MANAGING THE CHIP DESIGN DATABASE

Randy H. Katz

Computer Sciences Technical Report #506

May 1983

Managing the Chip Design Database

Randy H. Katz
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

ABSTRACT: Many organizations are obtaining a VLSI design capability by acquiring design tools from diverse sources. A design environment constructed in this way rarely forms a coherent design system. An integrated *design database* is a necessary component of an integrated design system. Unfortunately, conventional database systems do not completely satisfy a design system's data management needs. A *design data management system* bridges the gap between database systems and design tools. We describe the structure and function of such a system. Pieces of the system are under development at the University of Wisconsin-Madison.

1. Introduction

The Mead-Conway design method [MEAD80] has made it possible for computer scientists to experiment with custom integrated circuits as a new implementation medium for their algorithms. These designers not only build circuits; they also build design tools. A new generation of tools has been developed for circuit layout, synthesis, and verification. These include circuit editors (e.g., CAESAR [OUST81]), data-path generators (e.g., MACPITTS [SUSK82]), design rule checkers and switch level simulators (e.g., ESIM and NL/RNL [BAKE80]), and timing analyzers (e.g., TV and Crystal [JOUP83, OUST83]).

These are tools, not *systems*. Tools created by different groups are extremely difficult to integrate (e.g., see [KATZ83a]). Cumbersome translation programs map between the idiosyncratic representations understood by individual tools. Even standard interchange languages are not a solution, since each tool uses its own "escape hatches" for describing certain critical design information.[1] How to integrate the tools into a coherent system is still an open research issue.

---

[1] As an example, the Caltech Interchange Format [MEAD80] has no standard mechanism to associate symbolic names with geometric features of the layout.

An integrated design database is a prerequisite for transforming a loose ensemble of design tools into a design system (see Figure 1.1). The database organizes information about a VLSI circuit design across its representations, alternative implementations, and evolutionary versions. The system controls concurrent designer access and guarantees that the data can survive system crashes. By placing all design information under the responsibility of a single data management system, the self-consistency of the design is more easily maintained. Dependencies among parts of the design are made explicit. Thus, ramifications of design changes can be discovered, and can be propagated in a controlled manner. It becomes possible to economically verify that all representations of the design remain equivalent after a change.

Design databases have long been of interest to the design automation community [EAST81, LOSL75, LOSL80], but have been largely ignored within the context of the new tools. Since the implementation of a complete design system is a major undertaking, it is not surprising that efforts have focused on isolated aspects. Our purpose is to describe what design data management is and how it can integrate design tools. The paper is organized as follows. In the next section, we define basic
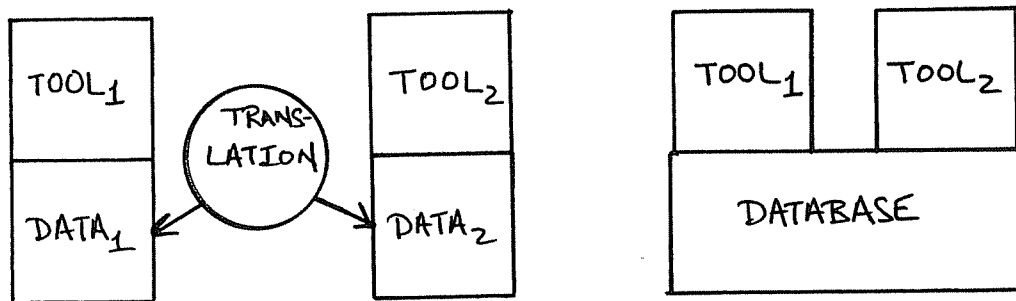

Figure 1.1 -- Design Tools vs. Design Systems

terms, such as database, database system, and design system. We assume that the reader has some familiarity with concepts of VLSI design, at least at the level of [MEAD80]. In section 3, we describe the unique data management requirements of design systems. Conventional database facilities and their shortcomings for supporting design data are described in section 4. The architecture of a prototype design management system is presented in section 5. Designs are organized into a richly interconnected data structure, based on an *object data model*. The structure of the design storage component, and recovery, concurrency control, design validation, and browser subsystems are described. Section 6 contains our summary and conclusions.

2. Basic Terms

Confusion exists over the terminology used by the design automation and database system communities. The terms we use throughout the paper are described in this section.

Data structures are logical organizations of data. Database systems typically support either (1) tabular (relational), (2) tree (hierarchical), or (3) graph (network) structures among records. The UNIX file system [THOM78], for example, supports a constrained graph structure among directories (internal vertices) and files (leaves). A link mechanism allows files to be incorporated in more than one directory. *Storage structures* are implementations of data structures on secondary storage. While a tree of design objects is a data structure, its implementation as a balanced multi-way tree on disk is a storage structure.

A *database* is the data describing the activities of an organization. For example, the chip design database contains information about how design objects are composed from primitives (geometries, transistors, gates, etc.), how a design is described in different representations (layout, circuit, logic, etc.), how a design has evolved over time, who is responsible for designing its parts, and so forth. A particular choice of data structures to represent a specific database is a *database schema*.

A *database management system* manages databases stored on secondary storage. It provides a standard interface to data for application programs. The data access operations supported by modern systems are based on data structure rather than storage structure, so the physical structure can be changed without affecting existing programs ("data independence"). The system controls access to protect the data from illegal actions and to maintain its consistency. It provides mechanisms to insure that changes can be recovered after a system crash. *Access methods* are storage structure specific routines for manipulating data on disk. The distinction between file systems and database systems is subtle. Perhaps most significant is that a database system supports atomic actions that span multiple files.

A *design data management system* chooses how to structure the design data within the database system. It provides a standard access interface for tools. While the database system does not interpret the data it manages, a design management system understands how the *structure* of the data describes a design project. It enforces design data constraints. For example, part of the design data structure identifies equivalent objects across representations. Such information can help reduce the effort needed to keep the database self-consistent after a design change. The design data and the complex consistency constraints are normally what the design automation community mean by "database."

The major components of design data management include (1) the storage component, reliably storing design data on disk, (2) the recovery subsystem, saving incremental changes and insuring resiliency to crashes either at workstations or database servers, (3) the Design Librarian, supporting check-in/check-out of design parts from the database, (4) the validation component, checking that design constraints remain in force after a change, (5) the design transaction component, using (2), (3), and (4) to control the creation of new versions of design objects, and (6) the Browser/Chip Assembler, providing an interactive frontend for creating and viewing the design data structure.

A *design system* is the marriage of design tools, project management aids, and design data management facilities. The design tools create pieces of a design and validate its correctness. The project management aids assist in planning the implementation effort. Design data management is responsible for structuring the design, and for exploiting the structure to keep the design consistent.

3. What Design Systems Need

Design applications need to (1) structure design data hierarchically, (2) support multiple design representations, (3) maintain design versions and alternatives, (4) help the designers cooperate and interact as a team, (5) support remote design at workstations, with long term storage at database servers, and (6) maintain the consistency of the design. Each of these are described in this section.

Hierarchy and regularity are well-known tactics for reducing the complexity of a large design [NEWT81, SEQU83]. A design proceeds by a top-down decomposition of systems into subsystems and a bottom-up synthesis of building blocks from more primitive building blocks. The design is complete when all subsystems can be implemented by existing building blocks and primitives. The hierarchical structure of the design is mirrored in its data.

Design regularity reduces the size of the design database as well as the design effort. An object is designed once and used frequently. Its representational description appears once in the database. Additional data describes each usage of the object, specifying how it is instantiated and placed within the design.

VLSI circuits are described in several representations. A non-exhaustive list includes geometric layouts, transistor networks, logic schematics, and functional descriptions. Each representation is appropriate for a different phase of design. Geometries are used for mask making and geometric design rule checking. Transistor, logic, and functional descriptions are used for electrical rules checking, simulation, and timing verification at various levels of detail.

While the Mead-Conway style encourages a horizontal partitioning of a design -- a "tall thin man" is responsible for deriving all the representations of his portion of the design -- industrial designs are more typically partitioned into architectural, logic, circuit, and layout activities. Different groups may be responsible for the detailed design within each representation. Mechanisms are needed to insure that all representations describe the same object. Equivalent portions of the design across its representations must be explicitly identified.

Much of design activity is evolutionary or exploratory. The database must therefore support design versions and alternatives. A version is an improvement or correction to a design object. Alternatives are different implementations of the same object, with varying performance characteristics. Versions provide insights into design approaches and rationale. They are needed to document designs in the field and for legal reasons. Alternatives, especially within design libraries, enable designers to experiment with different implementations of the same function.

VLSI complexity circuits are normally designed by teams. While the advent of silicon compilers and better synthesis tools may eventually eliminate the need for team design, they will remain the primary organization for building VLSI circuits for some time to come. To help them cooperate harmoniously, design teams need aids such as check-out mechanisms, interface descriptions, and validation audit trails. Check-out mechanisms insure that at most one designer is modifying an object at a time. The design data is made self-documenting by placing interface descriptions, dependency information, and design responsibility in the database. This helps a designer understand *how* an object is used, *where* it is used, and *who* is responsible for its design. The design validation subsystem assists designers in keeping the design consistent. It identifies the portions of the design affected by changes and assists in propagating these throughout the design data structure. It maintains an audit trail so responsibility can be assigned for incomplete or faulty validation.

With the emergence of powerful engineering design workstations, design management must be provided in a distributed system of workstations and database servers. The database servers are shared repositories of design data. Designs are checked out to designers at workstations. Making the data resilient to system crashes is complicated by the lack of archival media at the workstations. Valuable design work can be lost unless efforts are made to frequently and automatically save changes on the servers. To keep the design database consistent, modified data can be checked back into the repository only when it has passed a lengthy validation process.

An embarrassingly frequent error is the incomplete propagation of a design change to all parts of the design that depend on it. This is a serious problem in design environments without integrated design management. An integrated design database must incorporate information about the dependencies across representations and design partitions.

4. Why Commercial Databases are NOT like Design Databases

Database facilities have evolved to support both high performance transaction processing and interactive use by non-programmers (an excellent description of commercial database system technology can be found in [DATE81, DATE82]). These include (1) structures for efficient access to data on secondary storage, (2) the concept of a transaction: collections of read and update operations treated as atomic units of database consistency, (3) protection and concurrency control mechanisms for controlled data sharing, (4) automatic integrity maintenance, (5) crash recovery services, and (6) user-friendly interfaces (graphical, natural language) and high-level query languages.

Not all of these services are valuable for design systems. For example, only very simple integrity constraints can be enforced automatically by the database system. Constraints on design data are not easy to specify or enforce. Compare the

constraint "salary must be greater than zero" with "the circuit must behave as speci-fied with expected performance." While fast access to records is important, the overhead of entering and leaving the database system to extract a record at a time is too great for the large quantities of data involved. A conventional database is best used as a shared, reliable repository. Large aggregates of design data are extracted and replaced as a unit. Data sharing is controlled by not allowing more than one designer to update the same portion of the design at the same time. Recov-erable update and archival mechanisms insure that data is never lost because of a system crash.

Access to a conventional database is through transactions [GRAY81]. Transac-tions are sequences of read and write actions that leave the database consistent: interleaved transaction executions are permitted as long as their result is the same as some serial execution of the transactions. Transactions are atomic: all changes become visible at once -- the transaction commits -- or none become visible -- the transaction aborts. Transactions are durable: once a transaction has committed, its changes are permanently installed in the database even if the data should be tem-porarily lost because of system failures.

Conventional transactions have been developed for the short duration, simple units of work typically found in transaction processing environments, such as airlines reservations. Unfortunately, they do not adequately model design interactions. A *design transaction* begins with a designer acquiring exclusive access to a portion of the design, modifying it over a long period of time, and committing the changes only when they are shown to be valid [LORI83, KATZ83c].

Many database implementation issues are actually simpler in the design environ-ment. Because of design teams and their strict partitioning of tasks, designer interference is rare. While many sophisticated techniques have been proposed for controlling concurrent access, for design applications simple techniques are suffi-

cient to resolve most conflicts. For example, negotiation among designers who need the same data is usually enough.

Design transactions are not atomic. To recover from a crash, database systems undo the effects of incomplete transactions and redo the effects of completed transactions. If the duration of a transaction is from the time an object is checked out from the repository to when it is replaced, then the system should never roll back the uncommitted design changes. Hours (or days) of work would be lost! The database should be restored to the most recent state possible, even past the last state "saved" by the designer.

While the effects of committed database transactions survive system crashes, design data lives beyond subsequent transactions. Design transactions create new versions of the objects they update. Conventional database systems provide no support for versions, even though this greatly simplifies the implementation of concurrency control and recovery mechanisms (as we shall see below).

Database systems do not adequately handle the basic storage needs of design data. They have been tuned for large volumes of regularly structured data. Design data is organized into complex structures, with large numbers of interconnected files of relatively small size. It is not easily formatted for storage in a conventional database. Few systems adequately support the kinds of variable length heterogeneous data typically created by design tools (see [HASK82] for ways in which database systems have been extended to handle this kind of data).

As can be seen from the above, there is an applications mismatch between what design tools need and what database systems provide. In the remainder of this paper, we will describe the structure of a system that can satisfy the data management needs of design tools.

5. A Design Data Management System Architecture

5.1. System Overview

Design systems are large, complex software packages, providing a number of services necessary for the design of integrated circuits. Most designers view the design tools as the "system." However, many services are duplicated from tool to tool. If these can be identified, and implemented as standalone subsystems, future tools could be built much more easily. For example, menu and window packages are now available as standalone subsystems. This is what we are attempting to do for design data management.

The system architecture appears in Figure 5.1. The *storage component* stores design data on disk. Updates are guaranteed to be atomic. A conventional database system can be used as a storage component [HASK82, KATZ82a], although a suitably

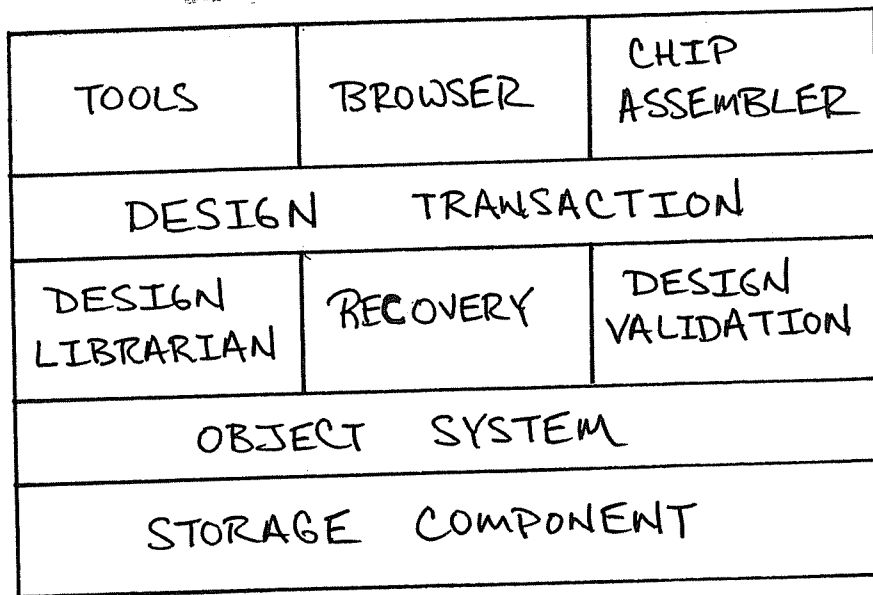| TOOLS	BROWSER	CHIP ASSEMBLER
DESIGN TRANSACTION		
DESIGN LIBRARIAN	RECOVERY	DESIGN VALIDATION
OBJECT SYSTEM		
STORAGE COMPONENT		

Figure 5.1 -- System Architecture

extended file system can also be used.

The *object system* maps the design data, viewed as a collection of interrelated objects, into the files and structures supported by the storage component. A reliable object-oriented file system, such as [REED83], can provide the facilities of both the storage component and the object system simultaneously. To support existing design tools, the object system can appear as a conventional file system.

The *recovery subsystem* collects incremental changes to objects to insure that it can reconstruct those that are lost in a system crash. To protect workstations from data loss, changes are continuously spooled to the server. Savepoints invoked by the designer force in-progress changes to be saved, thus guaranteeing that the object can be reconstructed to that point.

The *Design Librarian* is responsible for controlling shared access to design objects. Many designers can browse an object, but only one is allowed to create a new version of it at a time. The Design Librarian maintains information about who currently holds objects and when these are expected to be returned to the design repository. This enables the recovery subsystem to reconstruct the workstation's disk after a hard crash.

The *design validation subsystem* interpretes dependencies among design data to identify what portions are potentially affected by a change. Some simple consistency checking can be handled automatically, particularly for representations understood by the subsystem. More complicated validation, e.g., verifying equivalence across representations, requires designer intervention. An audit trail is kept of what validation activities have been performed by the designers.

The *design transaction* component encompasses the recovery, Design Librarian, and design validation subsystems. It insures that designers create new consistent versions of design data with their tools. Designers acquire access rights to the appropriate portions from the Design Librarian. Successfully acquired objects are

then transferred to the workstation. The recovery subsystem guarantees that all but perhaps the most recent changes can survive system crashes. The validation subsystem determines what is affected by the changes, identifying what must be reverified. A design transaction cannot complete successfully until the design is once again consistent.

Design transactions bring design data to the workstation and return it as new versions when done. *Design tools* manipulate the data through operations supported by the object system at the workstations.

The *Browser/Chip Assembler* is the interactive interface to the design data management system. Designers use the Browser to view the complex data structure describing the design. They use the Chip Assembler to manually construct the data structure from the pieces created by individual design tools.

In the next subsection, we describe the data structures for specifying a design. Then we will examine each subsystem of design data management in greater detail.

5.2. Object Model

The Object Model defines the basic primitives from which the design data structure is formed. [KATZ82a, LORI83, MCLE83] are some of the alternative proposals for structuring a design for storage in a database. Similar themes are apparent: object- rather than record-orientation, explicit representations of versions and alternatives, and support for interface descriptions. Our motivation is to make the design as self-describing as possible, to enhance the system's ability to keep it consistent.

5.2.1. Objects

Design objects are convenient aggregations of design information. They fall into two broad categories: *representation objects* and *index objects*. Representation objects describe a portion of the design in one of its representations, and thus are of a particular type, e.g., a layout object, a transistor object, or a logical object. Most

information about the design is organized through representation objects. Index objects introduce auxiliary structure for use by the browsing, configuration, and validation tools.

Each representation object is constructed from the composition of its type's representational primitives (e.g., geometries, transistors, gates) and other objects of the same type. We call the hierarchical collection of design data thus formed a *representation hierarchy*. A *design hierarchy* is the collection of these describing a full design, with additional structures linking together the representations and providing alternative groupings based on versions (configurations), alternative implementations, or common attributes. Besides composition information, representation objects have *interface descriptions*, describing their abstract behavior, usage information, and associated performance (speed, power, area). Design transactions do not overwrite existing representation objects, but create new versions of them. They are identified by their name extended with a unique version number.

Index objects provide a way to group objects together outside of the normal decompositions within representations (they can be used to group together objects of the same type as well). The Browser uses them to quickly find objects with similar attributes. For example, all ALU objects within a library may be grouped together by an index object. Indices can be composed in much the same way as representation objects. For example, the index of ALU objects can be incorporated within an index consisting of all datapath pieces. Unlike representation objects, index objects do not have interfaces or versions. A number of special index object types will be introduced below.

The hierarchies can be thought of as directed acyclic graphes. Vertices represent objects. Leaves are *primitive* objects, while internal vertices are *composite* objects. Designers determine what entities are primitives. Although individual geometries or transistors could be primitives, a primitive object is more likely to be

some simple function implemented with a small collection of these. A composite object is formed from the recursive composition of its descendants in the graph. Edges in the graph are directed from *composite* (parent) objects to *component* (child) objects.

Associated with each edge is how to create an instance of the component object. Sometimes we must fully (or partially) instantiate design objects, creating a tree rather than a DAG. For example, when simulating a design, each instance is different because each has different associated state variables. Only information that is unique to an object's instance needs to be represented in the tree (see Figure 5.2).

While an object is defined by the composition of its components, it can be modified independently of them, and vice versa. A new version of the composite object can be formed from new components. The creation of new versions of the components does not affect their original composite. It can continue to reference their original versions. If the composite object's designer wishes to take advantage of
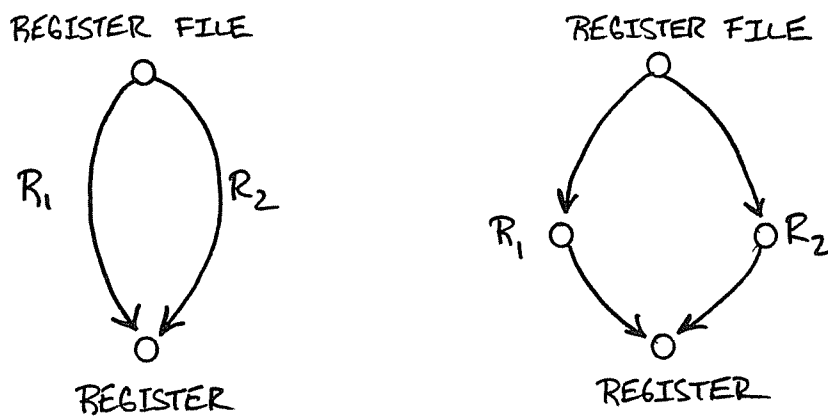

Figure 5.2 -- Instantiated Objects

the new versions, he must create a new version of the composite that explicitly includes them (see Figure 5.3).

Objects can either be *independent* or *dependent*. Independent objects exist within the database whether or not they are contained within other objects. Dependent objects are deleted from the database whenever they are no longer referenced.

Representation hierarchies need not have identical decompositions (see [MUDG81, BEET81] for alternative approaches in which all representations must be isomorphic). The functional decomposition of a design may be quite different from its physical decomposition, for example, if it is implemented in a gate array technology[2] (see Figure 5.4). The additional structure needed to identify equivalent objects across representations is furnished by a special index object type, an *equivalency* object. These tie together objects in different representation hierarchies, constraining them to be equivalent. For example, an ALU layout object and an ALU transistor object may be linked by an equivalency object to denote they are different representations of the same ALU. The constraints are enforced by the design valida-
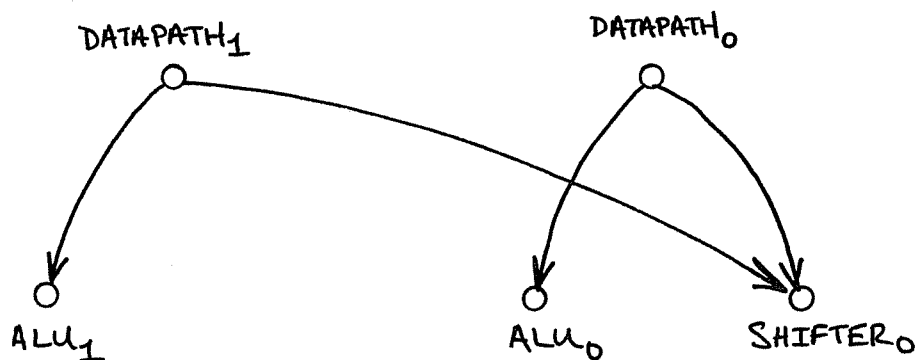

Figure 5.3 -- A New Datapath Object with a New ALU

_____

[2] In a gate array, a function can be implemented with primitive building blocks that may be distributed throughout the chip. One cannot look at the layout and easily determine what portion implements a particular function like the ALU.
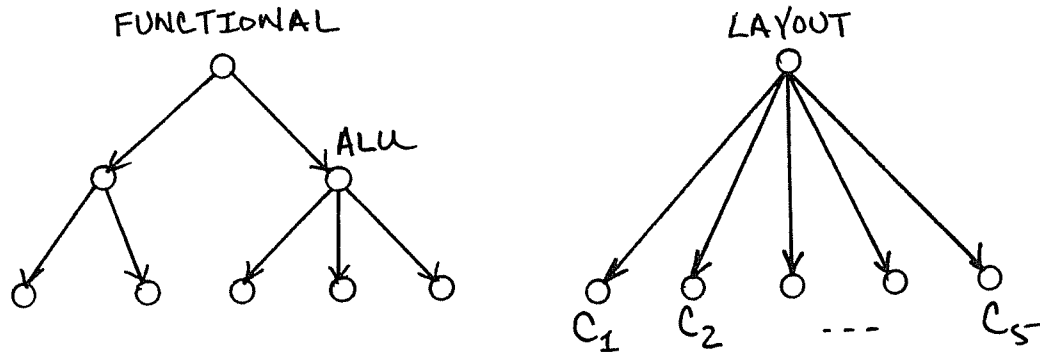
Figure 5.4 -- Non-isomorphic Representations

tion component. Equivalency objects are removed from the database when one of their referenced objects is removed.

Generic objects are another special type of index object. Along with index objects, they are "gateways" to the design (see Figure 5.5). They represent major subsystems undergoing frequent change and refinement. They are independent
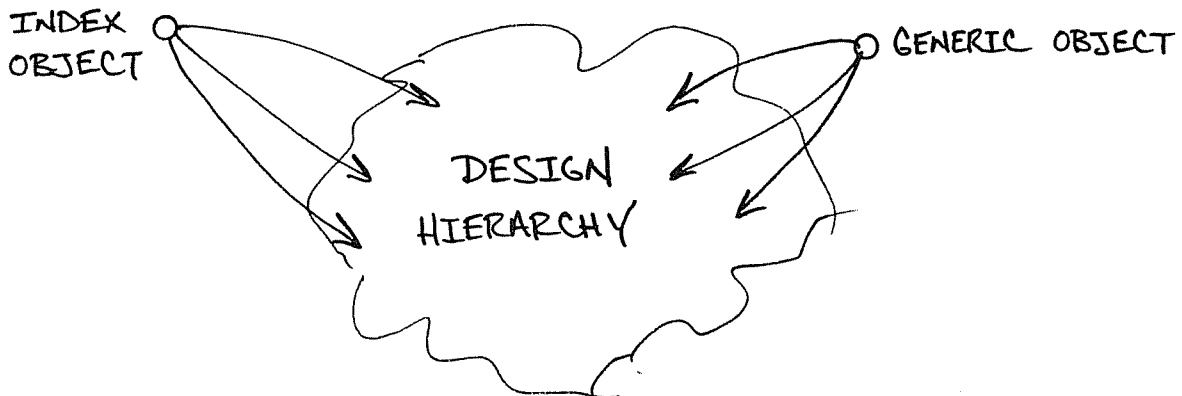

Figure 5.5 -- Gateways to the Design

objects, and can exist whether or not they are referenced by other objects. Additional structures for organizing alternatives and versions are used in their definition. A *version* configures the subsystem, by correlating versions of representation objects describing the subsystem in its different representations. Version objects are a special type of equivalency object, since the objects it groups together are constrained to be equivalent. They differ in that version exist independently of the objects they correlate. An *alternative* object is an index that groups together version objects, representing different versions of the alternative. Finally, a generic object (see Figure 5.6) groups its alternatives together. Each alternative has the same behavior as its generic parent, but its own performance characteristics. For example, the generic object "the ALU" is composed of *alternative* objects "fast ALU," "small ALU," and "low power ALU." The "fast ALU" alternative consists of various version objects, e.g., "fast ALU/version 0.0", "fast ALU/version 1.0," etc. "Fast
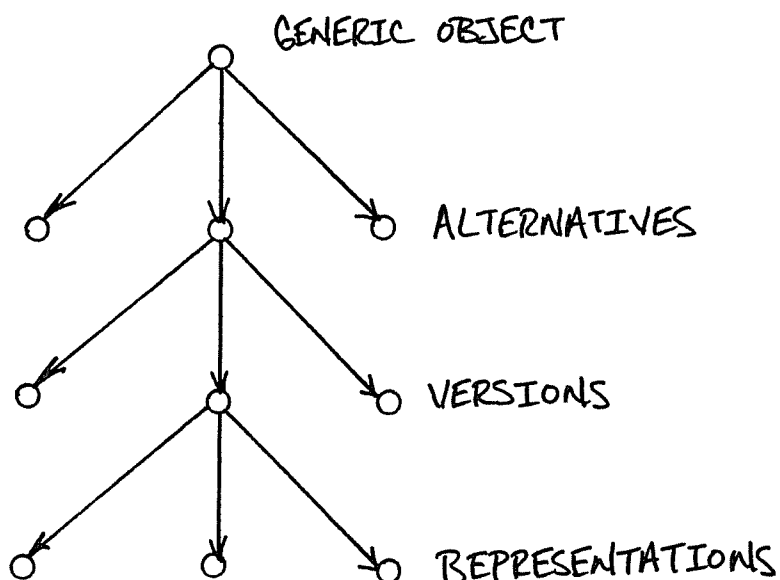

Figure 5.6 -- Generic Object Data Structure

ALU/version 1.0" in turn consists of the representation objects describing it, e.g., "fast ALU/version 1.0/layout," "fast ALU/version 1.0/transistors," and "fast ALU/version 1.0/gates."

Version objects, alternative objects, and generic objects offer a mechanism by which convenient groupings of representation objects can be formed. This is in addition to the representation hierarchies. Generic objects are nested on a representation by representation basis (see Figure 5.7). An *implementation* of a generic object is a version of one of its alternatives. To nest an ALU within a datapath proceeds as follows. An ALU implementation is chosen for inclusion within a datapath implementation. Each of its grouped together representation objects, e.g., the layout, transistor, and gate objects, incorporate the object of corresponding type in the ALU's implementation.

Library objects are generic objects of general utility, designed specifically for incorporation within other objects. The input/output pads used in almost every
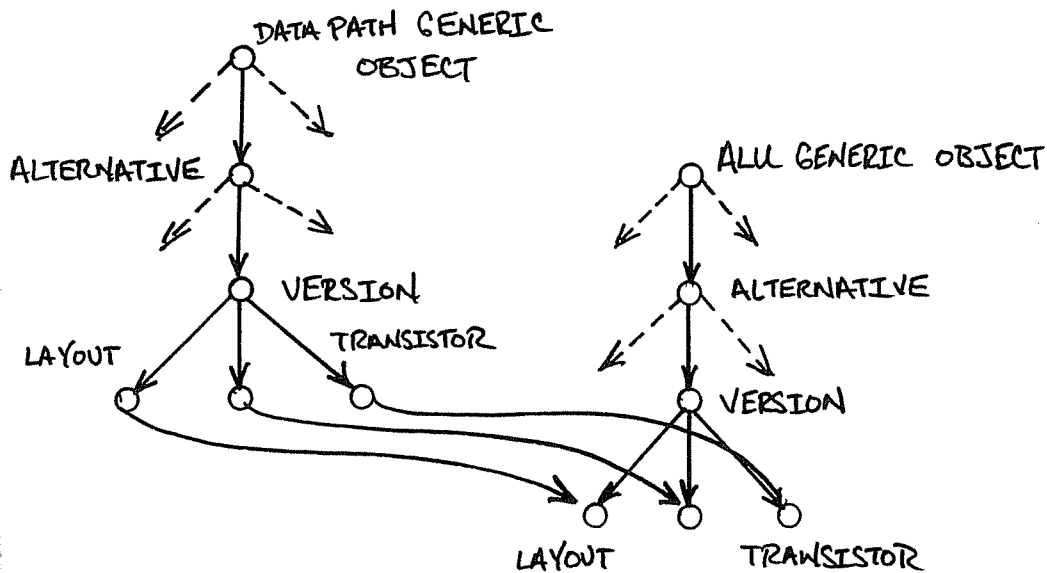

Figure 5.7 -- Incorporating Generic Objects

design are examples of library objects. *Designs* are special generic objects. Creation of the design is the objective of the design team. Designs cannot be nested. Only library objects can be shared among designs (see Figure 5.8).

5.2.2. Interfaces

Interface descriptions are associated with each representation object. An interface should contain enough information about the object so it can be used without having a detailed understanding of its implementation. It provides documentation of the object and makes connectivity information explicit. Interfaces include the following:

(1) **Name:** the name of the object, including its version.

(2) **Designer:** the designer who is responsible for its implementation (not necessarily the designer who implements it).

(3) **Description:** a description of the object's behavior (e.g., an English-language description, a truth table describing outputs in terms of input combinations, etc.). This description is primarily for documentation purposes, but could eventually be used by validation tools.
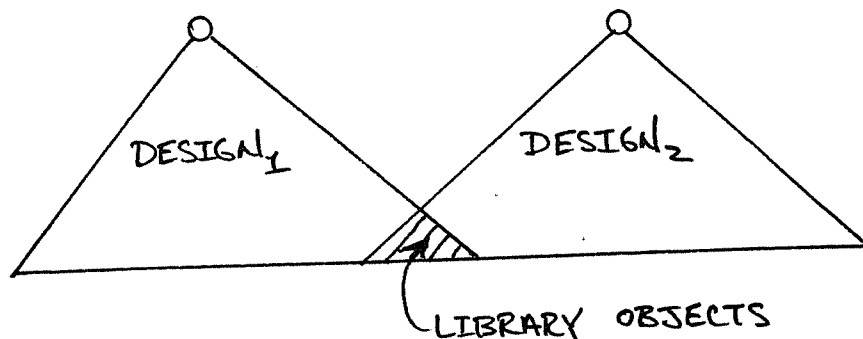

Figure 5.8 -- Designs and Libraries

(4) **Graphical Representation:** Objects are contained within *bounding polygons.* This is a useful way to specify the "outline" of the object for viewing on a graphics terminal. Bounding polygons can be associated with any representation type.

(5) **Connectivity Information:** *Ports* are the input and output connection points of an object. Ports are named, have types and directions, and are placed on the periphery of the bounding polygon. The same name can be assigned to more than one port. Ports with the same name must be connected by some signal path within the object. The directionality of ports are input, output, and bidirectional. The type information depends on the representation. For example, for the layout representation, type information includes interconnection layer and the form of the logic level expected or supplied by the port.

(6) **Performance Information:** Depending on the representation, the interface specifies constraints on power, area, and delay exhibited by the object in performing its function.

The interface contains information that is used to verify that a design self-consistent. Type systems simplify checking that object compositions are well-formed. One particular type system has been used in the Stanford Cell Library [NEWK81]. The input types are: (1) 4:1 ratio (expects regenerated logic levels), (2) 8:1 ratio (does not expect regenerated levels), (3) switch control (expects regenerated levels), and (4) switched (does not expect regenerated levels). Output types are: (1) gate (produces regenerated levels), (2) superbuffer (produces regenerated levels), (3) switch logic (produces unregenerated levels), and (4) precharged (special). The compatibilities among the types is determined from a table, such as Table 5.1. Other type systems are also valid.

Port names can be *local* or *global.* The ports of two objects with the same local port names are different, while every port with the same global name is implicitly con-

	Outputs			
Inputs	Gate	Superbuffer	Switch Logic	Precharged
4:1	O.K.	O.K.	NO	O.K.
8:1	O.K.	O.K.	O.K.	O.K.
Switch Control	O.K.	O.K.	NO	O.K.
Switched	O.K.	O.K.	(1)	NO

Note (1): O.K., but charge sharing problems are possible.

Table 5.1 -- Compatibility of I/O Types

nected. Vdd and GND are typical global port names.

The interface constrains the object's implementation. Checking that power and area are within constraints is straightforward. The area constraint is associated with the layout representation, and can be determined from the bounding box of its geometries. The power constraint is also associated with the layout representation, where the DC power consumption of a module can be determined by examining the width-to-length ratios of transistors. Delay (timing) is much more difficult to specify, and in general can only be checked through detailed timing simulations.

When an interface is specified before its implementation, the performance information is approximate, and is specified within ranges. Once an object has been implemented, its performance information can be readily determined.

Describing the behavior of an object for documentation purposes is difficult, since there are so many ways to describe behavior. For a given object, one form may be more appropriate than another. Truth tables or logical expressions are well-suited for describing combinational logic. Transition tables or state diagrams are appropriate for sequential logic. Alternatively, an object can be associated with a program that "simulates" its behavior. Many language-based functional simulators use this approach. An approach suitable for functional/timing simulations (or circuit testing) describes the behavior of a module by the input waveforms and expected output

waveforms. Maintaining detailed waveforms may be expensive in storage space, but is the easiest way to specify the expected timing behavior of a module. The description portion of the interface must be general enough to support any of these. At the very least, it can be an uninterpreted character string. We plan on identifying particular description types, and making these understood by the validation component of the system.

An object's implementation must be shown to agree with its interface. Implied connectivity between ports must be verified, as must be their types and directionality. An "interface extractor" program could aid in the verification. Note that implementations do not need to be checked if these have been derived from their interfaces ("correctness by construction"). Once could envision a PLA generator that takes the interface's behavioral description, perhaps specified as a set of Boolean equations, along with the locations of input, output, power, ground, and clock ports, and generates the PLA layout within the object's bounding polygon.

5.2.3. Composition and Interface

Composing objects to form composite objects can be viewed in graphical terms. Placed within the composite's bounding polygon are the polygons of its components. Component ports are wired together to "compose" them (wiring can be by *interconnection* or by *abutment*). The ports of a composite object are derived from component ports. This is specified by wiring the ports of the composite to the ports of its components (see Figure 5.9).

Information describing composition is associated with both composite and component objects. Each component object stores the names of all objects that contain it. A composite object stores the names of its components. It gives unique names to component instances, describes how each is oriented and placed within its bounding polygon, and describes how the ports are wired.
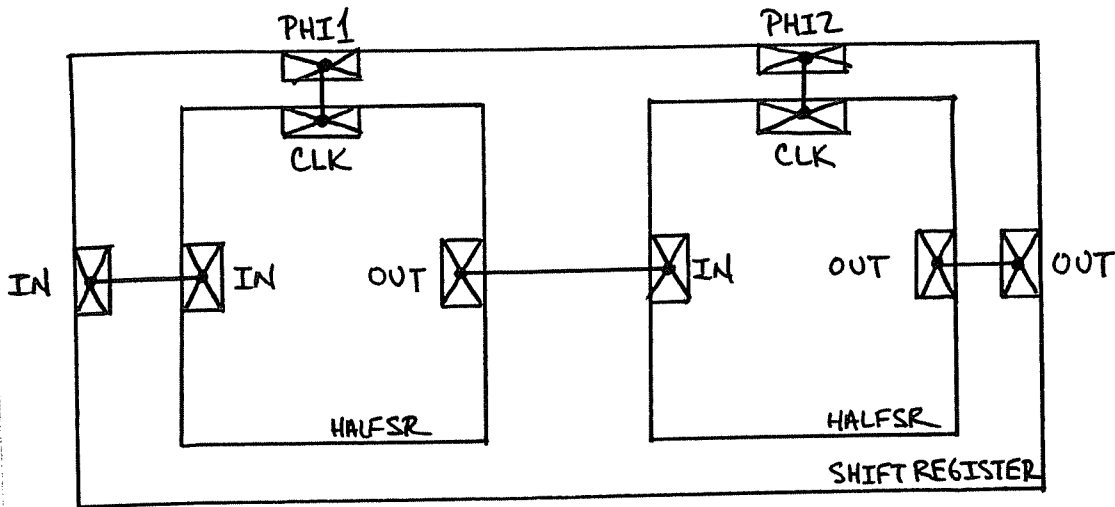
Figure 5.9 -- Graphical Compositions

5.2.4. Objects as Structured Files

Objects can be implemented as files extended with information describing the design data structure. The information relevant to the design management system is the object type, the interface specification, and the composition information (see Figure 5.10). A sample specification is given in the Appendix. Representational details are determined by the design tools, not the design data management system. We expect that new tools will be created that combine the creation of the interface, composition, and representation specifications. Existing design files can be referenced from within design object files to include their data in the design data structure.
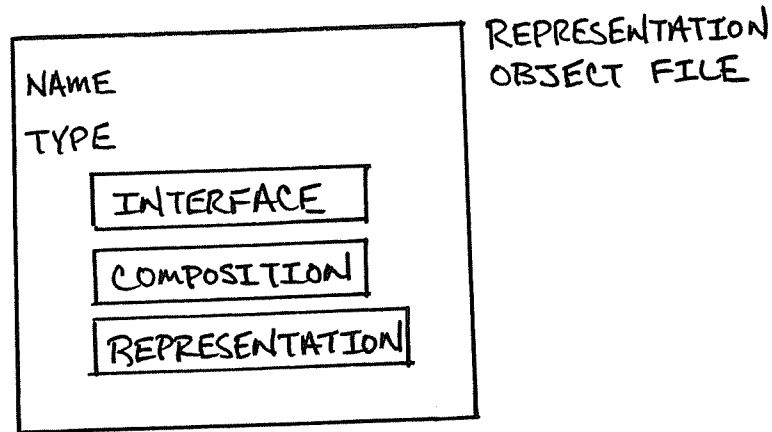
5.3. Design Data Management System

NAME

TYPE

INTERFACE

COMPOSITION

REPRESENTATION

REPRESENTATION OBJECT FILE

Figure 5.10 -- Design Objects as Files

5.3.1. The Computing Environment for Design

Designs are created at workstations, connected by a high speed network to a shared database server. The design is created and manipulated at the workstations, with the server providing a centralized location for long term data storage. The workstations have simple I/O configurations, typically a single Winchester disk, while the server has a large number of disk devices on multiple channels. Designers at the workstations transfer the relevant portions of the design to their local disks, through the design transaction mechanism. Work proceeds independently at the stations, with incremental changes spooled to the server, providing back-up in case of a workstation crash. The changes are incorporated into the public database only after they have been shown to be valid.

The distributed database is actually more complicated than what we have outlined above. Some validation can be performed at the workstation, but much is done on special purpose hardware or high performance machines. The database server must keep track of the location of design data at all times. In the following, we concentrate on the database server/workstation interaction, but the ideas apply in the

more general environment.

5.3.2. Storage Component

The storage component manages design data on secondary storage. Although a conventional database system can serve as a storage component [HASK82, KATZ82a, LORI83], a transaction-based file system, such as [REED83], provides the required facilities at a potentially lower cost. For example, we are implementing our system on top of the UNIX file system, which must be extended to provide atomic update.

The storage component is a reliable archive for design files, residing at the database server. The notion of a design object is implemented at a higher level. When a designer wishes to create a new version of an object, he first acquires exclusive access to it from the Design Librarian, which is a client of the storage component. The storage component does not need concurrency control mechanisms, since its actions are serialized by the Design Librarian. A copy of the original file is then created on behalf of the Design Librarian at the server, and is simultaneously transmitted to the designer's workstation. Complex storage structures are unnecessary since whole files are read and written by the storage component. Once at the workstation, the files are processed by design tools. These typically read the files into virtual memory, manipulate the data in virtual memory data structures, and write them back when done. Convenient mechanisms for mapping complex data structures on disk into virtual memory is still an open research topic (see [HASK82, LORI83]).

Save actions are supported by the recovery subsystem. Changes made to the files at the workstations are kept in special change logs. When the changes are saved, the logs are transmitted back to the storage component to be merged into the server's copy of the file. The merge operation is atomic, insuring that files are never left with partially merged changes.

The merge is implemented as a conventional database transaction, if these are supported by the storage component. Otherwise, the merge is made atomic by creating a temporary copy, merging in the changes, and when complete, renaming the temporary file and removing the original. The merge can be restarted if a crash occurs. While the merge may appear time consuming, it can be overlapped with continued activity at the workstation. Note that individual design files are relative small, so the time to do a merge need not be excessive.

5.3.3. Object System

Design objects are files containing design management specific information in addition to representational primitives (see Appendix). The following information is included: (1) the object's type, (2) the objects that contain this object, (3) the object's components and how they are composed, and (4) the object's interface.

The object system maps the abstract notion of an "object" into the data describing the object, stored in storage component files. Design objects can be stored either as a single file, with combined representation and design management information, or as separate files. For representation types that are not known to the system, the data is stored in a separate file referenced within the design object file. The object system can pass this "raw" representation data to existing design tools, thus appearing to them as a conventional file system. The design management information is made available to components of the design system that traverse the complex structure of the design, such as validation aids and browsing tools.

5.3.4. Recovery Subsystem

The recovery manager insures that as much data as possible survives a system crash. This is accomplished by maintaining multiple copies with alternative failure modes. Each checked out design object has five files associated with it in the workstation and database server storage components: (1) the *local working file,*

storing the object at the workstation, (2) the *local change file* stored at the workstation and holding a log of changes since the last savepoint, (3) the *mirrored file*, storing the object in the database server, (4) the *global change file*, which holds changes spooled to the server but not yet saved, and (5) the *redo log* which contains all saved changes since the last archive (see Figure 5.11). The local and global change files contain the differences between the local file and its mirrored copy. They protect against data lost because of workstation crashes. The redo log contains the difference between the mirrored file and the original file. It protects against loses due to database server failure.

The recovery manager supports *savepoints*. At a designer or design tool initiated savepoint, data and change file buffers are forced to disk by the workstation's buffer manager. The local change file is copied to the database server and is atomically appended to the global change file. Only the changes are written back, not the complete file. A background process copies the local change file entries to the global change file, providing a *continuous recovery capability*. This guards against data lose from a local hard crash, and reduces the latency of a save. Space on the local disk
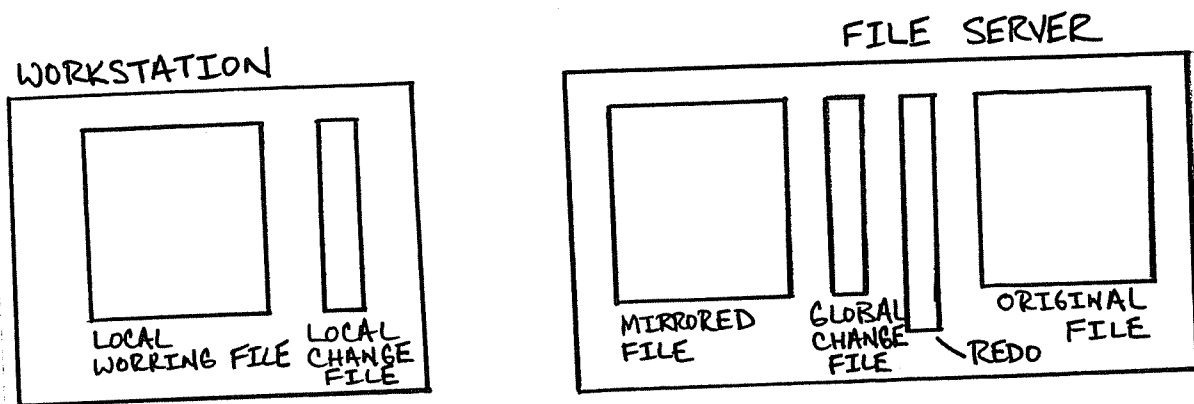

Figure 5.11 -- Files Associated with a Checked Out Object

is reclaimed as local change file entries are copied back. To commit the changes, the storage component atomically merges the global change file into the mirrored file and appends the global change file to the redo log.

Recovery from a soft workstation crash proceeds as follows. The last savepoint is reconstructed by copying the database server copy back to the workstation. A more up-to-date version is optionally reconstructed by merging the local and global change files into the local file. Some updates will be lost, since they were only in the local buffers, but others that were written to disk since the last save can be restored.

A hard workstation crash loses data on the workstation's disk. We assume that all local files are lost. The file is restored to its last saved state by copying the mirrored file back to the workstation. Alternatively, the file is restored to the last state known to the server by copying back the merged mirrored and global change files. The recovery subsystem determines what objects should be restored by accessing the information about who has what objects maintained by the Design Librarian.

The database server employs more conventional techniques to insure that its files are durable. The system is resilient to soft crashes because of the atomic operations supported by the storage component. Resiliency to hard crashes is provided as follows. Archival dumps are taken frequently. The mirrored files can be restored to their last savepoint from the archive copy plus the redo log. Note that the recovery subsystem only guarantees that objects can be restored to their last savepoint, although every effort is made to restore them past that point. Spooled changes that were not yet saved are lost since these are not normally logged. If the server loses the accumulated changes, it can always acquire the most up to date version of the file from the workstation. The global redo file should be duplexed to provide additional resiliency to hard crashes.

5.3.5. Design Librarian

The Librarian coordinates all access to shared design data. It creates and manipulates design objects and makes them available to workstations.

Designers (or design tools) acquire an object at a time. The designer uses browsing tools to navigate the hierarchies to find the interesting objects. Checking out an object for update gives him an exclusive right to create a new version of the object. The mirrored copies of design files, supported by the recovery subsystem on top of the storage component, become the new object versions when the changes are committed.

Representation objects are never affected by updates. Several designers could simultaneously create their own new versions, but such a proliferation of versions is undesirable from the standpoint of project management. The Design Librarian employs check out locks to guarantee that at most one in-progress version exists for an object.

In the rare instance of one designer holding an object that another designer needs, the designer who has it is permitted to complete its revision. He creates a new consistent version of the object. The requesting designer can then acquire it, creating his new version based on any of its previous versions. He is not restricted to creating a new version of the most up-to-date existing version, although this will be the most common situation.

Versions provide a flexible method for managing concurrent access to design objects. Previous versions can be browsed without regard to in-progress update activity. Simple locking protocols protect against the proliferation of versions. This is an example of how the design environment requires much simpler mechanism than are needed in the general transaction processing environment.

5.3.6. Validation Subsystem

The validation subsystem helps designers keep track of what has to be revalidated after a design change. Validating the fully instantiated design is prohibitively expensive. The design's structure must be exploited to keep the validation effort within reason.

A design is self-consistent if the following kinds of constraints are in force: (1) an object's implementation satisfies its interface (*conformance constraints*), (2) the composition of component objects is well-formed (*composition constraints*), and (3) objects specified as equivalent across representations are shown to be equivalent (*equivalence constraints*). Design transactions cannot complete until all constraints are satisfied.

We do not advocate a completely automatic approach for design validation. Designers will frequently need to intervene to apply the appropriate validation tools and to interpret their results. To facilitate this, the validation subsystem provides an audit capability, similar to that described in [NOON82]. Designers report who they are, what constraints they have validated, and what tools have been used for the validation. New validation tools can be written to directly record such information in the log. If a design failure is traced to a particular design object, the designer responsible for its "validation" can at least be identified.

An obvious area for further tools development is in interface extraction. Some parts of the interface are easy to extract, e.g., the location, type, and drive capability of ports. On the other hand, performance aspects like detailed delay information can only be determined through simulations. The design team performs the simulation and reports their results to the subsystem.

A composite object is well-formed if the interconnected ports of its components are type and direction compatible, e.g., restored output ports driving restored input ports. Component ports connected to the composite object's ports must be type

compatible and must have the same direction, e.g., an input port of the composite is connected to an input port of one of its components.

Some objects are generated from other representations. For example, PLA layouts are generated from Boolean equations in the functional description. Equivalence in this case can be checked by analyzing the validation audit trail. As long as (1) the last update time of the PLA layout object is the same as when the generator was run, and not later, and (2) the last update time of the PLA functional object is before that time, then the objects are known to be equivalent. Such equivalent constraints can be actively enforced by regenerating dependent objects on request.

Equivalence constraints can reduce the effort to validate equivalence across representations. Suppose we must show equivalence between the layout and transistor representations of a design. Normally, equivalence is checked by first extracting the full layout into a transistor description, and then providing the same stimulus to simulations of both the extracted transistor description and the original transistor description. Ideally, only the changed portions of the layout description and parts of the design that depend on them should be extracted and simulated. Equivalence constraints help to reduce both the time to extract and the time to simulate. If two objects in the different representations are already equivalent because they are connected by an equivalency object, then the transistor object can be substituted for its counterpart in the extracted transistor description without further analysis. A multilevel simulator can simulate the object at this level without needing to fully instantiate it. New validation tools could be written to take advantage of the equivalence constraints. In the meantime, designers validate the constraints piece by piece, and enter their results in the validation log.

A design transaction cannot complete unless all newly created objects (1) implement their interfaces, (2) are equivalent to the composition of their components, and (3) are equivalent to identified objects in other representations. Objects unaf-

fected by changes do not have to be revalidated.

5.3.7. Design Transaction Management

Design transactions are the mechanism by which designer create new consistent versions of design objects. Design transactions consist of *work, validation,* and *completion* phases. While work and validation can be intermixed, validation must be complete before a design transaction is allowed to enter completion.

During the *work* phase, a designer requests design objects from the Design Librarian. If the object has not been granted to another designer, the request is honored and the appropriate files are transferred to the workstation's disk. Additional *mirrored* copies are made in the database server, providing redundant copies used for recovery purposes.

If the object has already been acquired by another designer, its holder and its expected time of return are identified. "Deadlock" is rare, but nonetheless possible. Since designers can always determine who has what objects, deadlock is resolved through negotiation. Once the needed objects are at the workstation, the designer manipulates these with his design tools.

We have described the savepoint mechanism in our discussion of the recovery subsystem. These protect transactions from loss of data due to local crashes. Since the changes are continuously being spooled to the server by a background process, most can be restored. Note that activity at the workstation can continue even though the connection to the server is broken (fortunately rare!), but this is not advisable. It exposes the designer to serious loss of data in a local crash.

The mirrored copies at the shared database server simplify browsing of in-progress data. A relatively recent, savepoint consistent version of the design can be viewed without accessing the data stored at the remote workstation.

When design work is completed, the transaction enters a *validation* phase. Verification programs are invoked by designers to check that the modified design data is self-consistent. If validation fails, the inconsistencies must be located and corrected. The transaction reenters the work phase and validation is retried. Checking is usually distributed throughout the lifetime of the transaction, and need not only occur at the termination of design activity. The validation subsystem insures that all relevant constraints have been enforced before a transaction is permitted to enter completion.

During the *completion* phase, the mirrored copies of design objects are made available as new objects. If part of an independent object, the new version is added to those pointed to by its appropriate alternative object. If a designer decides to abort his transaction, the global and local copies are destroyed and the original objects are made available again for checkout.

To see how a designer creates a new design, consider the following example (see Figure 5.12). He wishes to create a new version of a microprocessor with a revised register file design. He invokes a design transaction, acquires the register file object from the Design Librarian, and revises it, creating a new version. To
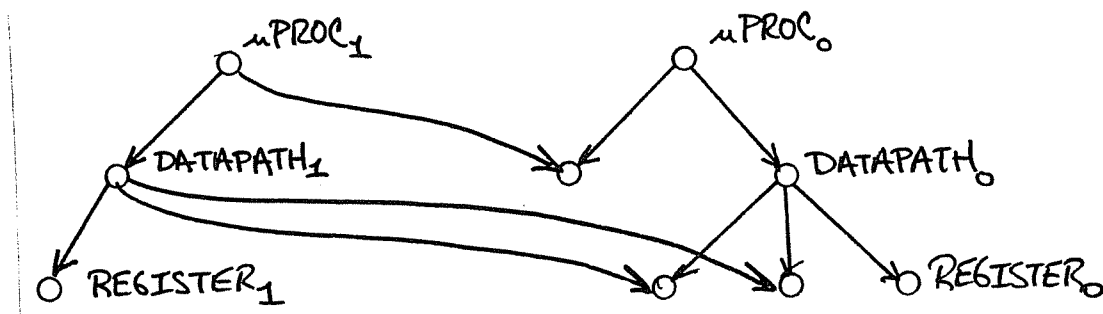

Figure 5.12 -- Creating a New Design

create the new microprocessor, he must first create a new datapath object incorporating the new register file. He acquires the datapath object, and recomposes it from its original components and the new register file. He then acquires the microprocessor object, and repeats the process. The new microprocessor design, with revised datapath and register file, is validated. The design transaction can now complete successfully with the new objects becoming a permanent part of the design database.

Design transactions make a heavy demand on disk resources. However, redundancy is unavoidable if resiliency to crashes is to be obtained. Since the server is dedicated to providing database services to the network of workstations, it should be equipped with a large number of disk devices.

5.3.8. Browser/Chip Assembler

The Browser and Chip Assembler are the interactive interfaces to the design management system [KATZ83b, MUDG80, MUDG81]. The Browser allows a designer to navigate through the complex structures describing a design, making extensive use of graphics to present the data structure, and menus to direct the navigation.

Part of the motivation for interface descriptions is to provide on-line design documentation. Browsing through object versions provide insight into how it has evolved over time. Alternatives within design libraries can be scanned to find a particular implementation of a function suitable for incorporation into a new design. A design can be browsed within a representation, or across representations, at any level of composite object or at the fully instantiated level of primitive objects. Several ways of browsing the design can be underway simultaneously. Each browse is presented within its own window.

The Browser assists the designer find objects with particular attributes. Index objects group together objects with similar attributes. For example, the ALU index object is composed of all generic ALU objects stored in the design database. The

ALU index object can be further included in a datapath index object, etc., creating a hierarchy of indices. Index objects are entry points into the design database.

The Chip Assembler takes data created by synthesis tools, such as graphic editors and datapath generators, and packages them as objects for inclusion within the design database. In conjunction with the Browser, it provides an interactive frontend for constructing the design data structure. It supports the interactive (1) composition of objects from more primitive objects, (2) specification of object interfaces, (3) identification of equivalences across representations, (4) construction of design configurations through version and alternative objects, and (4) creation and manipulation of index objects.

6. Summary and Conclusions

Design data management is a crucial feature of an integrated design system. The design data must be reliably stored on disk, concurrent access to it must be controlled, its versions and alternatives must be organized, and efforts must be made to keep it well-formed and consistent within and across representation hierarchies. Existing design tools tackle design synthesis and validation problems, but frequently ignore the important design data management issues. A design data management system provides these services to the ensemble of design tools, creating an environment in which tools become integrated into a coherent design system.

We are in the process of implementing a system with many of the design management features described here. Since we could find no suitable database system for our storage component, we initially created our own data management system tailored for design applications (the Wisconsin Storage System). We have decided to redesign the storage and object components for direct use with the UNIX file system, enhancing the transportability of our design management system. Efforts are underway to implement the object model, Design Librarian, Browser, and Chip Assembler described here.

7. Acknowledgements

8. References

[BAKE80] Baker, C. M., C. Terman, "Tools for Verifying Integrated Circuit Designs," *Lambda*, Fourth Quarter 1980.

[BEET81] Beetem, J. F., "Structured Design of Electromic Systems using Isomorphic Multiple Representations," Ph.D. Dissertation, Stanford University, (December 1981).

[DATE81] Date, C. J., *An Introduction to Database Systems*, Third Edition, Addison-Wesley, Reading, MA, 1981.

[DATE82] Date, C. J., *An Introduction to Database Systems, Vol. II*, Addison-Wesley, Reading, MA, 1982.

[EAST81] Eastman, C., "Database Facilities for Engineering Design," *Proc. IEEE*, V 69, N 10, (October 1981).

[JOUP83] Jouppi, N. P., "TV: An nMOS Timing Analyzer," Third CalTech Conference on VLSI, Computer Science Press, Rockville, Md, 1983.

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

[KATZ82a] Katz, R. H., "A Database Approach for Managing VLSI Design Data," Proc. 19th ACM/IEEE Design Automation Conference, Las Vegas, Nv., (June 1982).

[KATZ82b] Katz, R. H., T. Lehman, "Database Support for Versions and Alternatives of Large Design Files," submitted to *IEEE Trans. on Software Engineering*.

[KATZ83a] Katz, R. H., et. al., "'Big Caesar and Little Caesar': Adventures in Modifying and Extending CAD Software," *VLSI Design*, January/February 1983.

[KATZ83b] Katz, R. H., S. Weiss, "Chip Assemblers: Concepts and Capabilities," Proc. 20th ACM/IEEE Design Automation Conference, Miami, Fl., (June 1983).

[KATZ83c] Katz, R. H., S. Weiss, "Design Transaction Management," submitted to 9th Intl. Conf. on Very Large Databases, Florence, Italy, (October 1983).

[LORI83] Lorie, R. A., W. Plouffe, "Complex Objects and Their Use in Design Transactions," ACM SIGMOD Conference on Databases for Engineering Design, San Jose, CA, (May 1983).

[LOSL75] Losleben, P., "Data Structures, Data Base, and File Maintenance," in *Digital System Design Automation: Languages, Simulation & Data Base*, M. A. Breuer, ed., Computer Science Press, Inc., Woodland Hills, CA, 1975.

[LOSL80] Losleben, P., "Computer Aided Design for VLSI," in *Very Large Scale Integration VLSI*, D. F. Barbe, ed., Springer Series in Electrophysics 5, Springer-Verlag, Berlin, 1980.

[MCLE83] McLeod, D., et. al., "An Approach to Information Management for CAD/VLSI Applications," ACM SIGMOD Conference on Databases for Engineering Design, San Jose, CA, (May 1983).

[MEAD80] Mead, C., L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

[MUDG80] Mudge, J. C., Peters, C., Tarolli, G. M., "A VLSI Chip Assembler," in *Design Methodologies for Very Large Scale Integrated Circuits*, Nato Advanced Summer Institute, Belgium, 1980.

[MUDG81] Mudge, J. C., "VLSI Chip Design at the Crossroads," in *VLSI 81: Very Large Scale Integration*, J. P. Gray, ed., Academic Press, London, 1981.

[NEWK81] Newkirk, J., et. al., "Stanford nMOS Cell Library," Stanford Information Systems Laboratory, Technical Report TR 001, (July 1981).

[NEWT81] Newton, A. R., "Computer-Aided Design of VLSI Circuits," *Proc. IEEE*, V 69, N 10, (October 1981).

[NOON82] Noon, W. A., K. N. Robbins, M. T. Roberts, "A Design System Approach to Data Integrity," Proc. 19th ACM/IEEE Design Automation Conference, Las Vegas, Nv, (June 1981).

[OUST81] Ousterhout, J. K., "Caesar: An Interactive Editor for VLSI Layout," *VLSI Design*, Fourth Quarter 1981.

[OUST83] Ousterhout, J. K., "Crystal: A Timing Analyzer for nMOS VLSI Circuits," Third CalTech Conference on VLSI, Computer Science Press, Rockville, Md., 1983.

[SEQU83] Sequin, C. H., "Managing VLSI Complexity: An Outlook," *Proc. IEEE*, V 71, N 1, (January 1983).

[SUSK82] Suskind, J. M., et. al., "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions," Proc. Conf. on Advanced Research in VLSI, Paul Penfield, Jr., ed., Artech House, Dedham, MA, (January 1982).

[THOM78] Thompson, K., "UNIX Implementation," The Bell System Technical Journal, V 57, N 6, Part 2, (July-August 1978).

[REED83] Reed, D., "Implementing Atomic Actions on Decentralized Data," ACM Trans. on Computer Systems, V 1, N 1, (February 1983).

9. Appendix -- Sample Object Specification

The graphical presentation of the object is as in Figure 5.9.

```
(
  (NAME ShiftRegisterCell)
  (VERSION 1.0)
  (DESIGNER Randy H. Katz)
  (TYPE Independent Representation Layout)
  (TIME Mon May 2 21:43:15 CDT 1983)
  (WITHIN (Register 1.0) (Register 2.0))
  (INTERFACE
      (POLYGON (0 0) (0 20) (20 20) (20 0))
      (PORTS
          (LOCAL PORTNAME In DIRECTION Input TYPE 4:1 LOCATION (0 10))
          (LOCAL PORTNAME Out DIRECTION Output TYPE Gate LOCATION (10 10))
          (GLOBAL PORTNAME Phi1 DIRECTION Input TYPE 4:1 LOCATION (5 20))
          (GLOBAL PORTNAME Phi2 DIRECTION Input TYPE 4:1 LOCATION (15 20))
      )
      (DESCRIPTION 2 phase dynamic shift register cell)
      (PERFORMANCE (DELAY 5 ns) (AREA 42 um BY 42 um) (POWER 10 uw))
  )
  (COMPOSITION
      (INSTANCE x NAME HalfRegister TRANSLATED (0 0))
      (INSTANCE y NAME HalfRegister TRANSLATED (10 0))
      (INTERCONNECT
          ((y In)  (x Out))
          ((x In)  (ShiftRegisterCell In))
          ((y Out) (ShiftRegisterCell Out))
          ((x Clk) (ShiftRegisterCell Phi1))
          ((y Clk) (ShiftRegisterCell Phi2))
      )
  )
  (REPRESENTATION)
)

(
  (NAME HalfRegister)
  (VERSION 1.0)
  (DESIGNER Randy H. Katz)
  (TYPE Dependent Representation Layout)
  (TIME Sun May 1 09:31:15 CDT 1983)
  (WITHIN (ShiftRegisterCell 1.0))
  (INTERFACE
      (POLYGON (0 0) (0 20) (10 20) (10 0))
      (PORTS
          (LOCAL PORTNAME In DIRECTION Input TYPE 4:1 LOCATION (0 10))
          (LOCAL PORTNAME Out DIRECTION Output TYPE Gate LOCATION (10 10))
          (LOCAL PORTNAME Clk DIRECTION Input TYPE 4:1 LOCATION (5 20))
      )
      (DESCRIPTION Half of dynamic shift register)
      (PERFORMANCE (DELAY 2 ns) (AREA 42 um BY 21 um) (Power 5 uw))
  )
  (COMPOSITION)
  (REPRESENTATION Boxes
```

```
(Metal (0 0) (10 4))
  .  .  .
(Poly (4 20) (6 0))
)
)
```