD literature.

Second, designs are constructed hierarchically. Hierarchy is generally accepted as the most effective method for *reducing* the complexity of a design, by making it more intelligible to designers and easier to process by design tools. However, it *complicates* the design description by introducing considerable additional structure. The design database must now describe how composite objects are built from components.

Finally, the entire description, across representations and within hierarchies, must be maintained across time. Individual portions of the design are superceded by newer versions. To maintain the design history, new versions do not overwrite the existing description. At least some of these must be kept on-line, so design alternatives can be evaluated and reviewed.

In this paper, we describe a *Version Server*, providing version and configuration management functions for design teams. It (1) organizes the design into an archival hierarchical description across representations, (2) maintains versions of design portions, (3) supports workspaces, in which designers can make private and tentative changes, (4) permits these changes to be shared in a controlled way, and (5) implements the "careful" update of the archive (a new version of the design must be *validated* before it can be placed in the archive). The Version Server is independent of design domain: while imposing an organization on design components, it does not restrict their internal structure.

The rest of this paper is organized as follows. In the next section, we define some basic terms. Section 3 is a detailed description of the structural relationships among design data that are explicitly supported by the Version Server. In section 4, the workspace model and the mechanisms for sharing tentative design changes are described. We discuss the transaction model, in particular, the design validation mechanism, in section 5. Our status and conclusions are given in section 6.

2. Basic Terms

The Version Server maintains a database of *design objects* and certain structural relationships among them. Objects are nothing more than logical aggregates of design information.[4] The Version Server supports a small number of domain-independent structural relationships defined over design objects, while their internal representations are determined by design tools (see Figure 2.1). This separation of *representation* and *structuring* decisions is crucial: it enables the Version Server to remain independent of design domain.

Consider a microprocessor design. One object within this description is the datapath layout object. Its internal structure could be arranged in several formats: as an ASCII file, as a collection of tuples from different relations [BATO85b], or as a linked-list of records [KELL82]. In addition, it participates in structural relationships. For example, it is the *composition* of the ALU, register file, and shifter layout objects. There is also a relationship between it and its equivalent transistor description. *Equivalence* relationships are represented as special objects in the design database (see Figure 2.2).

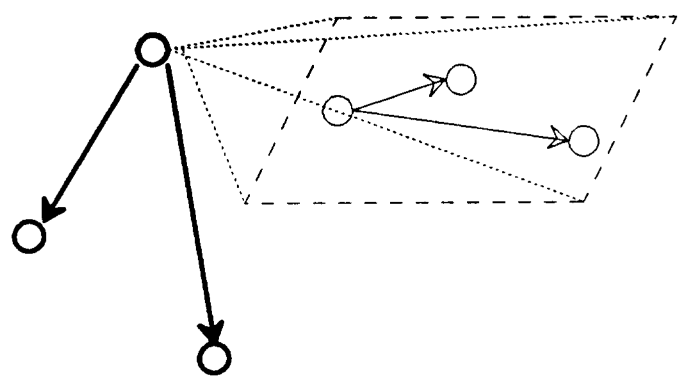


Figure 2.1 -- Coarse vs. Fine Grain Structure of Design Objects

The Version Server knows about the objects and their structural interrelationships (highlighted in darker lines). However, each of these objects may have their own finer grain internal structure, which may be formed hierarchically as well. The fine grain object structure is not known to the Version Server, to keep it free of representation details.

[4]At present, design objects are data-only. However, this does not rule out SmallTalk-style packages of data and manipulation operations in future implementations.

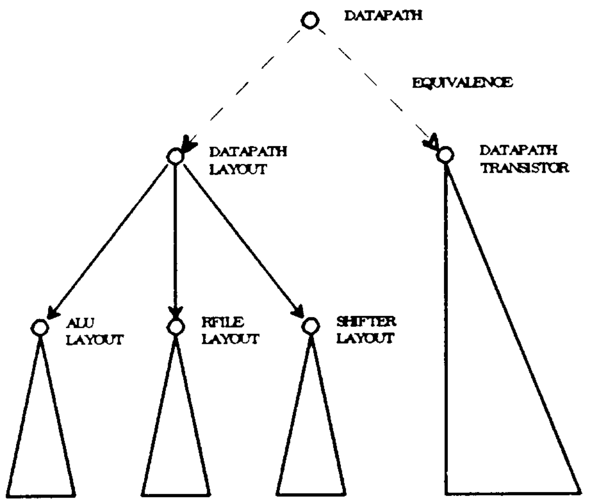


Figure 2.2 -- Hierarchical Design Description

The datapath object is decomposed into layout and transistor objects. The former is further decomposed into ALU, Register File, and Shifter layout objects. A datapath equivalence object constrains the layout and transistor representations to describe the same real world object -- the datapath.

We distinguish between a generic object, such as the "microprocessor datapath layout," and its particular *instances* over time, or *versions*. These versions contain specific collections of layout primitives. They are full fledged type-specific, or *representation*, objects. They are directly created by design tools. Generic objects, on the other hand, are not created by any particular design tool, but by the Version Server as a focus for grouping versions. In contrast to *representation objects,* objects introduced into the design database to represent structural relationships are called *structural objects*. These include both generic and equivalence objects.

The SmallTalk object/type mechanism has also been proposed for representing versions [BATO85b]. Generic objects become *types*, and instances/versions become *objects*. In the SmallTalk model, objects can inherit part of their description from their associated type, e.g., a portion of the interface description that is common across versions. Inheritance is not explicitly supported by the Version Server; to do so would require it to interpret objects' internal formats. However, this capability can be provided by an application program that traverses the object/version data structure and interprets the internal structure of objects, distinguishing between "type" and "object" objects.

Representation and structural objects are either *composite* or *primitive.* Primitive objects cannot be further decomposed into components (at least as far as the Version Server is concerned), while composite objects are composed of more primitive composite and primitive objects. Configurations are related to hierarchical compositions: they are *synchronized* collections of *versions* of hierarchically related objects. Thus, a datapath version, incorporating specific versions of its ALU, Register File, and Shifter components, forms a configuration for that portion of the design. Versions of the root of the design hierarchy represent different configurations of the complete design.

3. Three Structural Relationships

A design database must distinguish between *representational* and *structural* details. By "representational", we mean the specific choices of how to represent the details of the design, i.e., the description formats for layouts, schematics, logic gates, etc. They are determined by the design tools. Structural considerations lead to the choice of relationships among objects. The Version Server manipulates the structure of the design, while design tools create and manipulate its representational details.

The organizing principle behind the Version Server's data structures are three kinds of structural relationships: (1) version histories, (2) hierarchical compositions, and (3) equivalences. They impose an organization on the collection of objects that over time constitute the design.

3.1. The Version Plane

The version plane organizes the many versions an object has over its lifetime into a *version history.* While it is natural to arrange versions into a linear sequence arranged by creation time, this is not flexible enough. Because an object may have many simultaneously valid versions, i.e., *alternatives*, the history is a tree. When a version is created by modifying an existing version, we

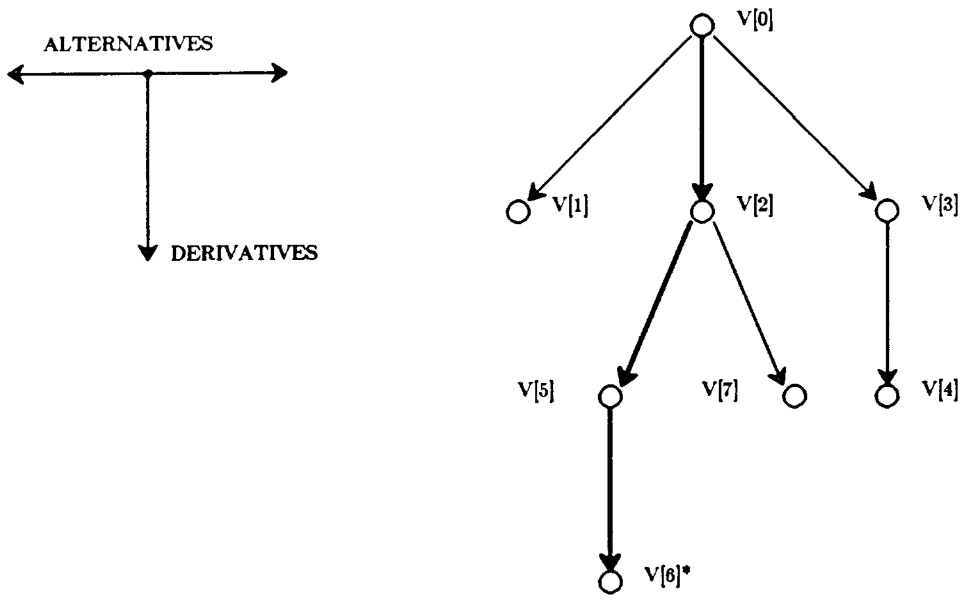


Figure 3.1 -- The Version Plane

A hierarchy is imposed on the versions of an object. The order of creation is indicated by the subscripts. Parallel versions are alternatives, while versions in series are derivatives. At any point in time, one leaf version is distinguished as "current" (marked by an asterisk). The path from the root to the current version is the "main derivation."

call it a *derivative* of the original.

Figure 3.1 shows the organization of the version plane for generic object V. The subscripts indicate the order of creation. Derivatives are arranged vertically, while alternatives are arranged horizontally. Every object must have an initial version, e.g., V[0]. Alternative versions V[1], V[2], and V[3] are derived from V[0].

Without some control mechanisms, version histories can branch widely. We need to identify the preferred or default version 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 descendants of the current version. Currency can be set explicitly to allow the design team to follow any desired system release policy.

To see how the version history of Figure 3.1 could have been derived, consider the sequence of events shown in Figure 3.2. V[0] is created and made the current version (see Figure 3.2a). 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 version (see Figure 3.2b), and no further derivatives of V[1], V[3], or V[4] can be created 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 3.2c). V[7]

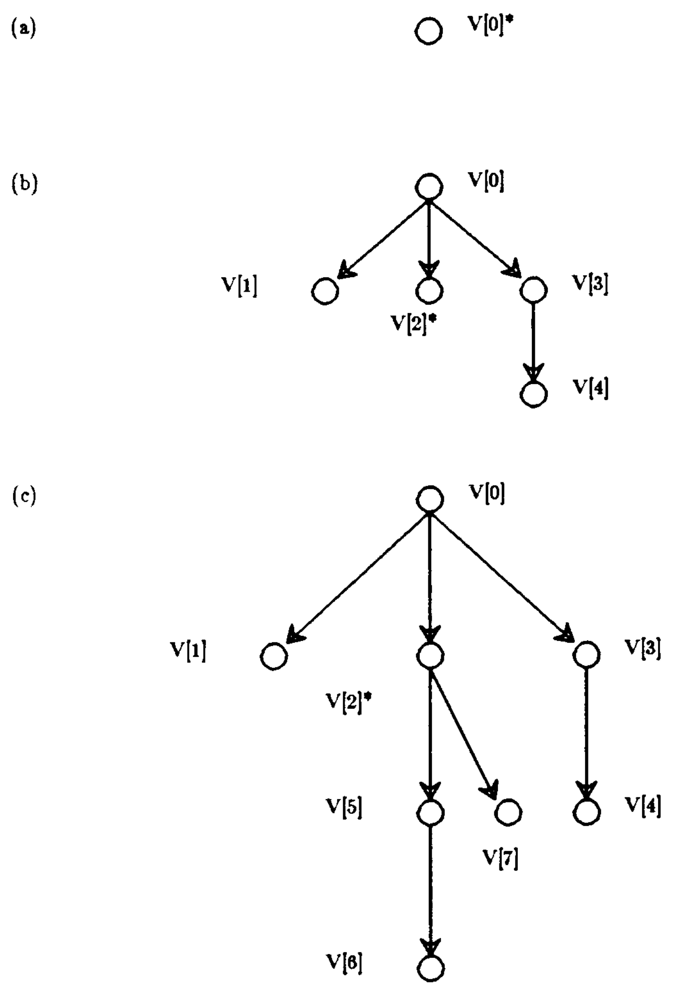


Figure 3.2 – Example Derivation of a Version History

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].

is created as a new alternative derived from V[2]. Now V[6] is made current, disallowing further derivations from V[2] or V[7], as shown in Figure 3.1.

3.2. The Configuration Plane

Figure 3.3 shows a portion of the configuration plane rooted at object instance U[i]. All nodes shown in the figure are representation objects. Configurations are formed by combining versions of different objects into composites. For example, V[j] is formed from the versions of W, X, and Y denoted by W[k], X[l], and Y[m]. Since composition information is associated with each composite object, a version of a composite object simultaneously defines its configuration.

Versions and configurations are orthogonal (see Figure 3.4): the configuration plane constructs objects as a hierarchy of components, while the version plane shows how instances of individual objects have evolved over time.

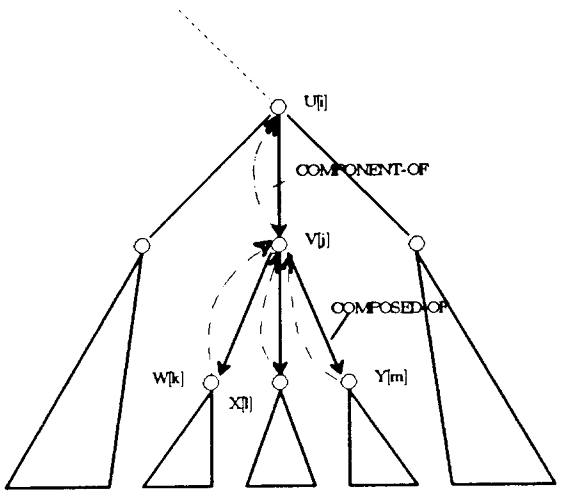


Figure 3.3 -- The Configuration Plane

Object V is composed of objects W, X, and Y, and is a component of U. The jth version of V is configured from the kth, lth, and mth versions of W, X, and Y respectively. The configuration plane, which organizes objects to form composite objects, is orthogonal to the version plane, which organizes the component instances of the same object.

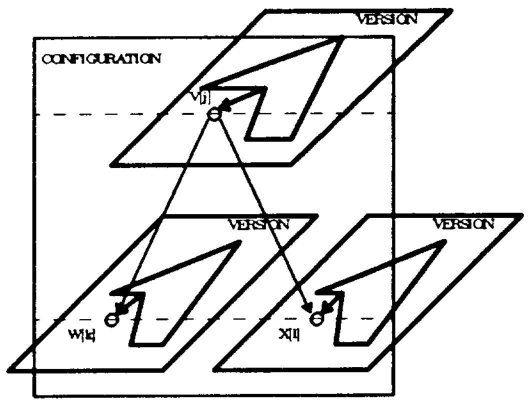


Figure 3.4 -- Configuration and Versions

The version and configuration planes are orthogonal. An object version is configured from components and is configured into composites. It is also a derivative of some other version in its own version plane. For example, object instance V[j] is configured from instances W[k] and X[l]. Each is also derived from some previously created object instance in its individual version plane.

At the time of its creation, the object instance V[j] is composed from instances of the objects W and X, but which instances? We defer, until section 4.2, a discussion of how instances are bound to configurations.

3.3. The Equivalence Plane

A real world object is described by several objects in the design database: one for each of its representations, arranged on their own version planes and participating in their own configurations. The equivalence plane ties together equivalent object instances across configurations in alternative representations.[5]

Figure 3.5 shows how the equivalence among the objects V', V", and V'" is represented. Their instances, V'[i], V"[j], and V'"[k], are joined together through an equivalence plane associated with the structural object **V**.

[5]Equivalences can also be defined over objects of the same representation.

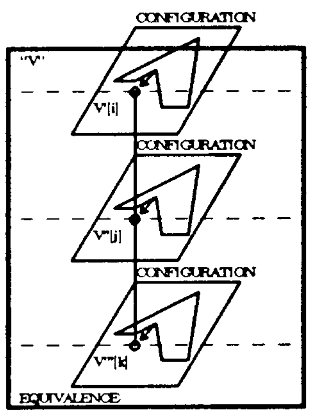


Figure 3.5 – The Equivalence Plane

Configurations in different design representations are correlated through the equivalence plane. In this figure, three different representations of the object V are grouped together by an equivalence object.

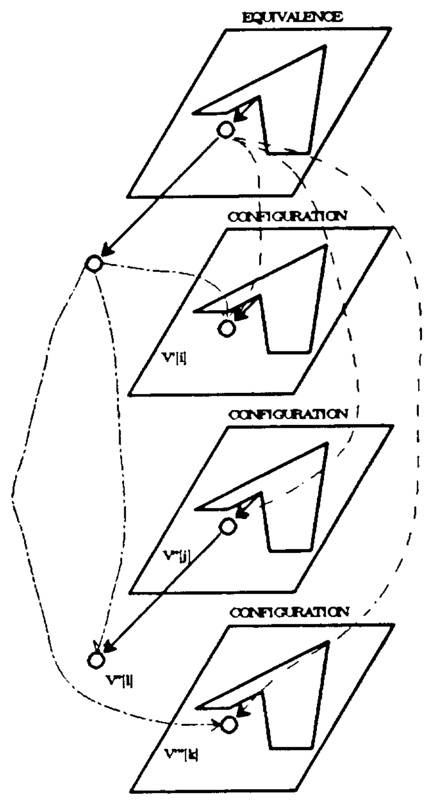


Figure 3.6 – Equivalences as Structural Objects

An equivalence object ties together V'[i], V''[j], and V'''[k]. A new version of V'', V''[l], is created, and associated with V'[i] and V'''[k] through a new equivalence object version.

Equivalence objects are composites with versions and associated version histories (see Figure 3.6). The semantic distinction among equivalences and configurations can be exploited to good advantage in assigning design objects to disk. Design object clustering can be based on configuration (an object and its components of the same representation stored near each other on disk), although clustering by equivalence may be desired for certain applications (an object and its equivalents in different representations stored near each other on disk).

Equivalence objects represent constraints on the database. They must be *validated*, i.e., shown to be in force, before the database can be *consistent* (see section 5.2). Consider what happens when a new derivative is first created. Since the database starts out being consistent, we assume that all existing equivalence constraints are already in force. The new version will inherit the equivalence relationships of its parent in the version history. The Version Server automatically introduces new equivalence objects to represent these. During design, these new equivalences must be validated to show that the database is still consistent. The designer can add new constraints or override inherited ones at his option. Additional constraints appear as new equivalence objects, while equivalence objects associated with overriden constraints are deleted.

4. Workspace Model: Controlled Evolution of the Design Database

In the previous section, we described how to organize design objects. In this section, we describe the Version Server operations that allow the design team to make changes to their data in a controlled way. The project database is viewed as an *archive* of design objects. It can be read by any member of the design team. However, new versions are added without overwriting existing objects. The Version Server allows designers to create *workspaces,* and provides *check-out/check-in* operations for moving copies of objects between workspaces.

4.1. Archive, Private, and Semi-Public Workspaces

Workspaces are named collections of object instances. There are three kinds of workspaces: *archive, private,* and *semi-public.* The archival workspace, or *Archive Space,* exclusively contains validated object instances, arranged into version, configuration, and equivalence planes as described in section 3. The Version Server provides mechanisms for selectively migrating old instances off-line. There is usually one Archive Space per design project. In addition, there can be any number of design libraries.

Private Workspaces are created for individual users by the Version Server, and are accessible only by that user. Objects are brought into private workspaces from an associated archive space (or semi-public workspace, see below) through Version Server check-out (see Figure 4.1a). When an instance is returned from a private workspace as a new validated version, it is added as a derivative of the originally checked-out instance (see Figure 4.1c). Configuration and equivalence relationships are implicitly inherited at check-out time. New configuration and equivalence

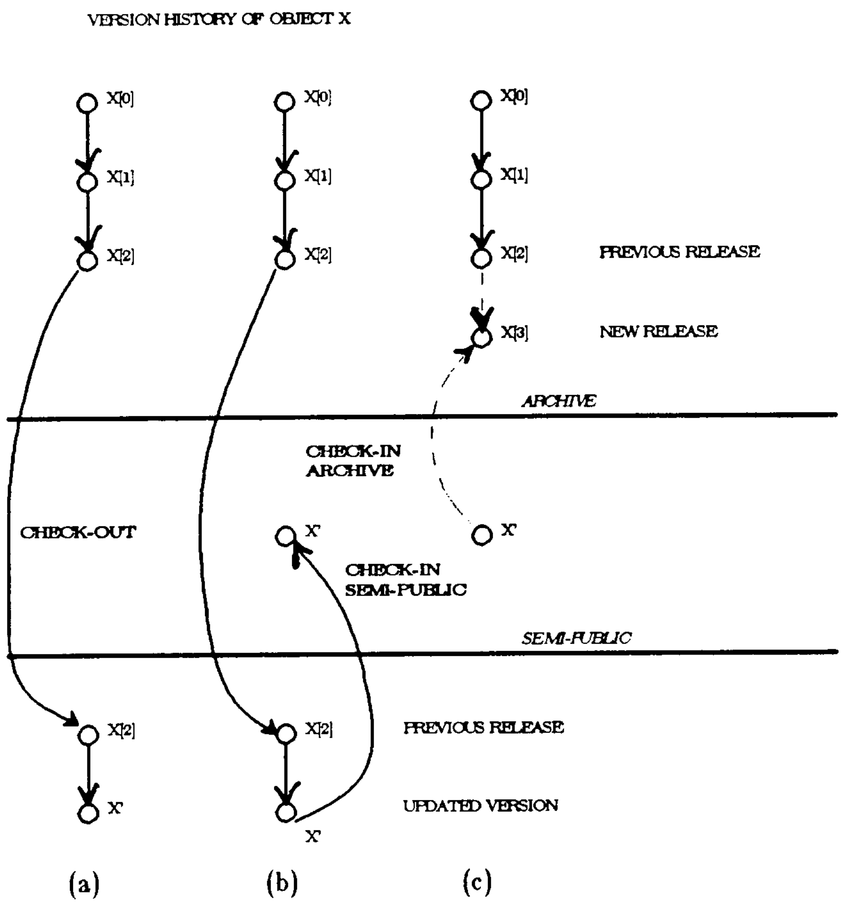


Figure 4.1 -- Workspace Model

An object version, X[2], is **checked-out** from the *Archive* to a *private workspace* [a]. Changes are made to create a new version X', which is **frozen** and then moved to a *semi-public workspace* for integration with the changes made by other designers. Once the assembled pieces of the design have been verified, the new versions are **committed** back into the *Archive*.

relationships can be created in the workspace, but these must be validated by check-in time to be recorded in the Archive Space.

Since it is only possible to place validated objects in the Archive Space, *Semi-public workspaces* provide a mechanism through which designers can share incomplete or partially verified objects. A designer can selectively check-in an object into a semi-public workspace (see Figure 4.1b). Other designers, with access rights to this workspace, can either read these objects or check them out if they wish to make further changes. Semi-public workspaces are associated with design transactions (see section 5).

4.2. Dynamic Configurations 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. Once a new version is committed to the archive, its configurations must be bound to specific versions.

Layers [GOLD81] support dynamic configuration binding. The Version Server provides operations to partition the database 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.[6]

The power of layering is that the designer determines which versions will be bound simply by *specifying the layer search order.* This choice of ordering is an *environment.* The Version Server allows layers to span any kind of workspace accessible to the defining user, and supports the grouping of layers into environments. There can be many user-defined environments for each database. All object accesses are evaluated with respect to a specified (perhaps default) environment.

As an example, consider the creation of layers as shown in Figure 4.2. 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

[6]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]).

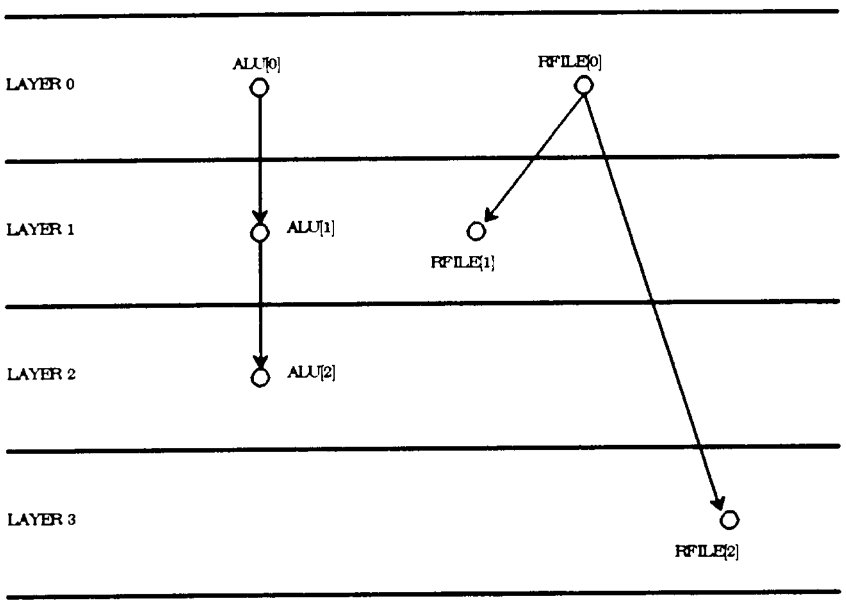


Figure 4.2 -- Layers and Environments Example

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].

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.

5. Transactions

Implicit in the Workspace Model are the permitted kinds of sharing: all design team members have access to the Archive Space; only an individual designer can access his private workspaces; semi-public workspaces allow limited sharing among designers. Transactions constrain how objects can be moved among workspaces. For example, only a fully validated object can be placed into the Archive Space. Check-out/check-in operations and semi-public workspaces are associated with particular design transactions.

5.1. Nested Subtransaction Model

A transaction is a packaged sequence of actions that map a state of the database into a new consistent state. A design transaction, associated with a single designer, corresponds to the sequence of design object check-outs, tool invocations, and object check-ins that together lead to the creation of new versions of design objects. It is well known that conventional database transactions do not model design database interactions (see [KATZ83]).

Nested subtransactions [KIM 84, KATZ84b] provide a better model by allowing partially completed versions to be shared among a group of designers. A project manager can create a *master* transaction to cover objects potentially shared by his design group. Individual designers may check-out their objects directly, circumventing the master, or may attach to its semi-public workspace, thus becoming a subtransaction. In the former, any derivatives they create cannot be shared until they are validated (see next subsection). Otherwise, derivatives can be checked-in to the master's semi-public workspace without verification, and can then be checked-out by other subtransactions. By joining a master transaction, the designer agrees to accept objects that may not have been fully validated.

5.2. Validation

The Validation Subsystem assists designers track portions of the design that must be revalidated after a change. It logs designer activity, either automatically, such as during check-out actions, or with their assistance, for example, to record the success or failure of a simulation run. Most design constraints are validated through successful execution of simulation tools, but some constraints can only be validated through a complex sequence of validation programs. These constraints are described by validation scripts that are matched against the actual log of design events.

Equivalence constraints are the most complex to validate. 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 of the objects being returned to the archive. The equivalence object associated with the constraint may need to identify the input test data set under which the objects are to be compared for equivalence, as well as any "technology" parameters that must be provided to the programs that perform the comparisons.

We have implemented a simple Validation Subsystem in PROLOG. The PROLOG system is used as 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 for the returned objects (see Figure 5.1).

We are investigating how to use PROLOG to infer unvalidated equivalence relationships from those that have been validated. Suppose that A and B are equivalent. A 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.

6. Implementation Status and Conclusions

We are implementing the Version Server system described above in a network environment of Digital Equipment Corporation VAX-11 computers and SUN Microsystems workstations. Figure 6.1 lists a command summary. Extensive use has been made of the interprocess communications and networking primitives available in Berkeley 4.2 BSD UNIX. The object system is built directly on top of UNIX files, but we hope to be able to make use of an Object Data Manager being developed by Professor Richard Newton and his students when it becomes available. A Version Server shell has been developed to automatically map generic object names *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 5.1 – 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.

vsshell/vsdone
    Enter/leave the Version Server command interpreter.

```
checkin       <object-name> <environment-name>
checkout      <object-name> <layer-name>
view          <object-name>
return        <object-name>
```
    Check-in/out objects from/to the identified Archive or Semi-public Workspace. Read-only objects can be viewed and returned without validation.

```
startxact/endxact     <workspace-name>
viewxact
```
    Begin or end a design transaction. View validation status.

```
define-layer                  <layer-name> <object-name>
define-environment            <environment-name> <layer-name>
make-known/make-unknown       <semi-public-name> <user-name>
```
    Place objects in layers. Place layers in environments.
    Make environments available to other designers.

```
register/semi-register   <workspace-name>
```
    Attach current transaction to a new Archive or Semi-public Workspace.
    In the latter case, the transaction becomes a subtransaction.

```
set-currency                           <object-name> <versionID>
store-version/restore-version     <object-name>
```
    Set current version of version history.
    Archive or restore versions older than the current version.

Figure 6.1 -- Version Server Command Summary

into their appropriate versions, given an environment specification. We are experimenting with how layers/environments and workspaces can be used by design teams to more effectively share their data. We are also considering how to build a graphical browsing capability on top of the design database.

Our work focuses on the importance of separating structural considerations (i.e., the organization of the design database into versions, configurations, and equivalences), from representation details (i.e., how to store layout geometries). The Version Server provides an "arms-length" database capability -- the applications and the Version Server are not closely coupled. Integrating design applications and databases, perhaps at the level of in-memory database structures, will require further work.

7. References

[BATO85b] Batory, D. S., W. Kim, "Modeling Concepts for VLSI CAD Objects," ACM SIGMOD Conference, Austin, TX, (May 1985)

[ECKL84] Ecklund, E. F., Jr., D. M. Price, "Multiple Version Management of Hypothetical Databases," Oregan State Technical Report, 1984.

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

[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, Albuquerque, 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).

[KELL82] Keller, K., A. R. Newton, S. Ellis, "A Symbolic Design System for Integrated Circuits," Proc. ACM/IEEE 19th Design Automation Conference, Las Vegas, NV, (June 1982).

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