Papyrus: A History-Based VLSI Design Process Management System

by

Tzi-cker Chiueh

B.S. (National Taiwan University) 1984
M.S. (Stanford University) 1988

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering -- Electrical Engineering
and Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor Chittor V. Ramamoorthy
Professor Sadashiv Adiga

The dissertation of Tzi-cker Chiueh is approved:

Randy H. Katz 11/8/92

Chair Date

11/8/92

Date

11/8/92

Date

University of California at Berkeley

November 1992

Papyrus: A History-Based VLSI Design Process Management System

Copyright © 1992

by

Tzi-cker Chiueh

Abstract

Papyrus: A History-Based VLSI Design Process Management System

by

Tzi-cker Chiueh

Doctor of Philosophy in Electrical Engineering
and Computer Science

University of California at Berkeley

Professor Randy H. Katz

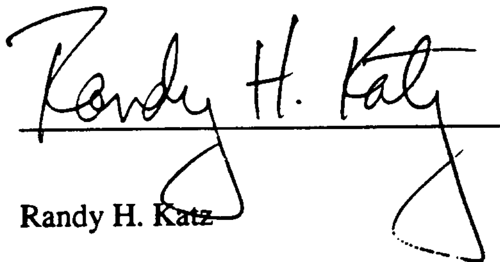
With the advent of powerful computer-aided-design (CAD) tools and increasingly complicated VLSI systems, the notion of circuit design has evolved into managing complexity rather than manipulating electronic devices. Complexity arises from enormous amounts of design data as well as the complicated process of creating design data. An important observation is that modern circuit designers spend more time managing the created design data than actually running the CAD tools. The thesis of our work is: the key to further enhance VLSI designers' productivity is not better CAD tools but a more responsive infrastructure. The focus of this dissertation is thus on support mechanisms that allow composition of a set of potentially heterogeneous tools into a coherent design system, and facilitate the integration of design data and design process management.

We develop a design process support model called the *Light Weight Transaction* (LWT) model, which captures both the *structured* and *exploratory* aspects of VLSI design. The former corresponds to the design procedures that are well-understood and thus can be specified in advance, whereas the latter denotes the creative part of a design process. We developed a script facility to support routine design activities and proposed a history-based rework mechanism to allow interactive exploration of the design space. Unlike conventional database transaction models, the LWT model is based on a data visibility abstraction: users can operate on a piece of data only when it is visible to them. We

have shown how this abstraction can support both design exploration and cooperative group work.

To demonstrate the feasibility of the LWT model, we built a prototype implementation on top of the *Sprite* operating system, the Tcl/Tk facility, and the Berkeley OCT CAD tool suite. This implementation features a transparent load balancing scheme to exploit the computation power of networked workstations, an atomicity-guarantee mechanism to preserve the high-level task abstraction. In addition, the rework mechanism depends on a single assignment update principle, which in turn could pose serious storage overheads. Our implementation alleviates this overhead by performing a history-based object reclamation in the background.

Based on the design operation history, we propose a novel design management paradigm: Rather than requiring users to supply design meta-data, the system maintains and analyzes the design history to deduce the metadata, in particular, object attributes and inter-object relationships, according to a suite of domain-specific knowledge and inference procedures. This paradigm can be viewed as a generalization of the approach used in syntax-directed editors. However, we believe this to be the first attempt to apply the idea in the context of design database management systems. Instead of using abstract syntax trees, we use a special representation of the design history called *augmented derivation graph* as the basis for design metadata inference. This paradigm opens a new way of thinking about creating information that are interesting to the system, be that a user, an operating system, or a database system.



Randy H. Katz

Acknowledgement

First, I would like to thank my advisor, Professor Randy Katz, for providing a level of support that a graduate student can ever want. By example, he shows that a professor who performs world-class research can also be a caring and enthusiastic teacher. During the four-year stay in Berkeley, I enjoyed and benefited enormously from the numerous stimulating discussions and enlightening conversations. Most of all, Randy offers a latitude for me to work with that no professor can possibly give. For all of these, I hope I can repay him by following his step when I become a professor myself. I also want to thank Professor Adiga and Professor Ramamoorthy for being on my dissertation committee and for patiently reading through the drafts of my thesis.

Like all great research universities, stimulation from fellow students makes up a significant part of one's graduate education. I was fortunately enough to have the following people as my officemates: Ann Chervenak, Mike Dahlin, Valerie King, Ethan Miller, Fred Obermeier, Srini Seshan, Mario Silva, and David Wood. I want to thank them for everything they've done for me. In particular, I want to thank David for setting an example on computer architecture research, Fred for teaching me various tricks of using workstations, Valerie for early discussions on my work, Srini for interesting ideas and discussions, and Mario for his warm friendship.

My parents and brothers deserve special thanks for being always understanding when I said that I need four years to finish my PhD degree. Without their constant prodding, I might still wander around without getting anything done. My big brother, Tzihong, in particular convinced me that theoretical physicists are no more than problem solvers, and that a good scientist should not restrict himself to any particular field. Looking back, I can only say: "How true! How true!" For that I want to thank him.

Last, I want to thank my wife, Sheng-I. For still believing in me after I broke several promises on honeymoon trips. For always patiently listening to my complaints

and frustrations. For taking three hours of commute to work everyday in order to let me walk to school. For trying painfully to understand my work upon my insistence. For cheering with me when my paper got accepted and griping for me when things didn't go our way. For just being there so that I know I can never go too wrong. There must be tons of barriers on the way to get a PhD. But Sheng-I renders them so trivial and makes my journey a delightful adventure. Now that I am done, Honey, let's go to Hawaii!

Contents

Chapter 1 Introduction 1

1.1 The Problem 1

1.2 New Dimension of Design Data Management 2

1.3 What A Design Flow Manager Is Not 3

1.4 Functional Requirements of A DFM 5

1.5 Thesis Overview 9

Chapter 2 Related Works 12

2.1 Process Support Systems 12

2.2 Electronic Design Environments 13

2.2.1 EDA's PowerFrame 13

2.2.2 Automatic Design Manager -- *VOV* 15

2.2.3 Tool Execution Control Systems at Carnegie Mellon 17

2.2.4 Distributed Design Methodology Management 22

2.2.5 IDEAS 23

2.2.6 Petri Net-based Systems 23

2.3 Software Engineering Environments 25

2.3.1 Process Programming 25

2.3.2 Hewlett Packard's Tool Encapsulator 26

2.3.3 IBM's Programming Process Architecture 26

2.4 Office Automation Environments 27

2.4.1 POISE 27

2.5 Summary of Previous Systems 29

Chapter 3 The Conceptual Model 32

3.1 Introduction 32

3.2 Basic Assumptions 32

3.3 Light Weight Transaction Model 35

3.3.1 Design Step 36

3.3.2 Design Task 37

3.3.3 Design Thread 41

3.3.4 Interaction Among Design Threads 47

3.3.4.1 Thread Manipulation 47

3.3.4.2 Thread Synchronization 51

3.4 Summary 54

Chapter 4 Design Task Management 58

4.1 Introduction 58

4.2 The Task Description Language 59

4.2.1 Tool Command Language 60

4.2.2 Extensions 63

4.2.3 Examples 66

4.3 Implementation 71

4.3.1 Tool Navigation/Encapsulation 71

4.3.2 Parallelism Extraction 74

4.3.3 Re-Migration 76

4.3.4 Programmable Abort Semantics 77

4.3.5 History Recording 79

4.3.6 Attribute Management 80

4.4 Summary 80

Chapter 5 Design Activity Management 82
5.1 Introduction 82
5.2 User Interface 84
5.3 History Management 92
5.4 Storage Management 96
5.5 Summary 101

Chapter 6 Automatic Metadata Inference Based on Design History 102
6.1 Introduction 102
6.2 Related Works 104
6.3 Models of Design History Representation 105
6.4 Incremental Meta-data Construction 108
6.4.1 Type Inference and Attribute Evaluation 110
6.4.2 Relationship Establishment 114
6.4.3 Discussion 119
6.5 Conclusion 121

Chapter 7 Conclusion and Future Works 122
7.1 Research Contributions 122
7.2 Future Directions 125
7.2.1 What's Next in VLSI Design Systems? 125
7.2.2 Other Lines of Research 127
7.3 Final Words 128

Chapter 1

Introduction

1.1 The Problem

After three decades' development of computer-aided-design (CAD) tools for integrated circuit design, modern VLSI design is characterized by extensive use of automation tools. From initial conceptual specification to low-level physical layout, almost all design work is carried out through tool invocations. Circuit designers, once the experts in manipulating electronic devices, now become *managers* of design tools and the design data that they generated. Because of the rapid evolution of semiconductor technology and electronic industry, state-of-the-art VLSI design is also moving towards more and more technology-dependent, methodology-driven, and product-oriented. Accordingly the functionality of individual design tools becomes specialized, which leads to a proliferation of CAD tools that are usually created by different vendors with rarely compatible interfaces. Consequently VLSI circuit designers are now facing not only a great deal of design data but also a large number of design tools interacting with one another in complicated ways.

With the advent of these highly automated tools, circuit design basically consists of two types of activities: running CAD tools and managing the created design entities. Almost all the CAD tool research in the last thirty years was focused on the former, i.e, on how to perfect each individual step in the design process. However, just because each such step is optimized, it doesn't mean that the entire design process is necessarily optimal. It has been observed in the electronic design community that a set of powerful tools is *not* equivalent to a productive design environment. Why? The reason is today's circuit designers actually spend more time in monitoring the evolution and integration of design data, and figuring out which tools/tool sequences to invoke, than in actually running the tools.

To further improve the productivity of VLSI designers, CAD researchers should take one step back, examine where the bottlenecks are in the entire design process, and strive to reduce the impact of or completely eliminate these bottlenecks. It is our thesis that as individual CAD tools mature, helping circuit designers to use the right tools at the right time, and to keep track of design data evolution, is the single most effective way to achieve significant improvements in design technology. As a result, rather than concentrating on a specific CAD tool, we take a *systems* approach towards the development of an integrated VLSI design environment. The emphasis is on the mechanisms to transform a set of loosely-coupled tools into a coherent whole, and the abstractions to manage complicated design data. This dissertation describes the design and implementation of a prototype design process system called *Papyrus*, which is based on such an integrated approach.

1.2 A New Dimension of Design Data Management

To put the contributions of this work in perspective, we briefly describe the evolution of VLSI design database management systems, and show where this work stands in relation to previous research efforts. First-generation design databases [WONG79] [MITS80] [ROBE81] [ZINT81] [CHU83] focused on efficient storage and manipulation of internal design data structures such as circuit schematics and physical layouts.

Second-generation design database systems [KATZ87] [WEIS86] [BATO85] [OCT91] recognized the complexities of VLSI design data consisting in not only the internal representations but also the intertwined relationships among pieces of a design, and therefore emphasized the *structural* aspects of design data by providing primitives to model the interrelationships among design components, such as version, configuration, and equivalence relationships. In both generations, the systems only dealt with the *static* aspect of VLSI design, i.e., internal representations and the external structures of VLSI circuits Almost no attention was paid to the *dynamic* aspect of VLSI design, i.e., controlling the design process and monitoring design data evolution.

We believe that the next evolutionary step in design database research is to provide support to facilitate the *process* that creates the design data, and to integrate the management of design data and design process. We use the term *design flow management* (DFM) to refer to this new breed of design database systems. A DFM consists of two components: a *design data manager* (DDM), which manages both design data and metadata, and a *design process manager* (DPM), which controls the sequencing of design activities. These two components are functionally orthogonal, but interact with each other in a synergistic way as we will show in the following chapters. This dissertation mainly focuses on the design process management aspect of a DFM, assuming that a generic object-oriented-like database system takes care of design data management. In our implementation, the design database is OCT [HARR86]. A set of concrete functional requirements that characterizes a DFM are listed in Section 1.4.

1.3 What A DFM Is Not

Because design flow management is a relatively new concept, it is also important to understand what it is not, just to set a common ground to continue this presentation. First of all, a DFM is not an implementation of the *CAD Framework Initiative* (CFI) specification. The CFI proposal specifies standard formats for various types of VLSI design data, and procedural interfaces for individual CAD tools and their communications. Although it does have a design methodology/flow management component, the

ideas presented in this work are completely independent of the CFI specifications, and certainly doesn't conform to the standard in any way.

A DFM is not just another design version control system. Although version management systems offer a primitive way of recording design evolution in the form of version history, a DFM not only provides more refined support for tracking design data, but also helps designers to create the design data in the first place. On the other hand, because a DFM manages both design data and design process, a DFM can actually complement a version control system by inferring version-related metadata from the design process, as explained in Chapter 6.

Although it is tempting to categorize a DFM as a long-term transaction system [], it turns out that the fundamental goals of our work are very different from those of long-term transaction systems. For example, in this work no attention is directed to crash recovery. Also maximal concurrency is not as important as design context maintenance. In the choice of concurrency control algorithms, conceptual simplicity dominates theoretical elegance. The reason is that these mechanisms are supposed to be used directly by circuit designers, who may not have the slightest idea of what a transaction is, not to mention more complicated notions of nested or compensating transactions [MOLI87].

Even though a DFM could automate some of the design steps during the design process, it is not a high-level synthesis tool. There are two major differences. First, the major goal of a DFM is to facilitate the designers' interaction with the design environment. A DFM assists but doesn't replace human designers. It is still the human designer that makes most of the design decisions. Second, a DFM is a generic software layer that is built on top of a set of tools and a design database system. In this sense, a DFM assumes an *open* architecture that is independent of the rest of the design environment, which is in contrast to the typically close-coupled synthesis systems whose ultimate goal is to automate the entire design process.

1.4 Functional Requirements of A DFM

Instead of giving a precise definition of a design flow management system, it is much more easier to describe a DFM in terms of a list of design goals for such a system. Specifically we have the following functional requirements that have guided the design and implementation of *Papyrus*.

Tool Encapsulation

Modern VLSI CAD tools typically have dozens of parametric options, each of which typically specifies how the tools behave in a particular aspect. As the tools become more powerful and versatile, the users tend to become confused and intimidated by the complicated tool command interfaces. The concept of *tool encapsulation* separates what a CAD tool does from how it is used. Ideally, users only need to express what needs to be done but not how to do it. By providing a layer of indirection, users are insulated from the invocation details of individual tools such as command options and input/output object naming. Consequently, users only interact with a consistent and high-level (preferably graphical) user interface. Actual invocations of CAD tools are handled by an interpreter that is completely user-transparent. Moreover, this extra level of indirection makes it easier to compose a set of heterogeneous tools into a coherent design environment since users are shielded from the underlying tool set. Modification on individual tools or replacement of one tool with another functionally equivalent one would not affect the users' perception towards the design environment.

Tool Navigation

Because of the proliferation of CAD tools, choosing the appropriate set of tools and applying them in the right order to achieve a given design objective is not a trivial task. Given a goal, a DFM provides a navigation mechanism that guides users through the tool invocation sequence which accomplishes that goal. Tool navigation facilitates routine design work by freeing experienced circuit designers from memorizing the exact tool sequence while leading novice users through an otherwise convoluted web of tools. Tool navigation is essentially a by-product of raising the design abstraction one level beyond

individual CAD tools. Instead of invoking primitive tools, users think and interact with what we call *tasks*. Tasks can automate pre-specified tool execution pipelines when user intervention is not needed, and/or can enforce high-level design methodologies that are specified in terms of constraints on tool execution sequencing.

Support for Design Exploration

Design by its very nature is a trial-and-error process. Exploration of various design alternatives and choosing the one that fits best according to some criterion is one of the salient characteristics of engineering design. And yet, the current design environments provide almost no support for what we call *design exploration*. From the users' standpoint, a DFM should allow them to travel back and forth between the *design states* that correspond to various design choices, and to examine the detailed design tradeoff without having to do the bookkeeping themselves for the mapping between design alternatives and the associated subset of design objects in the database. Design typically also involves many iterations of an identical sequence of steps, e.g., applying the same procedure over and over again to optimize certain property. A DFM facilitates this iterative refinement process by automating the iteration steps, and structuring the intermediate data objects in such a way that users can easily keep track of their correspondence relationships with the iterations.

Recording of Design Evolution

Being conservative, engineering design database is typically version-oriented: modifications are made on the copy of a design object rather than on the object itself. With this so-called *single-assignment* update semantics concurrent development on a design entity becomes possible when there is no need to guarantee mutual exclusion. The notion of *version history* in previous design databases [CHAN89] captures a design's history in the form of snapshots of the database. However, it is often desirable to maintain the design history beyond the simple "which object is derived from which" relationships. For example, the UNIX *Make* facility requires the knowledge of the detailed tool execution sequence that are involved in creating an object, i.e., its *derivation history*, to

reconstruct the design object when one or more of its dependent objects are modified. The concept of derivation history is an operation-based representation of the design history, in contrast to a version history based on snapshots.

Context Management

The notion of *directory* in a file system is very successful in helping users to organize files into a manageable hierarchy. Now that a DFM manages both design data and design process, one can apply the same idea to cope with complexity. We use the term *context* to refer to the set of design operations (i.e., CAD tool invocations) AND their input/output data objects that are associated with a particular design entity, which could be either a library cell, a circuit module, or a subsystem. Thus, a context is actually a generalized notion of a file system directory: the latter clusters related data, and the former clusters related data *and* the operations used to create these data. A context provides a focal point for circuit designers to concentrate their cognitive capacity only on relevant data and operations. A context answers that with respect to a design entity, what have been done? and what are the associated data objects? Contexts also offer a natural way for data protection and consistency enforcement via partitioning of the data space.

Support for Cooperative Work

Designing a VLSI chip is almost always a group effort. Although CAD tools have been developed that significantly enhance circuit designers' productivity, [CHOW88] has reported that there is still a significant portion of the overall development time spent in phases where automation tools are mostly lacking. In particular, as design technology matures the focus of VLSI design gradually shifts from improving individual productivity to facilitating project completion. One of the major problems is the so-called *integration problem:* individual modules work just fine, but when they are put together, something always goes wrong. This problem is especially serious when the interacting pieces are separately developed by different designers. There are essentially two aspects to the integration problem. The first aspect is related to version and configuration control. When a system is to be constructed, version/configuration control systems make sure that

the right configurations of the appropriate component versions are used in the composition process. However, simply guaranteeing right versions and configurations doesn't necessarily result in successfully integrated designs. The problem is that the interfaces of the interacting modules do not necessarily "fit" when integrated into a complete system. In other words, the modules are inconsistent with respect to VLSI domain semantics (e.g., timing/electrical characteristics). Incompatibility of interfaces may result from lack of communication, imprecise interface specifications, or both. In VLSI design, cooperation typically occurs via data sharing, for instance, a cell developed by one designer is used by the others, or two modules interacting with each other must agree on protocols at various levels of abstractions. A data sharing facility protects the rights of *producers* of design objects by preventing security violations, and keeps *consumers* from using objects that are not considered well-formed by prohibiting pre-mature access. In addition, change management in the form of active propagation or passive notification should be provided to maintain certain degrees of consistency.

Distributed Architecture

With the advent of powerful workstations in engineering design environments, a DFM should be able to take advantage of this technology by transparently distributing design data and tool execution across the network, thus making efficient use of the otherwise wasted CPU cycles. In a distributed environment it is important to guarantee certain atomicity property so that the side effects of a pre-specified tool execution sequence are either all or none across failures. It is also desirable to have some priority mechanism to prioritize tool execution and to load-balance workload in the networked computing environment. In this regard a DFM can be viewed as a specialized operating system that extends and tailors the services provided by the operating system to the needs of VLSI design.

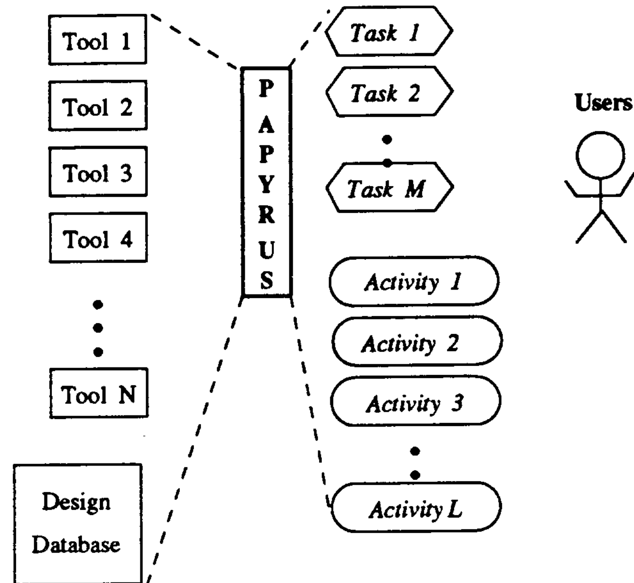


Figure 1.1 Conceptual Architecture of A Design Environment Using *Papyrus*

1.5 Thesis Overview

In summary, *Papyrus* provides the necessary hooks that glue a set of CAD tools, which may well come from different development sources, into a consistent environment as seen by the end users. In addition, it provides support mechanisms to facilitate generic engineering design activities such as alternative exploration or iterative refinement. *Papyrus* does not make any assumptions about the CAD tools and/or database systems. Figure 1.1 shows the global architecture of a design environment that uses *Papyrus* as the coordination entity of the design tools and design data in the system. From the user's standpoint, he or she only interacts with high-level abstractions such as design tasks and activities (to be explained in later chapters). As a result, individual CAD tools and their complicated interfaces and interactions are shielded from normal users, in this case the circuit designers. *Papyrus* also provides a link between design data and design process management. The rest of this thesis are organized as follows.

In Chapter Two, previous research efforts related to this work are surveyed. We are primarily interested in how previous software systems support *processes* in various

domains such as office automation, software engineering, and electronic design. The survey is meant to convey novel concepts developed in these systems rather than their complete descriptions.

In Chapter Three, the conceptual model that guides the design and implementation of *Papyrus* is described. We call this model the *light-weight transaction* (LWT) model. It aims to strike a balance between the rigorous control of data integrity in conventional database transactions and the user-friendliness requirements of an interactive engineering design environment. Rather than resorting to the notion of *atomicity*[*1], we advocate the concept of *visibility* as the central abstraction. The word *light-weight* is used to describe the fact that our model doesn't adhere to all the *ACID*[*2] requirements that characterize database transactions, but rather chooses to relax these requirements to accommodate creative design activities.

Chapter Four and Five describe the implementations of *Papyrus*'s two major components: the *Task Manager*, which is essentially a parallel and distributed process control system augmented with the capabilities of tool encapsulation and navigation, and the *Activity Manager*, which transparently maintains the contexts associated with various design alternatives, and provides a controlled data sharing mechanism to synchronize concurrent design activities. We discuss how these two subsystems are implemented on top of Sprite and the Tcl/Tk package, and the interaction among them.

In Chapter Six, we describe an application of the design operation history maintained for supporting design task and activity management. We propose a novel design data management paradigm in which design meta-data, such as per-object attributes and inter-object relationships can be inferred from a design object's derivation history, which is automatically collected as a by-product of activity management. This approach not only enhances the data management capability of a DFM but also relieves the designers from the burden of explicitly computing or entering some of the meta-data, thus indirectly improves the productivity of circuit designers.

*1 A set of operations is said to be *atomic* if either they all occur or none of them occurs.

*2 ACID stands for Atomicity, Consistency, Isolation, and Durability.

Chapter Seven summarizes the main research contributions of this thesis, re-examines how the proposed mechanism satisfy the stated design goals, recapitulates the lessons learned from this project, and provides a personal speculation on what future VLSI design systems should look like as well as the further research issues entailed by this vision.

Chapter 2

Related Works

2.1 Process Support Systems

A process support system aims to facilitate the work processes of developing artifacts, e.g., designing circuits or writing software. The notion of *process support* is a relatively new concept, which only has begun to emerge in the last decade. Before then, the consensus was that the data upon which an organization operated is the central focus of organizational computing. Correspondingly the emphasis of computer support has been placed on the database facility that allows fast and reliable access to the data. With the advent of ever more advanced database services, organizations recognized that to further promote their productivity, it is at least as important to improve the *process* of creating data as managing the data after they are created. In some domains, the process of generating data and the final data are both considered precious knowledge that needs to be documented and maintained.

In this section, we review a set of representative process-support systems in electronic design, software engineering, and office automation domains. The goal of this

survey is to understand the design issues for supporting work processes, how the proposed solutions in the surveyed systems address these issues, and to make the case for why design process management systems are needed. We make no attempt to describe the systems in greater detail than necessary. Despite this field's relatively young age, this list is also not meant to be exhaustive. We have based our selections on new ideas, not an exhaustive listing of new systems. Wherever possible, we examine and analyze the systems along the dimensions corresponding to the functional requirements outlined in the last chapter. Because we are mainly interested in VLSI circuit design, the survey is necessarily biased towards that particular domain.

2.2 Electronic Design Environments

2.2.1 EDA's PowerFrame

Primitive forms of VLSI design process management products have emerged in the last three years, e.g., the *Falcon* system from Mentor Graphics, the *Framework* system from Cadence, and the *PowerFrame* system from Digital Equipment Corporation. *PowerFrame* [GOLD88] was originally developed by EDA Systems and is perhaps the earliest one among these offerings. Since the takeover of EDA by DEC, *PowerFrame* is emerging as the de facto standard for design process management products in the industry. As can be seen in Chapter Three, *Papyrus* shares many ideas with *PowerFrame*.

In a word, *PowerFrame* attempts to make it easy for end users (i.e., circuit designers) to use the right tools on the right set of data objects. Moreover, the design process should be automated whenever possible. *PowerFrame* allows routine tool execution sequences to be specified in advance, and stored in a database for later use. These pre-defined sequences are called *templates*. Users do their design work by invoking templates. Multiple tasks can be invoked in parallel. *PowerFrame* instantiates templates and navigates users through the associated tool execution sequences.

PowerFrame provides a graphical user interface in which templates are represented as an annotated directed graph with various process and edge operators. For example, in

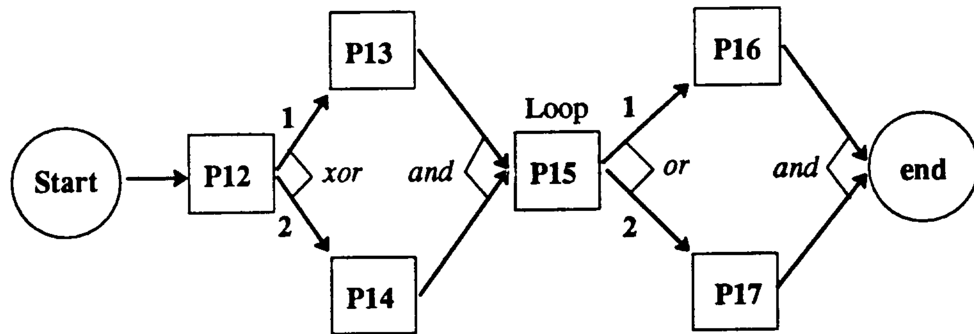


Figure 2.1 Graphical Representation of *PowerFrame*'s Task Templates

Figure 2.1, a task P consists of six tool invocations: P12, P13, P14, P15, P16, and P17. The sequencing among these tool invocations is specified in terms of edge operators such as *and, or,* and *xor*. The numbers on the edges represent priorities. For example, after the completion of P12, the *xor* operator indicates that only one of P13 and P14 is allowed to execute and P13 has a higher priority. The *or* and *and* operators assume obvious semantics. There are also process operators such as the *Loop* operator on P15, which means that P15 will be invoked on elements of a set or a queue.

On the data management side, *PowerFrame* provides several mechanisms for organizing and maintaining design data. The basic abstractions are *workspace, filter,* and *configuration*. The notion of workspace allows grouping of logically related data to form a unit of protection and consistency. Each member of a design project can separate his own data from others by forming a private workspace. At a higher level, a group workspace is formed as a synchronization point for data sharing. The filter mechanism takes a VLSI module and returns a selective part of the module. Filters allow users to concentrate on certain aspects of design data, e.g., a specific representation of a circuit. Configurations are aggregations that bind together components of a design entity. This is particularly useful for maintaining VLSI design data, where complex representations and hierarchical structures are ubiquitous.

Although *PowerFrame* seems to achieve the goals of tool encapsulation and navigation by providing a coherent view towards the underlying CAD tools, there is a missing

link between its process management facilities and data management services. In particular, the design history is not maintained and therefore object versions are not tied to the operations that create them. Moreover, the system doesn't offer any assistance beyond script-like automation/navigation. As interesting designs require more interactive decision-making than running routine scripts, it is essential for a process-support system to support higher-level design activities such as trial-and-error and/or iterative refinement. *PowerFrame* also lacks the capability of exploiting the power of distributed computing environments.

2.2.2 Automatic Design Manager -- *VOV*

The initial motivation of *VOV* [CASO90] was to assist novice VLSI designers to use the CAD tools in the Berkeley OCT [HARR86] environment. The central abstraction of *VOV* is the *trace,* a history of design operations performed by circuit designers. Figure 2.2 shows an example trace, where shaded rectangular boxes represent a tool invocation and square boxes represent files involved in a tool invocation. Design flows from top to bottom in Figure 2.2, i.e., the files above a tool invocation are inputs while those below are outputs. Different icons are used to represent alternative types of files, e.g., a square with a cross represents an executable file while a square box with a circle represents a normal UNIX file, and so on.

Traces contain the data dependency relationships between design objects and the tool invocations involved in creating them. When an object is modified, an automatic *retracing* facility consults the trace database to infer the affected set of objects. It re-runs the associated tool execution sequences to regenerate derived objects, keeping the consistency among objects that are related by the derivation relationships. During retracing, objects are updated in place. Portions of a trace can be used as example tool execution sequences, which novice circuit designers can imitate to accomplish certain tasks. *VOV* provides various data services such as concurrency control to guard against accidental overwrites, and *measurements,* which serve as the basis for validation and high-level performance estimations. For example, designers can use delay measurements of a layout

object to determine the parametric values of a place_and_route tool invocation. Finally *VOV* has the capability of transparently dispatching tool executions on a network of workstations.

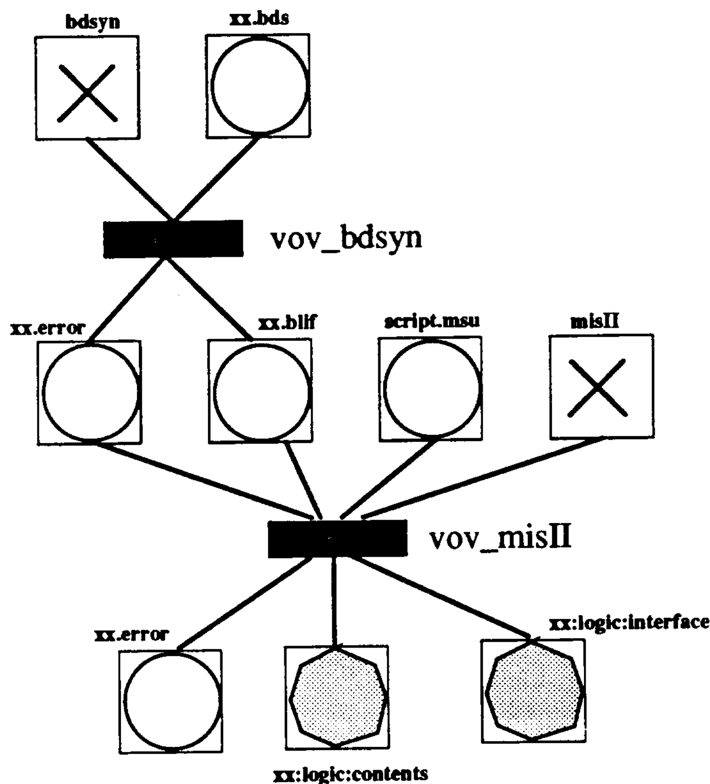


Figure 2.2 An Example Trace of a VOV's Session

The main drawback of *VOV* is that it assumes a centralized-server architecture, where all traces of all members of a design project are collected in a flat (i.e., non-hierarchical) database. In other words, there is no structure built into a project's traces. When traces grow as a project evolves, this architecture not only makes it difficult for the system to manage traces but also discourages users from browsing traces interactively. The philosophy of *VOV* is learning from examples, and *VOV*'s author argues against *PowerFrame*'s script-like templates. The reason is the claim that no innovative designs can come out of invoking pre-defined scripts only. However, even *VOV* later provides example traces to help novice users. On the other hand, traces are little more than

automatically-generated UNIX make files. Although retracing facilitates consistency maintenance, it is not clear how *VOV*'s traces address other design process management problems such as tool encapsulation and navigation, and support for high-level design activities.

2.2.3 Tool Execution Control Systems at Carnegie Mellon

Groups at Carnegie Mellon University have built several generations of CAD tool execution control systems, most of which are based on artificial intelligence techniques such as the *blackboard* model. A blackboard is a global database that maintains a set of facts against which rules can be matched and applied. An inference engine matches rules with the facts. If there are multiple matched rules, the inference engine chooses one of them and fire it. New facts can be added to a blackboard as a result of firing matched rules. The goals of these projects were two-fold: automatic tool execution and tool integration. The former aims at automating design work beyond the level of individual CAD tools, while the latter focuses on providing a flexible framework for adding and deleting CAD tools from a design environment without re-programming the environment and/or disrupting the users.

The first project that came out of this group, called *Ulysses* [BUSH89], modeled both CAD tools and circuit designers as *knowledge sources* (KS). In *Ulysses*, a knowledge source can post design goals to a global database called the *blackboard* to request services from other knowledge sources. On the other hand, every knowledge source continuously monitors the blackboard and volunteers to provide services when it detects that its capabilities match posted goals. Every knowledge source is equipped with the following information: *Precondition* file match patterns, *conflict resolution* parameters, and an execution method. The latter is the action performed by a knowledge source when it is chosen to execute.

Knowledge sources are activated by the presence of data on the blackboard that matches their preconditions. When multiple KS's are activated, a special KS called the *scheduler* ranking the capabilities of volunteering knowledge sources and chooses the

one most qualified. The ranking is done by considering the conflict resolution parameters of each KS such as the execution priority, required computation resources, and etc. After a KS is selected, the system sets up inputs and parameter options for invoking the chosen KS's action event, which is usually one or more tool invocations, thus isolating low-level details from the users. Because the scheduling of tool execution in *Ulysses* is based on individual tool's precondition matching mechanism, deletion or addition of CAD tools won't affect the rest of the environment, except the quality of the resulting designs. This is what *Ulysses* claimed as the open and distributed mechanism to integrate heterogeneous CAD tools.

Ulysses provides a script language for describing *knowledge sources*, *design tasks*, and *consistency maintenance rules*. Knowledge source specifications contain the same information described above. A design task is intended to guide the actions of knowledge sources and are interpreted by the *scheduler*. Design tasks specify procedurally the sequences of actions to be performed. Some of the actions can be performed by directly invoking tools, and the others can be posted to the blackboard as design goals for other knowledge sources to take over. The script language allows common control constructs such as if-then-else, for-loop, while-loop, and etc. A task can be composed of other subtasks.

Consistency maintenance rules serve the purposes of handling exceptions and cleaning undesired side effects. Specifications of consistency maintenance rules are similar to design tasks. Whenever a tool fails to satisfy the constraints (or goals), certain consistency rules are triggered to perform backtracking, make local modifications to the design, and retry the tool. Rules can also be used to ensure consistency among design components when local modifications made on one component affect the others.

Cadweld [DANI89] extends *Ulysses* by providing a class hierarchy for modeling CAD tools. Each CAD tool is modeled as a *knowledge object*, which is specified in terms of a *frame body* and a *control body*. A knowledge object's control body is similar to a knowledge source's precondition file matching pattern in *Ulysses*, which specify the conditions under which the knowledge object can be activated. A knowledge object's frame body contains tool invocation information such as the location of the binary file, the host

machine, the input/output file type, etc., and corresponds to the action sequences in *Ulysses*'s knowledge sources. The following is an example frame body specification. Detailed descriptions of its semantics can be found in [DANI89].

CLASS: PROCESS-SIMULATOR
USER-NAME: "FabricsII"
HOST: TUCANA
PATH: "/usr/jbb/bin"
USER-INTERFCAE-TYPE: TTY
MAN-PATH: "/../maxwell/usr/eecad/man"
DESCRIPTION: None
COMPUTING-EFFORT: 90
INTERACTIVE: No
PRIORITY: 10
AGE-TYPE: STANDARD
INPUT-TYPE: ".fin"
OUTPUT-TYPE: ".spin"
OUTPUT-VIEWER: None
TYPE:STATISTICAL

A tool class hierarchy allows sharing of the control body among CAD tools with similar capabilities. Both *Cadweld* and *Ulysses* adopt a rule-matching paradigm for automatic tool selection, which shields the impact of tool addition or deletion from the users. High-level design-specific knowledge is embedded in scripts to control the exploration of design space.

It is unclear how this knowledge is derived in the first place (e.g., capability ranking of multiple volunteering CAD tools), how effective this knowledge is in real designs, and how far this automatic design exploration paradigm can go. Furthermore, these systems do not assume that human designers play central roles in the design process. In fact they are treated as just another knowledge source. This is in contrast to our philosophy: we

believe that designers are the ultimate control of the design process, so the design system should aim at providing a friendly environment for exploring the design space, rather than embedding heuristic rules to control the exploration process.

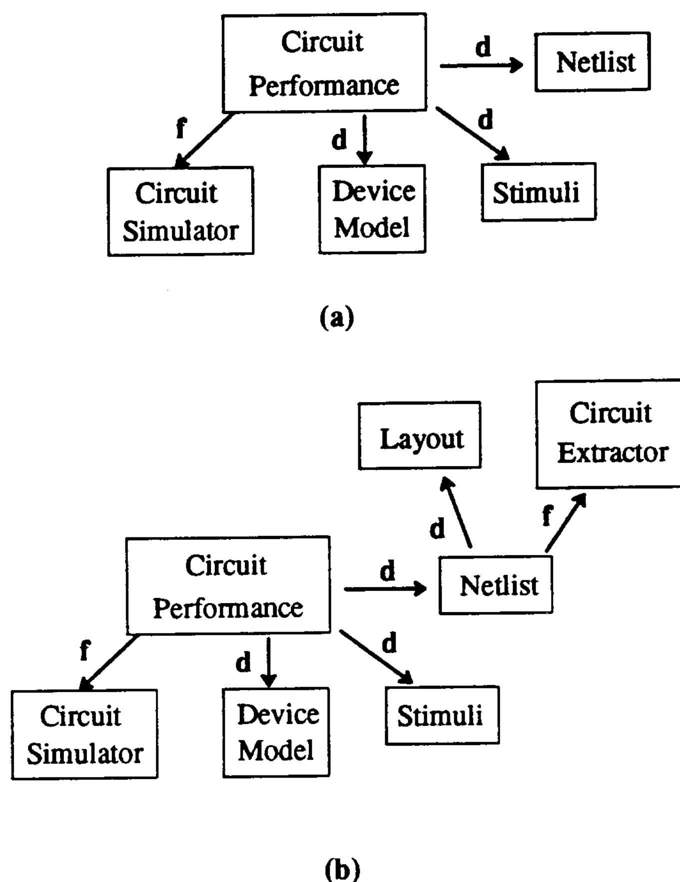


Figure 2.3 Task Schema and Task Derivation

A recent system from the same group, *Hercules* [BROC91], abandons the blackboard architecture and instead takes a procedural approach. *Hercules* proposes the concept of *task schema*, which is another form of tool execution template. Each task schema has a *target entity*, which is the intended output when the task is instantiated, and a set of *support entities*, which are either inputs or executables. Figure 2.3(a) shows an example task schema, where circuit-performance is the target entity, while circuit-simulation, device-model, stimuli, and netlist are support entities. The relationship between target

and support entities are represented by directed arrows from targets to supports, with annotations that specify the nature of support entities. An "f" denotes an executable support entity while a "d" a data file. Users run tools through task *instantiation*, which consists of four steps. The first step is called *task derivation*, which composes complex task schema by connecting primitive task schema, as shown in Figure 2.3(b). In this case, a circuit extraction task schema is combined with a circuit simulation task schema. *Hercules* provides a graphical interface for composing task schema. The second step is *resource management*, which fills in actual arguments (i.e., file names) for the entities in the derived task schema. The third step is *task execution*, in which the system traverses the derived task schema and invokes corresponding tools. The last step is *storage*, in which the resulting instance of the target entity is actually stored. It seems to us that the idea of task derivation would put unbearable burdens on the part of the users, and the concept of task schema really didn't offer any more modeling power than a vanilla script language.

Another group at CMU [VIDO89] (Integrated Design Environment : IDE) used similar blackboard models to address tool integration and management. IDE is also based on the notion of a design task, which is specified as a set of design goals and design constraints. A design task is mapped into a tool execution sequence called a design process. This mapping is either by an automatic planner or manually by circuit designers as a fall-back mechanism. A design process is a directed graph, where nodes represent tool invocations and arcs represent flowing data. Internal details of a design process is transparent to the users. The other feature of IDE is its emphasis on distributed and parallel implementation. The idea is to build the entire system on top of a parallel programming environment, harnessing the increasing horsepower of distributed computing environments.

In summary, most of the CMU systems take an "intelligence-embedding" approach in which domain-specific and application-specific domain are exploited to help designers beyond the design task level. Our position is that if something can be specified in advance, either in terms of procedures or rules, it probably can also be embedded into tasks. Moreover, there is always a limit on how much knowledge the system can exploit.

Beyond that limit, the CMU systems are completely ineffectual. It seems to us that as far as design process management is concerned, these systems simply address the wrong problem because of the ignorance of the interactive nature of VLSI design. For example, almost no effort is spent in coupling design data and process management. To be fair, the CMU systems should be better viewed as descendants from their high-level synthesis systems, rather than a true standalone design process management system.

2.2.4 Distributed Design Methodology Management

Microelectronics and Computer Corporation's (MCC) CAD Framework Methodology Management System (MMS) [ALLE90] provides users with a task abstraction and a distributed tool execution mechanism. A task is a sequence of tool executions written in LISP functions. Therefore it can take advantage of all the language features of LISP. Circuit designers interact with tasks rather than individual tools. A task engine interprets the specifications in the invoked task templates and communicates with the Process Control Server (PCS) facility to determine initial process placement for load balancing. Moreover MMS provides specialized communication services on top of the Remote Procedure Call facilities offered by the underlying operating system. Processes on any supported host, enabling multiple processes to cooperate in a distributed manner while being isolated from the underlying network transport mechanism. This facility makes it easy to develop parallel programs on a heterogeneous computer network.

Like *PowerFrame*, MMS doesn't provide any support for high-level design activities. The coupling between design data and design process management appears to be weak. In addition, because MMS only provides initial process placement as opposed to process migration, long-running tool executions could still occupy a machine even when its owner returns and wants to reclaim the machine. In this case, either the tool execution is aborted or the machine's owner is forced to tolerate degraded performance.

2.2.5 IDEAS

AT&T Bell laboratories developed an electronic design framework called IDEAS [MEHM87] [TAYL87], which provides mechanisms for description, control, documentation, and automation of a design process. IDEAS consists of two components: a design data tracking module (DDT) and a design methodology management module (DMM). DDT in turn is built on two abstractions: *design state* and *design file tracking*. Design states allow designers to have a comprehensive view of design versions and data management. The design file tracking mechanism records tool invocations and their inputs/outputs to keep track of the derivation relationship among design files and the associated tool executions. Facilities are provided to allow designers to query the design process history and the dependency relationships among design files.

The design methodology management system provides a hierarchy of process abstractions to describe various granularities of a design activity. In particular, a *design thread* is a description of a generic methodology for accomplishing a certain goal, just as a task in previous systems. A run-time facility, called the process specification agent, takes design thread specifications and guides users to customize design threads to the requirements of specific designs.

IDEAS is probably the closest work to *Papyrus* in terms of its mechanisms and software architecture. The only defect of IDEAS is the lack of a clean conceptual model, which makes IDEAS more a set of loosely-coupled mechanisms than an integrated environment. In addition, it doesn't address the issues of group work and distributed computation.

2.2.6 Petri Net-based Systems

Because of their superior capability for describing parallel and concurrent activities, Petri Nets have been used in several systems as the central formalism for design task description. Among these systems, *Monitor* [JANN85] is probably the earliest work. A Petri Net is basically a bipartite graph, as shown in Figure 2.4, with two types of nodes: *places* and *transitions*. Tokens flow on the nodes via the following rule: A transition T is

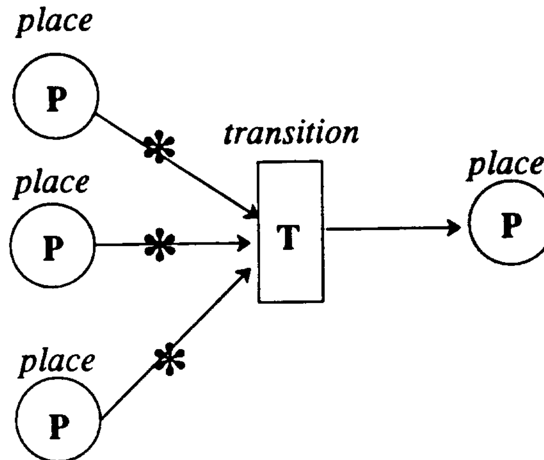


Figure 2.4 Generic Petri Net Formalism

enabled if and only if all input places to the transition hold one token and all output places hold no tokens. *Monitor* shows how this formalism can capture the data dependency relationships among tool executions that constitute a high-level task. It also provides a graphical user interface for users to specify and interact with the Petri-Net design process specifications. Another system [BRET90], developed at Siemens uses a more refined version of Petri Net called Predicate-Transition Petri Nets. In this formalism tokens can assume arbitrary types in the form of n-tuples over values from n domains (most frequently used domains are strings, integers, reals, etc.). Typed tokens allow a generic Petri Net to be used in different instances that share an identical structure but different types of data objects. Combining this formalism with rules, the system is able to make complex decisions based on the current state of the design. As a result, both static data dependencies in the form of tool sequencing, and dynamic decision-making in the form of backtracking and conflict resolution, are supported within a single framework.

The drawback of these systems is that they are pre-occupied with abstract models for representing CAD tool invocations and neglect the fact that there is more to design

than invoking tools such as managing design data. In addition these systems implicitly assume that all the operation sequences involved in a design process can be specified in advance. Consequently the templates in these systems are generally flat, and no composition operators are provided that could be used to organize design operations to reflect a particular design's development process. And again, no effort is spent in integrating design data and process management.

2.3 Software Engineering Environments

2.3.1 Process Programming

Process programming paradigm [OSTE87] advocates that software engineering activities, such as code development and maintenance, should be viewed as systematic processes that themselves should be described thoroughly and rigorously in program-like notations. In other words, programming itself can be described as a program. With this model, a run-time support system executes process programs by guiding programmers through the software development lifecycle according to the specifications. An important part of this paradigm distinguishes the activities performed by humans from those by automatic tools, and orchestrates the actions of humans, the support system, and tools. Activities involved in a software development lifecycle are neither completely mechanical and automatable, nor completely spontaneous and undefinable. Rather they are a mixture of these. It is believed [TBOW87] that this mixture can best be specified and communicated to the users by expressing it in a concrete form such as a process program. Unfortunately so far there has not been any significant systems that have been built and tested based on this paradigm. As a result, process programming remains an intriguing academic curiosity whose advocated advantages are still left to be validated.

2.3.2 Hewlett Packard's Tool Encapsulator

HP's SoftBench [FROM90] is an integrated software development environment in which a set of software tools such as editors, compilers and debuggers are integrated under a single framework. *HP encapsulator* is the tool integration and process specification facility of SoftBench. There are two mechanisms of *HP encapsulator* that are particularly interesting. *Encapsulator Description Language* (EDL) is a specification language that allows integration of non-native tools into SoftBench, and provides interfaces for encapsulated tools to make use of certain systems facilities in SoftBench such as the broadcast message server, which provides a multicast service to registered processes. A *tool trigger* represents cause-effect relationships between tools. A trigger occurs when a notification message is sent from one tool to its destination tool(s). Then one or more tools respond to that notification by taking certain actions. A trigger is actually an event-action pair that is specified in EDL. Therefore EDL not only allows customization of the appearance of each individual tool (e.g., the user interface part), but also supports tailorable project-specific management policies via trigger specifications. In summary, tool encapsulation consists of two tasks: interface customization and trigger specification, and they are both written in EDL. There is no process support mechanisms beyond tool encapsulation.

2.3.3 IBM's Programming Process Architecture

Programming Process Architecture (PPA) [RADI85] [HOFF85] [CHRO90] promises a framework for formally describing software development processes. In this framework, a software development process is decomposed into primitive process stages called *activities,* which represent one or more tool invocations. For each activity, the following are defined: (1) a list of Entry criteria that should be satisfied before starting the activity, (2) a set of Task descriptions that indicate *what* is to be accomplished, (3) a Validation procedure to verify the quality of the work items in the task specifications, (4) a checklist of eXit criteria that should be satisfied before the activity is considered completed. Based on this so called ETVX paradigm, the system provides process management operations

that capture and analyze product/process data in each process stage and give immediate feedbacks to the programmers. Early corrective measures, such as specifications modification, could be taken if errors are detected.

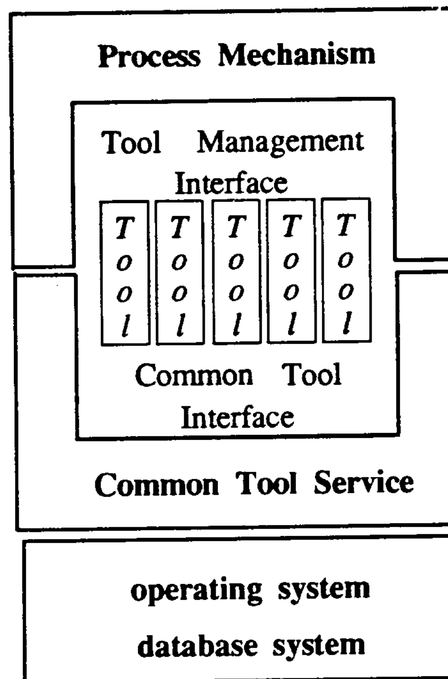


Figure 2.5 Architecture of the PPA System

As shown in Figure 2.5, the system consists of two components: a *process mechanism* (PM) and a *common tool service* (CTS). CTS presents a uniform data model and offers data access services to the tools in a development environment. It also includes a well-defined interface to the underlying operating systems. Tools built on top of this interface are independent of the underlying operating system and thus are portable across different platforms. A key feature of PPA is the separation of the design process from the products developed using the process. The PM provides a *process management language* for users to define the sequencing among activities. The PM encodes this process definition as a set of life-cycle control rules. During process execution, the PM accepts user requests for data and development tasks to be performed, and records

actions taken for later analysis. It accesses the process rules to determine if a request should be granted, based on whether the actions and conditions specified in the process as prerequisites to this request have been satisfied. If so, the PM either invokes appropriate tools through a *tool management interface*, to satisfy the request, or notifies the user to carry out the task. One implementation of this model is AD/Cycle [CHRO90].

Because PPA is really a software life-cycle support system, the proposed process model is too *large-grained* to be directly applicable to a VLSI design environment. For example, there is no support for choosing the right tools to accomplish a certain task. Also, the model as it stands, is heavily oriented towards pre-specified scripts for high-level management operations only, which doesn't necessarily ease low-level tool selection/invocation.

2.4 Office Automation Environments

2.4.1 POISE

POISE [CROF84] is an earlier system that aimed to reduce the overhead of office information flow by providing hierarchies of task descriptions. Task descriptions specify the steps in a task, the corresponding tool invocations, and their goals. POISE acts as an intelligent interface between users and tools in an office. Three types of information are used by POISE: the procedure library contains task descriptions; the semantic database contains the objects operated on by tools, and the tools themselves. A particular user's state includes partial instantiation of a task, parameters derived from user actions, as well as the objects involved.

POISE provides an *Event Description Language* to express a task's sequence of actions in a procedure. The language also allow specifications of preconditions for a task to start. Tasks can be nested to an arbitrary depth. The lowest level in this hierarchy corresponds to an individual office tool execution. Data is modeled in a semantic database model, which is based on a frame-based representation language. A frame is a named collection of attribute-value pairs called frame slots. Objects are mapped to

frames, with their attributes corresponding to frame slots. Composite objects are described via sets of frames. POISE can work in two different modes -- *interpretation* or *planning*. In the interpretation mode, users posted goals and POISE attempts to match the user's goals with task descriptions in the procedural library. Once a match is found, the corresponding sequence of tools are invoked. In the planning mode, users invoke procedures directly and POISE automates as much of that procedure as possible. By providing both a goal-based and a procedure-based descriptions of a task, POISE automates the operations within a task as well as provides navigation among task invocations. Unfortunately, the variety of operations and data representations in an office environment are relatively simple compared to those in VLSI design. As a result, the issue of highly complex tool sequencing and data control is not really stressed in POISE.

2.5 Summary of Previous Systems

Judging from the systems surveyed so far, we can safely conclude that the need of computer support for *processes* in addition to *data* is emerging as a major component of future interactive computer systems, in particular the VLSI design systems. This further confirms our assertion in Chapter One that next-generation VLSI design database systems should focus on the support for design processes as well as the interaction between design data and processes.

From these systems, we draw the following observations concerning the design issues of software systems that support processes.

- The first component of process support is a high-level abstraction that encapsulates primitive steps provided by the underlying environment, e.g., CAD tools in VLSI design. In most systems, this abstraction is called a *task*. Tasks address the issues of tool encapsulation and tool navigation at the same time.
- Beyond the task level, it is desirable to provide a facility that could allow domain-specific expert knowledge to be exploited in process support systems, thus offering scenario-dependent assistance beyond script-oriented tasks. The systems surveyed implemented this facility through mechanisms ranging from blackboard rules,

predicate-transition Petri Nets, to goal-driven template matching.

- Since most future software systems, if not now, will be based on a group of computers connected through a local area network, it is essential to effectively exploit the computational resources in this type of environment. For computation-intensive engineering design environments, this issue will assume increasing importance.
- One serious drawback of all the systems surveyed, except IDEAS [MEHM87], is the level of integration between processes and data. After all, to invoke an operation requires selection of the right tools AND the right data subset. To genuinely facilitate tool execution, a process support system should help users track the evolution of data as well.

In Table I, we summarize the systems reviewed according to the functional requirements listed in Chapter One. *Papyrus*, as a process support facility for VLSI design, fulfills all the functional requirements and can be viewed as a meta-tool for making more efficient use of the CAD tools. Early on in the project, we decided that user-friendliness is the predominant concern. As a result, rather than drawing various complex mechanisms that handle specific scenarios, we put simplicity as the first consideration in the design of the conceptual model that guides the implementation of *Papyrus*. In the next chapter, we introduce *Papyrus*'s conceptual model. It combines some of the mechanisms of previous systems with several new ideas to form a coherent substrate for supporting VLSI design process model.

Characteristics Summary of Process Support Systems							
System Name	*Tool Encapsulation*	*Tool Navigation*	*Design Exploration*	*Data Evolution*	*Context Management*	*Cooperative Work*	*Distributed Architecture*
Powerframe	Yes	Yes	No	No	Yes	No	No
VOV	Yes	No	No	No	No	Yes	Yes
Ulysses	Yes	Yes	Yes	No	No	No	No
Cadweld	Yes	Yes	Yes	No	No	No	No
Hercules	Yes	Yes	No	No	No	No	No
IDE	Yes	Yes	Some	No	No	No	Yes
MMS	Yes	Yes	No	Yes	No	No	Yes
IDEAS	Yes	Yes	No	Yes	Yes	No	No
Monitor	Yes	Yes	No	No	No	No	No
Siemens	Yes	Yes	Some	No	No	No	No
SoftBench	Yes	Yes	Some	No	Yes	No	No
PPA	Yes	Yes	No	No	No	No	No
POISE	Yes	Yes	Some	No	No	No	No
Papyrus	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table I Comparison of Process Support Systems

Chapter 3

The Conceptual Model

3.1 Introduction

Engineering design is generally characterized by two types of activities: *exploration* and *cooperation*. Designers usually explore various alternatives before settling down a particular design decision. Design exploration typically takes the form of creating versions of objects, and choosing the best subset of object versions according to certain criterion. As the design complexity increases, keeping track of the correspondence between object versions and design decisions, which we call *version mapping*, becomes a difficult problem in practice. To address the version mapping problem, the system should maintain a separate "context" for each design decision, and allows users to visit these contexts without being confused. For conceptual clarity, the notion of context should be tied to the processes that create contexts.

Engineering design is also almost always a group effort. Designers cooperate by sharing data, e.g., they assemble components and reuse common modules. Uncontrolled data sharing could lead to inconsistency due to un-coordinated updates to shared objects.

There is a consensus in the database research community that two-phase locking[1] is inadequate because of the long lifetime of these so-called *long-lived transactions* (as distinguished from *database transactions* in commercial data processing, whose lifetime is relatively short.), and hence the degraded concurrency. A better approach, particularly in an interactive design environment, is to take an optimistic concurrency control approach and to make it relatively easy to resolve conflicts when they arise.

If one models a CAD tool invocation (called a *design step* hereafter) as an indivisible operation against a shared database (e.g., a file system), then it is natural to model a sequence of design steps as a conventional database transaction. Various models [BANC86] [KAIS90] [PUKA90] [WIDY86] have been proposed that either provide complicated transaction structures or relax serializability[2], with a view to enhancing concurrency among long-lived transactions. However, a closer examination reveals that certain properties of engineering design activities simply preclude the application of database transactions. Notable among them are *interactivity*, which implies that it is impossible to confine a designer's data access behavior to a particular pattern, e.g., locking objects in two phases, and *open-endedness*, which means that there is no commit operation as in database transactions.

In the course of developing a process model to support engineering design, it gradually becomes clear that the model should stand at a higher level of abstraction than database transactions. Rather than maintaining data integrity and enhancing concurrent data access, the design process model is supposed to support design exploration and cooperative work. For that matter, even long-lived transaction models are not appropriate. On the other hand, we find that the concept of *data visibility* can adequately support both design context maintenance and cooperative data sharing. Data visibility is the abstraction that controls whether a piece of data is visible to the outside world. Based on this abstraction, we develop a so-called *light-weight transaction* model (LWT), which

1. In a two-phase locking scheme, a transaction can't acquire more locks after it starts to release locks.

2. An execution schedule of concurrent transactions is serializable if the effect on the database is as if these transactions are executed in some serial order.

provides a set of operational constructs for users to manage the visibility of the data objects *interactively* and *dynamically*. Just as a light-weight process doesn't carry as much state as a normal process, a light-weight transaction doesn't provide as much integrity guarantee as a normal transaction. Specifically the light-weight transaction model provides constructs for controlling data visibility rather than guaranteeing data integrity. Moreover, a system based on the LWT model is built on top of a conventional database transaction facility that enforces all-or-nothing atomicity at the physical storage level. In our case, since a design step, which corresponds to a UNIX process, is modeled as an indivisible operation, the LWT system delegates the concurrency control and error recovery issues within a tool execution to the underlying design database management system, i.e., OCT. As shown in Figure 3.1, concurrent accesses to the design database are controlled by OCT, while users interact with a light-weight transaction system to manage their design process.

3.2 Basic Assumptions

An *object* is a uniquely identified piece of data, whose identifier is known to the users. An object can contain other objects. Updates to an object follows a *single assignment* semantics. Modifications to an object are not performed in place, but made to a copy of the original object, thus creating a new version of the object. Version numbers are managed by the system, which automatically provides a new version number whenever a new object version is created. Under the semantics of single assignment update, an update can be viewed as a transformation, with the old version as the input and the new one as the output.

The fundamental principle of the LWT model is that *visibility dictates accessibility*. Users can only access those objects that are visible to them, and visibility can change depending on the context from which a design operation is issued. The system enforces access control by properly managing the data visibility to individual users. When an object is accessed, the system verifies whether that object is visible to the user that issues the access operation.

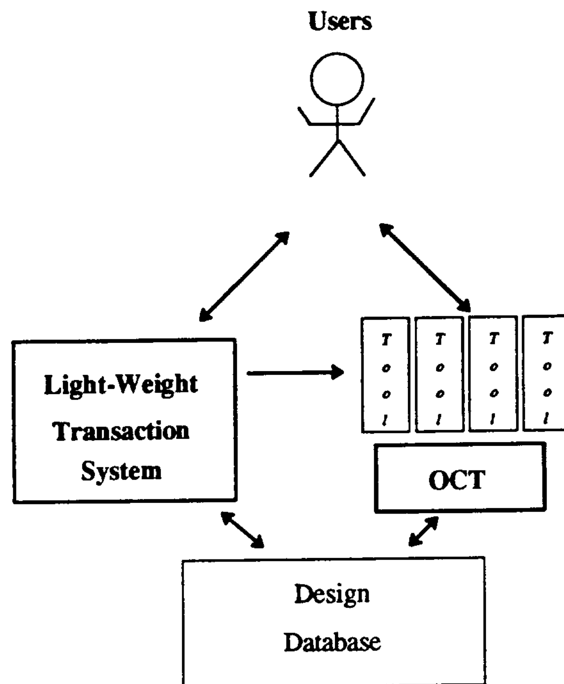


Figure 3.1 Relationship Between a Light Transaction System and Its Underlying Database Management System

3.3 Light Weight Transaction Model

As in previously proposed models, our Light Weight Transaction (LWT) model provides a hierarchy of operational constructs for users to manage their design process. As shown in Figure 3.2, there are three levels in this hierarchy, each of which is intended for a specific purpose and provides a different set of visibility control operations. *Papyrus* is an implementation of the LWT model. Briefly, a *design step* corresponds to an individual CAD tool invocation. A *design task* provides users with a higher-level abstraction than primitive CAD tool executions, thus serving the purpose of tool navigation, automation, and encapsulation. A *design thread*, consisting of a set of design tasks and the data objects involved in the tasks, is meant to be the focal point for collecting the design operations and objects related to a logical design entity, e.g., an arithmetic logic unit (ALU). Design tasks automate routine design work while design threads support

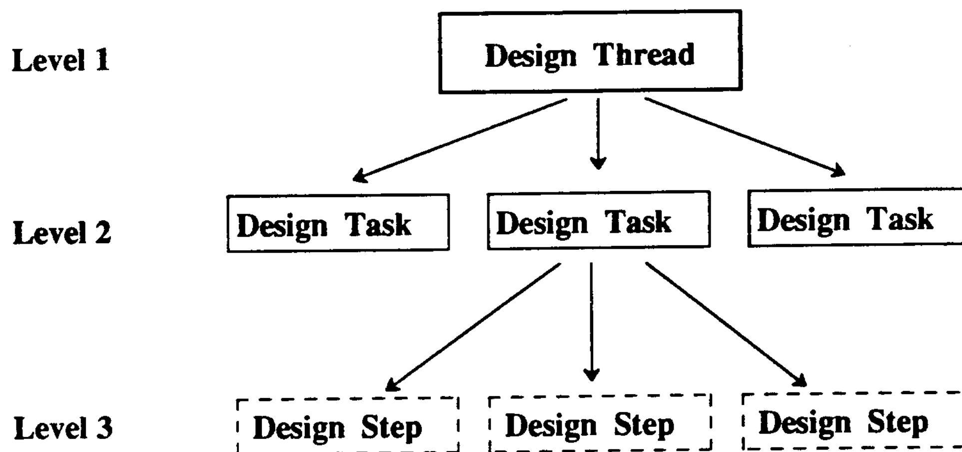


Figure 3.2 The Light Weight Transaction Hierarchy

interactive exploration of design space.

3.3.1 Design Step

The lowest level in the hierarchy is called a *design step,* which corresponds to a VLSI CAD tool invocation, e.g., an execution of a Boolean logic minimizer that optimizes combinational logic circuits. A design step is the most primitive unit of action. Although there may be many database operations within a tool invocation, it is assumed that the underlying design database system could guarantee concurrency and failure atomicity. By separating the management of physical database transactions from that of design processes, both systems implementation and user conceptualization are greatly simplified. In addition, because of single assignment update semantics, the output of a design step is a new version of the output object. Moreover, a completed design step can be aborted by deleting the corresponding output versions. Using the data visibility abstraction, *Papyrus* "deletes" objects by making them invisible. A garbage collector is running in the background to reclaim "deleted" objects that haven't been "undeleted" for a specific time period, at which time the objects are physically deleted.

3.3.2 Design Task

The next level up in the LWT hierarchy is called a *design task*. Intuitively, a design task is a parallel shell script that provides an atomicity guarantee. We develop a *Task Description Language* based on Ousterhout's Tool Command Language (Tcl) [OUST90], whose syntax will be described in Chapter Four. This language allows composition of a high-level abstraction from primitive CAD tools.

The Task Description Language has two features that distinguish it from other script languages. First, it provides linguistic constructs for expressing parallel tool executions. Second, the language allows specifications of the abort semantics tailored to an individual task. When a design task is aborted on a particular design step, it doesn't necessarily mean that the side effects of the entire task have to be removed. Instead users can specify a default *resumed task state* for each abortable design step in the task.

Specifications written in the *Task Description Language* are called *task templates*, which are usually specified by expert circuit designers or systems managers. In other words, most circuit designers don't need to write task templates. A graphical representation of an example task template is shown in Figure 3.3(a). A fork at **a** means that both **step1-step2** and **step3-step4** sequences could be executed concurrently after **step0.** A join at **d** means that both **step1-step2** and **step3-step4** sequences must be completed before **step5** could start, i.e., a barrier synchronization.

The design steps in a task template can be either batch-oriented, e.g., a two-level logic minimizer, or interactive, e.g., a schematic editor. Users are committed to follow the execution order specified in a task template once they invoke the task. Design tasks serve the dual purposes of *tool encapsulation* and *tool navigation.* Casual users only need to interact with task-level abstractions, and the system can lead them through the detailed execution sequences. Low-level invocation details are completely shielded from the users, including transparently dispatching concurrent tool execution in parallel. Design tasks are also useful in enforcing high-level design methodologies that a design group or company wants to impose on the design process.

Papyrus maintains a detailed design history down to individual design steps. An instantiation of a design task leaves a *history trace* of the form shown in Figure 3.3(b).

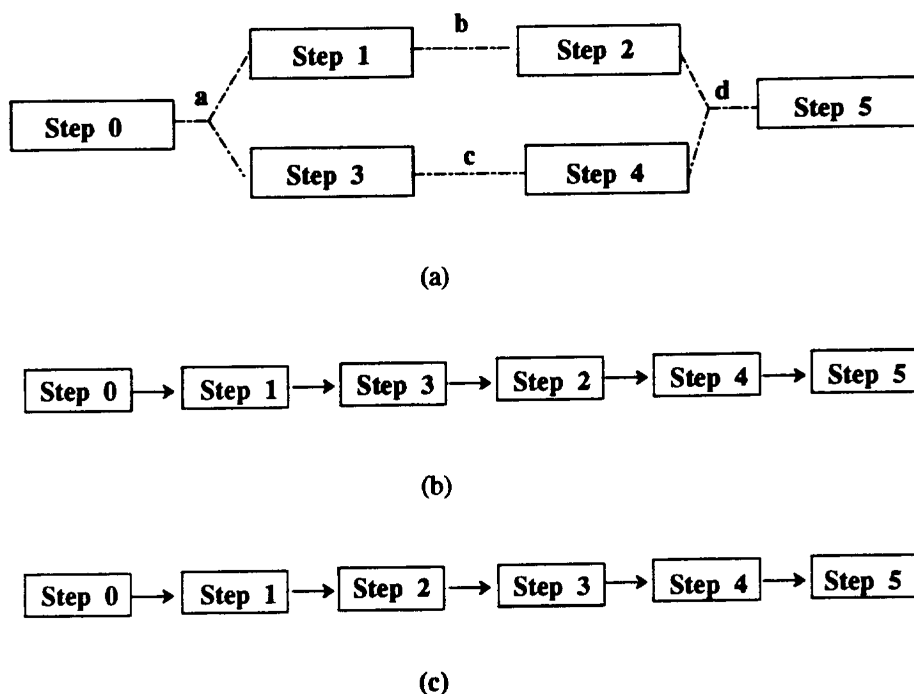


Figure 3.3 A Task Template and Its Corresponding History Traces

The linear ordering of design step invocations in a history trace is determined by their "completion" times. Therefore although more than one step can run simultaneously, they can be still ordered according to their completion times. Different invocations of the same task template may leave different traces as shown in Figure 3.3(b) and Figure 3.3(c), both of which are legal with respect to the template in Figure 3.3(a). Between two consecutive design steps in a task's history trace marks a *task state*, which is defined by the set of design objects that are referenced as inputs and created as outputs from the start of the trace up to that task state.

A design task is said to *commit* when every design step of the task completes successfully. However, users don't need to explicitly issue commit operations. Only the specified input and output objects of a committed task invocation are made visible to the enclosing environment. The other objects created during a task instantiation are treated

as temporaries and discarded after commit. A task can be aborted by the users or by an error from the environment. If a design task is aborted at a particular design step, the task is restarted from the step's corresponding *resumed task state*. If the design step doesn't have an associated resumed task state, the task is restarted from the beginning by default, which is the abort semantics for normal database transactions. With this construct, *Papyrus* allows users to preserve useful work by exploiting the semantics of the design task in question.

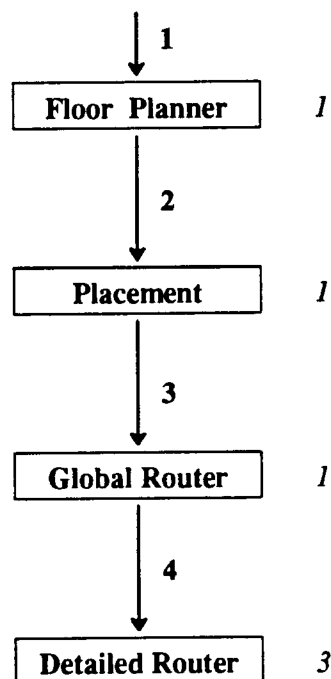


Figure 3.4 The Concept of Resumed Task State

For example, consider a long-running multi-step VLSI placement and routing task that consists of a floor-planning step, a placement step, a global-routing step, and a detailed routing step, as shown in Figure 3.4. The labels on the arrows identify particular task states, while those next to the design steps designate their corresponding resumed task states. Suppose this task is aborted at the detailed-routing step because of insufficient

routing channels. Further assume that routing channels are assigned by the global-routing step. In this example, the *resumed task state* of the detailed-routing step is the state right after the placement step, but before the global-routing step. When the task is restarted after aborting on the detailed-routing step, the work performed before the global-routing design step are preserved, and users can try different parameters with the global-routing step to avoid breakdown of the following detail router.

A design step S's resumed task state is specified by its *resumed step,* from which the resumed task state of S immediately follows. For example, in Figure 3.4, the resumed step of the detailed-routing step is the placement step, whose following task state is the detailed-routing step's resumed task state. Note that the abort semantics of a design step may depend on the run-time conditions. In Figure 3.3.(a), the effects of both **step1** and **step3** have to be undone (by deleting the objects they created), although the latter does not interact with **step2**. Had the task history trace been Figure 3.3(c) rather than Figure 3.3(b), only **step1** needs to be undone. In any case, *Papyrus* restarts a task in a manner completely transparent to the users.

According to the task restart semantics, not every design step is qualified as a resumed step for an arbitrary design step in the same task. A design step S_1 is eligible to be the resumed step of S_2 if and only if S_1 is a *logical predecessor* of S_2. A step S_1 is a logical predecessor of S_2 if there is a directed path from S_1 to S_2. For example, **step1** is the logical predecessor of **step2** but **step3** is not. The Task Description Language also allows specifications of *dynamic* resumed task states, which are not specified in terms of a resumed step. A typical example specifies the most recent task state as the resumed task state, which corresponds to the abort semantics of the nested transaction model if one views a design step as an atomic transaction. Because of the potential parallelism within a design task, the latest task state is a run-time variable and cannot be determined statically.

In summary, users *instantiate* task templates to carry out design procedures that are well-defined and can be pre-specified in advance. To support high-level task abstractions, internal side effects of a task instantiation should be completely transparent to the users, even when a task is aborted. If a design step is aborted, the enclosing task is restarted

from the step's corresponding resumed task state. Before restart, the side effects of the steps that are both the logical successors of the resumed step and the logical predecessors of the aborted step are undone by deleting the objects created by these steps. Other steps are also aborted whenever at least one of their logical predecessor is aborted. This undo process iterates until it converges, at which point the task is ready to proceed.

3.3.3 Design Thread

Organization of A Thread

Just as every UNIX tool is invoked with respect to a particular file directory (the current directory), a design task is instantiated with respect to a particular *design thread.* A design thread consists of a set of design tasks that are instantiated in the thread, and the data objects involved in task instantiations. While design steps and design tasks are pre-specified, design threads are *open-ended* in that users create and manipulate them interactively and dynamically. Users instantiate design tasks in any desired order, and *Papyrus* maintains the history of task instantiations according to their temporal *completion* order. A design thread is not just another level in the LWT model hierarchy. It is introduced specifically to embody the notion of a design entity's *context.* While design steps and design tasks facilitate tool invocations, design threads make it easy to find the right design objects to operate on.

Three state elements are associated with a design thread: a *thread workspace*, a *control stream*, and a set of *frontier cursors*. A thread workspace consists of the set of objects referenced as inputs and created as outputs in the design thread. Because of the single assignment update semantics, the thread workspace grows with the progression of a design thread. The control stream of a design thread refers to the structure and sequencing of already committed design tasks. Figure 3.5 illustrates the control stream for a typical design thread, where each vertical bar denotes a *history record,* which encapsulates the history of a committed design task, including the invoking details of the task's design steps and the input/output arguments. Note that a design thread's control

stream can have a branching structure.

Each committed task in a thread defines a *design point,* denoted as an arrow in Figure 3.5, e.g., design point 2 is associated with the second history record. A design point has an associated *thread state,* which is defined as the set of objects referenced as inputs and created as outputs from the initial state of a design thread up to the completion of the record's corresponding task. The *frontier cursors* of a design thread are the set of design points that don't have a following history record. In Figure 3.5, both design point 3 and 12 are frontier cursors. The union of the thread states of a design thread's frontier cursors constitutes the thread's thread workspace. One of the frontier cursors is called the *current cursor,* which is the default design point to which the records for newly committed tasks are attached. The current cursor automatically advances when a new history record is appended to the control stream. The *visibility* rule dictates that *every invoked task can only access the objects in the thread state associated with the current cursor.* Intuitively the current cursor's thread state forms a default view into the database, much like the notion of current directory in a file system.

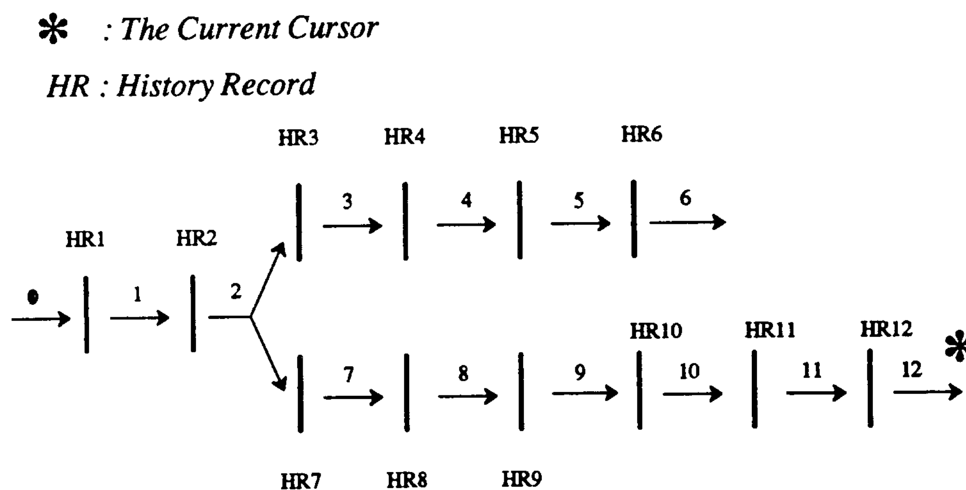


Figure 3.5 A Design Thread That Has Branching Structure

The Rework Mechanism

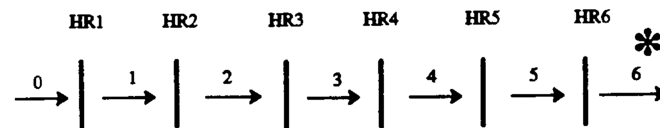
An interesting feature of the LWT model is the notion of *rework.* The rework mechanism allows users to move the current cursor to any existing design point in a design thread. Without rework, the committed tasks in a design thread are organized as a linear sequence, and the current cursor is by default the lastest design point. Rework allows users to override this temporal order by moving the current cursor to arbitrary existing design points and proceeding from there. Because by definition new tasks are invoked with respect to the current cursor's thread state, moving the current cursor to a previous design point effectively rolls the thread back to a previous state. This is similar to changing the default context in a file system by changing the current directory. When the current cursor is moved to an existing design point and subsequent tasks are instantiated, a new branch of the control stream is formed. This branch is conceptually independent of the existing branches: Objects created in the other branches won't be visible in this new branch, and vice versa.

Papyrus also allows users to selectively erase existing branches (and therefore their associated objects) as a side effect of the rework operation. For example, suppose Figure 3.6(a) is the original control stream, and a user moves the current cursor from design point 6 to design point 2 and chooses to erase the intermediate committed tasks, then a control stream of the form Figure 3.6(b) accrues. From the new current cursor, the user can start a new development path and construct a control stream shown in Figure 3.6(c). Had the user moved the current cursor to design point 2 *without* erasing the intermittent effects and started the same development path, the final control stream would have been Figure 3.5.

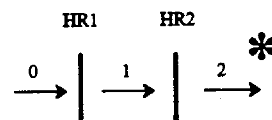
The rework mechanism is particularly useful for interactive design exploration, i.e., experimenting with different alternatives before committing to a design decision. Typically designers use this mechanism to roll back the status of a design to a previous state by moving the current cursor to the corresponding design point, and then experiment with different procedures or parameters. Consequently different branches of a control stream correspond to different approaches explored. Users can examine the results of these approaches by positioning themselves on the appropriate design points. The system

* : *the current cursor*

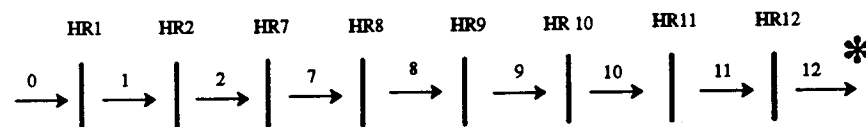
HR : History Record



(a)



(b)



(c)

Figure 3.6 The Rework Mechanism

maintains multiple non-interfering possible worlds and takes care of the mapping between an explored alternative and its related set of objects.

Figure 3.7 shows a snapshot of the control stream of a design thread whose goal is to synthesize a shifter. The numbers on the arrows identify particular design points and also signify their temporal sequencing. Initially the designer created a thread with a descriptive name, such as *Shifter-synthesis*. The thread state of the initial design point is empty. To create an ALU's logic description, the designer invoked a design task called *create logic description*, which actually consists of two steps: *enter-logic (Edit)* and *format transformation (BDSYN)*. Then the designer was lead through the task's operation sequence step by step and only needed to specify proper parameters and inputs/outputs.

Once the *create-logic-description* task completed, the designer invoked another task *logic simulator* to verify the design's logic behavior. Next the designer invoked the *standard-cell-place-and-route* task, followed by the *place-pads* task to generate a particular design. At this point the *current cursor* is at design point 5.

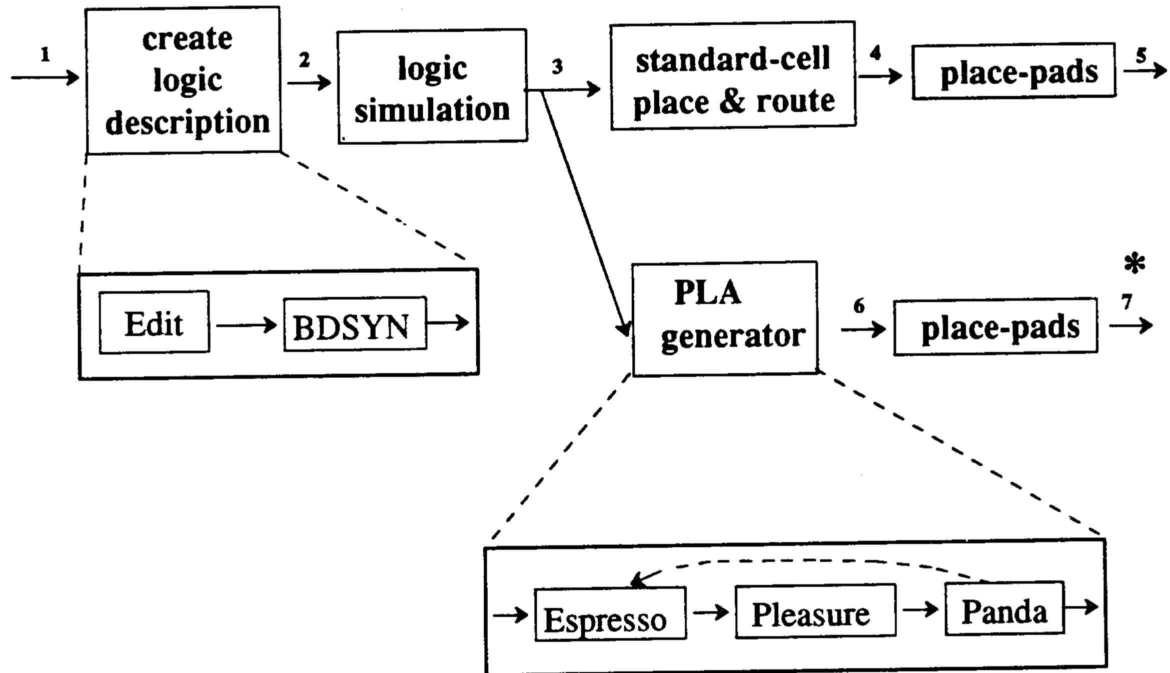


Figure 3.7 A Shifter-synthesis Design Thread

Suppose the designer was not satisfied with the result of the standard-cell approach and wished to explore another alternative, such as an implementation in a PLA design style. He could reposition the current cursor to design point 3, and invoked the *PLA-generation* task followed again by *place-pads*. Changing the current cursor restores the context back to the thread state associated with design point 3, which is the state before the standard-cell approach was explored. In other words, the designer can start with an identical context and explore different design procedures. Moreover, because objects created along one branch is independent of the others, the set of objects associated with a

particular alternative, e.g., the standard-cell approach in Figure 3.7, can be isolated and identified.

Note that the *PLA generation* task consists of three steps: two-level minimization (Espresso), PLA folding (Pleasure), and array layout (Panda). The dotted line from Panda to Espresso indicates that when Panda fails due to, say, a violation of the area constraint, the design task should restart from the state after the completion of Espresso, i.e., Pleasure should be re-executed. By providing programmable abort facilities, useful work, in this case the execution of Espresso, won't be wasted.

Using the rework mechanism, designers don't need to keep track of design versions at different design points: the mapping is automatically maintained. Moving the current cursor to explore alternative states of a design is conceptually simple and yet semantically powerful, especially when designers can browse the control stream and change the current cursor with a mouse-like direct-manipulation device. Most of all, it allows users to interactively change the course of a design development process without pre-planning the whole process in advance.

An alternative way to understand the rework mechanism is to view it as a *snapshot-taking* operation, as in temporal/historical databases. However, the rework mechanism is more general in the following senses. First, the history structure assumed by most temporal databases is a linear temporal order, whereas the control stream of a design thread can have arbitrary branching structures. That is, under user discretion the structure of a thread's control stream could be made to reflect the *causal* order of a design's development rather than a simple temporal order. Second, most temporal databases use "time points" as the access index (e.g., 7:00 PM in 9/11/91), while we use "design points" as the main access method. Coupled with a graphical interface, we believe the latter provides a powerful paradigm for traveling back and forth among various possible states of a design. Users are shown graphically the control stream of a design thread and allowed to move the current cursor around with a mouse. Just as desktop computing offers a spatial metaphor for structuring an office workspace, the "cursor" paradigm suggests an intuitive temporal metaphor for organizing a design activity. On the other hand, *Papyrus* does provide time-point oriented query interface for "long-range" time travel. For example,

users could request a design point corresponding to 4 PM, October 15, 1992. Third, with the rework mechanism users can go back to arbitrary previous states without pre-planning the snapshots, which again is a crucial advantage when designers are exploring the design space without concrete plans in mind. In the LWT model, since data objects never get deleted, taking a snapshot is a matter of maintaining associations between data objects and design points.

3.3.4 Interaction Among Design Threads

Design thread associates a context with a design entity, e.g., an ALU module. A designer is allowed to have multiple design threads active simultaneously, e.g., one for an ALU module and one for a register-file module, etc. These design threads evolve concurrently in an open-ended way. At the same time, other designers can have their own sets of design threads as well. In this section, the operations provided by the LWT model for manipulating individual threads and synchronizing concurrent threads are discussed.

3.3.4.1 Thread Manipulation

The LWT model provides a set of operations for users to manipulate a design thread as a first-class object. These operations are useful in re-organizing the structure of a design thread. In particular, they are designed to support the bottom-up design methodology: small-granularity design threads are combined to form larger ones as submodules are completed and integrated into a larger entity. With these combination operators, the granularity of a design thread can be arbitrarily small. For example, a design thread can be dedicated to the design of a register cell or a NAND gate. Thread combination operations in the LWT model are:

Cascade: The control streams of two design threads can be cascaded into one, as show n in Figure 3.8.

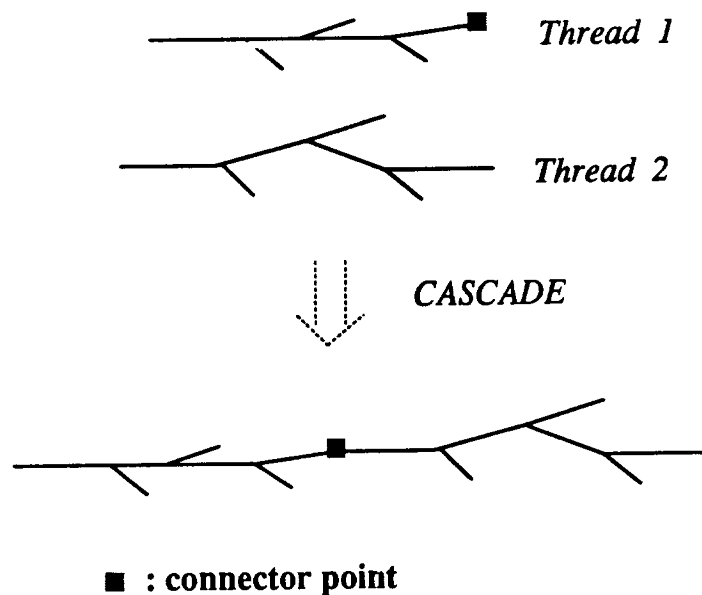


Figure 3.8 Cascading Two Threads Into One

Join: The control streams of two design threads can be joined at the head or at the end, as shown in Figure 3.9.

Fork: A new design thread can be created with its initial thread workspace inheriting from another design thread.

When a thread is created, its control stream, thread workspace, and frontier cursors are null. A thread can inherit its initial thread workspace from an empty set (the default), from a particular thread state or the entire thread workspace of another design thread. Semantically the newly created thread evolves completely independently of the inherited thread. Updates made to a shared object in the inheriting thread won't be visible to the inherited thread and vice versa.

The semantics of a thread combination operator is defined in terms of its effect on the thread's thread workspace, control stream and frontier cursors. Merging two design threads (both *cascade* and *join*) unions the two thread workspaces to form the resulting

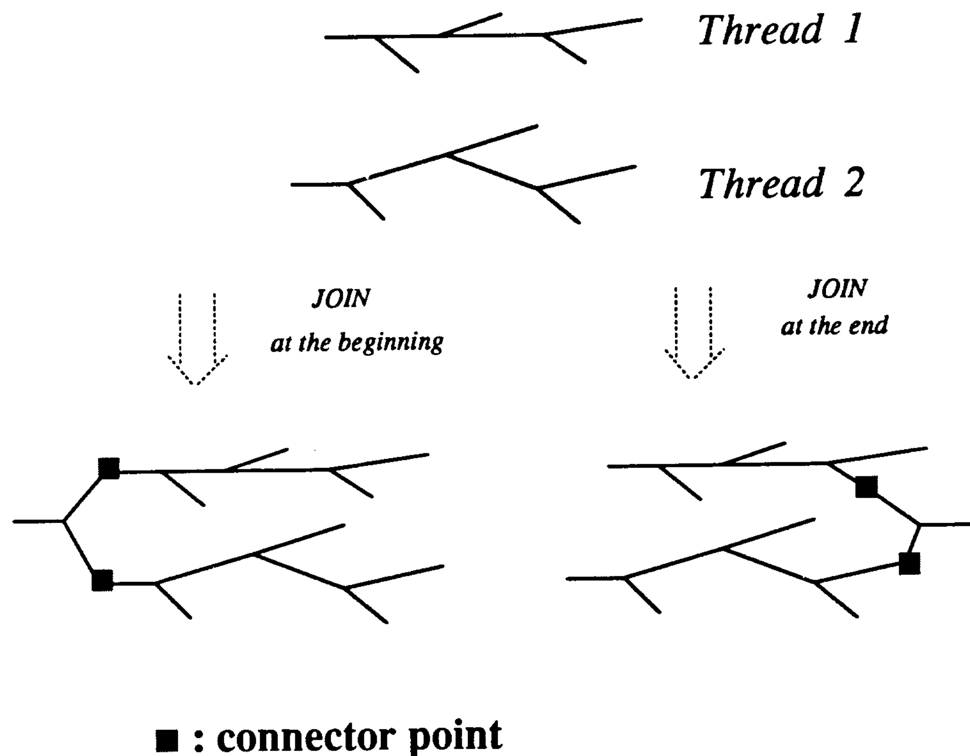
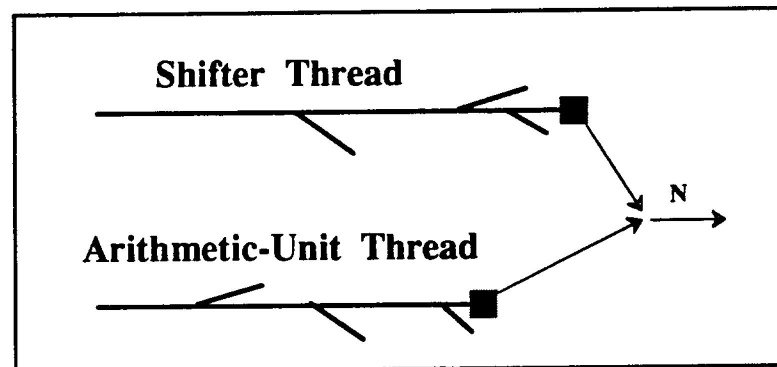


Figure 3.9 Joining Two Threads Into One

thread's workspace. Since it is a set-union process, duplicates are eliminated automatically. Because each control stream could have a branching structure, *connector design points* of two control streams from which merging takes place need to be specified by the users. Only frontier cursors can be used as connector design points. In the case of a *cascade,* only the connector point of the trailing thread requires specification, the connector point for the leading thread is its initial design point. The resulting thread's frontier cursors are the union of the frontier cursors of the two cascading threads except the connector point of the trailing thread. In the case of a *join,* connector design points of both threads need to be specified and they are combined into a new design point. The resulting thread's frontier cursors are the union of the frontier cursors of the two threads except the two connector design points.

These thread manipulation operations are useful for associating data objects with their derivation contexts. For example, suppose designer A is taking a long leave and designer B is supposed to take over A's unfinished work, simply making visible *what* A has been doing is usually not good enough. It is also necessary for B to understand *how* A has gotten to where he is now in order to continue A's work.

ALU Thread



■ : connector point

Figure 3.10 Merging Two Threads into One Thread

As an example for thread joins, suppose one designer is working on a shifter and another on an arithmetic unit. When both the arithmetic unit and the shifter are completed, these two efforts are to be merged and continued by one of the designers. As shown in Figure 3.10, these two threads can be joined at the end and form a new thread called the *ALU thread*. The control streams of these two threads are combined at the specified connector points. The workspaces associated with the two threads are unioned. Moreover, this combined thread works as if it had been created from the scratch. In particular, the designer can roll back to any design point in this new thread and modify the control stream in any way as desired. This capability of combining small threads into

larger ones supports the hierarchical bottom-up design methodology and facilitates cooperative team work. Note also that after merging, the original threads can continue independently of the new thread. Any modifications made on one thread won't be seen by the other, and vice versa.

3.3.4.2 Thread Synchronization

A design thread is an abstraction that integrates the concepts of file directory and operation history. Each thread workspace provides a name space that is protected against other threads. In other words, the objects in a thread are not visible to and therefore cannot be operated on by other users than the thread's owner. The LWT model does not provide explicit *thread commit* operations for two reasons. First, because design threads are meant to support open-ended and exploratory activities, *commit* operations are rarely used in practice. The second reason is that it is not clear what the effect of a commit operation should be in long-lived transactions. For database transactions, all objects updated or created by a transaction is made visible to the outside world at the commit time. In a long-lived transaction, this kind of commit semantics is inadequate because most of the objects created during the transaction are simply intermediates. Making everything visible only clutters the global database. The only sensible "commit" semantics for long-lived transactions is to allow users to make visible to the outside world *selective* portions of the thread workspace at *selective* times. The effect of a commit, which traditionally takes place at a particular time point for all objects created in a database transaction, is now distributed in time and is completely under users' interactive control.

Like a workspace, a *synchronization data space* (SDS) is a shared data repository that serves as a synchronization point for the threads that register with the SDS. With respect to a SDS, only the registered threads can contribute/retrieve objects to/from the SDS. A SDS's set of registered threads can be dynamically changing, so can new SDS's be created to accommodate an evolving development project. A thread can participate in multiple synchronization data spaces. The objects in a SDS never get updated; only new

versions are added. Objects belonging to a thread workspace will not be visible to the outside world until they are *moved* to a SDS. That is, data sharing can occur in SDS's. Note that it is a design thread that is the basic unit in thread synchronization rather than the user who owns the thread. This allows a finer granularity of control over data sharing, and makes the notification mechanism discussed later more contextually focused.

Transferring data objects between synchronization data spaces and/or thread workspaces is accomplished with a *move* operation, with a syntax of the following:

MOVE Object–ID, Source–space, Destination–space, Notification–flag, Predicate–set

A move of an object entails two operations: (1) physical copy from the source to the destination space; (2) if the source space is a SDS and the destination is a thread workspace, a notification flag is left behind. This is used to notify the corresponding thread when a new version of the object is added to the SDS. Users can choose to disable this flag when appropriate.

Because serializability is not an adequate criterion for concurrency control in a cooperative environment, there is no locking on objects in the synchronization data spaces to allow maximal concurrency. Instead, when a conflict arises, a notification mechanism sends a message to the associated threads and leaves the conflict resolution decision to the users who own the notified threads. Note that the destination of a notification message is a thread rather than a designer. This makes it easier for a designer to identify the conflict when he owns multiple active threads. To filter out unnecessary notifications, a set of predicates can be attached to a notification flag to describe the situation when a notification message should be issued. This predicate set overrides the default, i.e., when a new version of the moved object is put to the SDS, and is used to reduce the number of notification messages by imposing more specific notification-triggering conditions. For example, one can specify that notification is needed only when a new version is checked in *and* it is faster than the old one.

In the LWT model interaction among threads occurs via moving objects between synchronization data spaces and threads. To simplify shared data management, no direct data sharing among threads is allowed. Sometimes sharing data only through

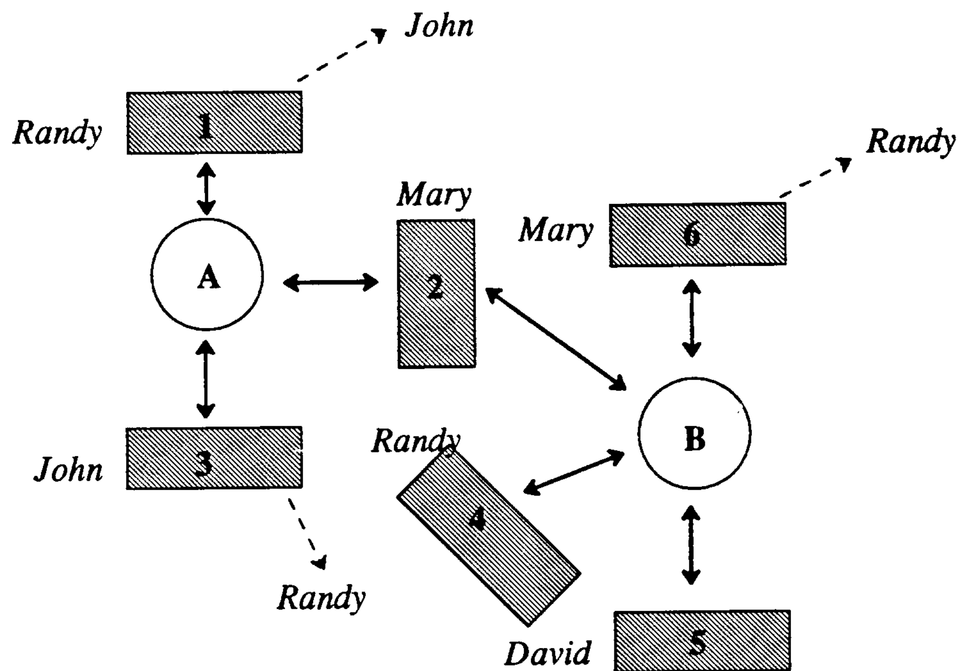


Figure 3.11 The Relationships Between Threads and SDS

synchronization data spaces could be cumbersome. For example, for a small group of designers collaborating intimately on a piece of design, requiring a member to explicitly move an object to a SDS just to let the other member skim through it, is simply too awkward. In this situation, members are assumed to communicate with one another frequently enough that potential inconsistency can be resolved through face-to-face interaction. Therefore, the LWT model offers a *thread import* mechanism to address the above problem. The idea is that designer A can import a design thread from designer B, and then A can *continuously* monitor the evolution of the imported thread but won't be able to participate in the imported thread. In other words, an imported design thread is a read-only thread to the importing user. Note that an imported thread is not a snapshot at the import time, but a continuous reflection of the original thread. Thread import is a unidirectional operation: Designer A may import B's thread but not necessarily vice versa. With the thread import mechanism, close collaboration won't be hampered by

restrictions imposed by the SDS paradigm.

Figure 3.11 shows a example configuration of the relationships among design threads, synchronization data spaces, and the thread import mechanism. Each shaded rectangle represents an evolving design thread, each circle represents a synchronization data space, and double headed arrows mean that the associated design thread can contribute /retrieve objects to/from the corresponding SDS. Numbers in each rectangle denote thread ID's while character strings beside rectangles are users who owns the threads. Characters inside each circle represent synchronization data space ID's. For example, Thread One, Two, and Three owned by Randy, Mary, and John, respectively, can access objects in the SDS A, but the fourth thread cannot. Single-headed arrows indicate the directions of thread import operations. For example, the third thread, which belongs to John, is exported to Randy, but Randy's threads are not exported to John.

3.4 Summary

A design process model called the *light weight transaction* (LWT) model is presented in this chapter. Because the model is targeted for an interactive VLSI design environment, flexibility is the pre-dominant design consideration. As a result, the LWT model differs from conventional database transaction models in significant ways. Concurrency control and error recovery at the physical storage level are not the focus of the LWT model and assumed to be handled by a underlying design database system. Specifically accesses to design data by concurrent CAD tool executions are controlled by a conventional database transaction system. The emphasis of the LWT model is on simplifying tool invocation, facilitating design exploration, and supporting cooperative work.

To simplify tool invocations, we take a semi-structured view towards the VLSI design processes. Two fundamentally different types of design activities are identified: *routine* and *creative*. Routine design activities refer to those whose solution procedures are well-understood and therefore can be specified in advance. For example, generating a PLA-style physical layout from a high-level behavioral specification. Under the single assignment update semantics, the LWT model provides a *design step* construct to

encapsulate individual CAD tool invocations as atomic operations against the design data space. To facilitate routine design activities, A higher-level abstraction than primitive CAD tool invocation called the *design task* is introduced in the LWT model. Design tasks either automate or navigate users through the operation sequences in the task templates. To maintain design tasks as a high-level abstraction, the internal side effects of a task abstraction must be hidden from the users. In particular, the LWT model ensures the atomicity property of a design task across voluntary or forced aborts.

However, interesting designs are always beyond routine design activities. Creative design activities denote those parts of a design process that are hardly understood and thus can not be specified beforehand. Most researchers who intend to help designers beyond routine design choose to use Artificial Intelligence techniques such as rule-based approaches. The basic tactic is to pattern-match the design scenarios to a knowledge base, from which to uncover the best fit design procedure. If such a procedure doesn't exist, human designers still have to step in and take control. So far this approach meets only limited success in special design domains. In our opinion, this approach is wrong because it ignores the fact that an innovative design is unique not only in its final product but also its solution process. Since the process is unique, it is by definition beyond automation.

Recognizing this fundamental fact, we take an approach based on *assistance* rather than automation. Creative design may proceed in a unique path, but there is a distinguishable pattern essential in this process: a creative design process always involves countless iterations of trial-and-error, i.e., creating and evaluating various alternatives in the design space. Instead of attempting to automating the generation and evaluation of design alternatives, as previous AI-oriented systems did, the LWT model aims at facilitating the management of the *contexts* associated with design alternatives. In particular, the LWT model provides the *design threads* construct to cluster the related data objects and design operations associated with a design entity such as an ALU module. Moreover, within a design thread, a rework mechanism is developed for users to refine and examine various design alternatives without having to keep track of the mapping between high-level design decisions and associated data objects, greatly simplifying design space

exploration. It is our belief that the key to support creative design is to reduce this book-keeping overhead rather than to generate the so-called intelligent design plans.

The fundamental issue in supporting cooperative work in VLSI design is to make an optimal tradeoff between sharing and protection. The *thread workspace* concept generalizes the notion of private workspace [CHAN89] in that it clusters data objects that are related to a logical design entity. It also permits a finer-granularity control over the visibility of data objects, i.e., enforcing the notion of context. *Synchronization data space* generalizes the public workspace concept in previous works [CHAN89]. Its structure is more dynamic since the set of associated threads can evolve over time , and it is active because concurrency control can be effectuated through a predicate-controlled selective change notification mechanism.

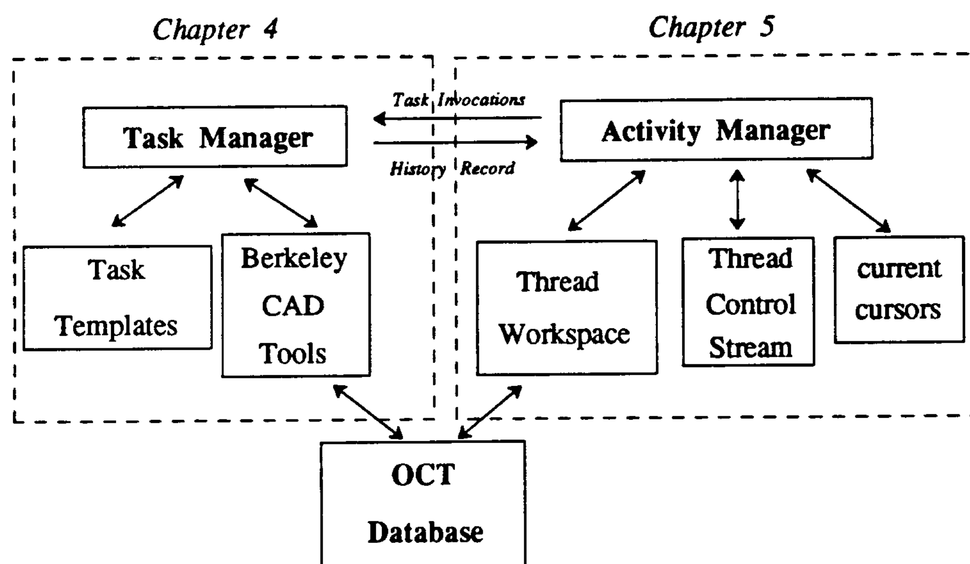


Figure 3.12 Software Architecture of ***Papyrus***

As it turns out, visibility is a unified abstraction to conceptualize the support mechanisms for design exploration and cooperative work. The movement of the current cursor within a thread basically allows users to control their view of the thread workspace by limiting the data visibility. The movement of data objects among thread

workspaces and synchronization data spaces confines the visibility of immature or temporary data objects to their owner and forces shared data to be visible only to those who have the access.

In *Papyrus*, the LWT model is implemented in two software subsystems: a *task manager* and an *activity manager*. The software architecture is shown in Figure 3.12. The task manager interprets task templates, helps users to choose CAD tool execution options, and takes care of low-level invocation details. The activity manager maintains the states of design threads, performs the mapping from design points and thread states, supports thread manipulation operations, and garbage-collecting unactive data objects to reduce the storage overhead due to single assignment update semantics. The interface between these two components are the task invocations from the activity manager to the task manager, and the history records from the task manager to the activity manager. Task invocations specify the name of the tasks to be invoked and their input/output arguments. History records encapsulate the history of design task instantiations, including the options of each design step in the tasks. These two subsystems are described in Chapter Four and Five, respectively.

Chapter 4

Design Task Management

4.1 Introduction

In *Papyrus,* users invoke a task under the control of the *design activity manager.* Once they invoke a design task, users are required to supply the names of the task's inputs and outputs. The activity manager first maps the given names to physical objects and then spawns an instance of the *design task manager* as a child process to act on its behalf, with the invoked task's template and input/output objects passed to the task manager. When a task is completed successfully, the task manager packages the detailed operation history associated with a task invocation into a history record, and reports it back to the activity manager. If a task invocation is aborted by the user, the task manager simply exits after removing all intermediate side effects induced by the aborted task. No history record is created in this case.

A task template is a specification of the design steps (i.e., individual tool invocations) and their sequencing order to accomplish a well-defined design objective. We define a language to specify task templates, which we call *Task Description Language*

(TDL). An earlier implementation [KING89] of the design task manager adopted a LISP-like syntax for task specifications, and stored the parsed task templates in a design database. We choose to take an interpretive apprcach. Task specifications are themselves task templates and are dynamically interpreted by the design task manager. In particular, TDL is built top of the Tool Command Language (Tcl, pronounced "tickle") developed by John Ousterhout [OUST90]. There are several reasons to choose this software architecture. First, because task specifications are just ASCII files, it is easier to add or delete task templates. There is no need to go through the design database when modifications are made to task templates. As a result, the task manager can be implemented independently of the underlying database system. Second, because Tcl is designed as a command language to be embedded into general applications, TDL can augment Tcl's intra-tool control capability by providing inter-tool sequencing coordination. Moreover, if newer CAD tools are written with Tcl as the common embedded command interface, a Tcl-based task description language could facilitate the integration of these presumably separately developed CAD tools. Third, because Tcl inherently provides an interpreter, developing a Tcl-based TDL is greatly simplified. In particular, most of the syntactic constructs in Tcl are readily available to TDL, and the implementation of TDL is restricted to the extension constructs that are added to Tcl.

In the next two sections, we will describe the details of the task description language and its prototype implementation.

4.2 The Task Description Language

In this section, the task description language is presented in two layers. The base is Tcl and the second layer are the extensions we have developed for controlling the sequencing and execution of design steps with a task. Because TDL uses the same parser as Tcl, all Tcl constructs are a part of TDL, except for a few exceptions as noted below. Since the detailed syntax of Tcl is available from other sources [OUST90], we will only present its general outline here.

4.2.1 Tool Command Language

Tcl only supports one data type: the string. Depending on the context, a string can be a command, an expression, or a list. The basic syntax of a Tcl command is a set of fields separated by while space. The first field of a command is the name of the command. The other fields are arguments to the command. Commands are separated by semi-colons or newlines. For example,

set a 27; set b test.C

are two commands that set **a** and **b** to **27** and **test.C**. As mentioned before, the values of **a** and **b** are strings, not numbers as in the case of **27**. To accommodate strings that contain white space, double-quotes or braces can be used to group a set of arguments into a compound argument. However, the actual arguments that are passed to the command don't include the braces and double-quotes. For example,

set a "This is a single operand"
set b {xyz {b c d}}

set **a** and **b** to **This is a single operand** and **xyz {b c d}**.

When an argument of a command contains the $ character, the characters following the $, up to the first character that is not a number, letter, or underscore, are treated as a variable name and these characters are replaced by the variable's current value. This is called *variable substitution*. Alternatively a pair of braces can be used to enclose a variable name after "$" to avoid ambiguity. For example,

set a 100
set b fg
set c Zs${a}d$b

sets **c** to **Zs100dfg.**

The second way of interpreting a string is to treat it as an expression. Several Tcl built-in commands such as **expr, for,** and **if,** view some of their arguments as expressions and call the Tcl expression to evaluate them. Expression strings consist of arguments and operators. Operators can be arithmetic (e.g., +, - *, /) or logical (e.g., and, or, not). A Tcl

expression has C-like syntax and evaluates to an integer result. The arguments to a Tcl expression must assume integer string values after command or variable substitution. There is no need for explicit data type conversion. The expression processor takes care of string-integer conversion automatically. Parentheses can be used for grouping. For example,

(4*2) > 7

($a + 3) <= [set a]

The third interpretation of Tcl strings is as lists. Lists consist of fields separated by white space, just like command strings, except that a newline character in a list is treated as a field separator rather than a list separator. Special commands such as *concat, index length,* and **list** allow users to list-related operations. For example,

ab&c dd {a book {now is}}

is a list with three elements: **ab&c, dd**, and **a book {now is}.**

Tcl also provides various C-like control structures such as **case, if-then-else, for-loop, foreach-loop,** and **proc** for various flow control purposes. These constructs are actually commands themselves. For example,

if {$a > 1} {set b 1} {set b 0}

is an **if** command that takes three arguments: **$a > 1, set b 1**, and **set b 0**. The **if** command first evaluates its first argument, and chooses the second or the third argument to evaluate depending on the evaluation result of the first argument. In this section we only describe those language features of Tcl that are relevant to TDL. Other syntax details and built-in commands can be found in [Tcl Manual].

The basic language features of Tcl is very similar to UNIX shell command languages like the C-shell or Bourne-shell. However, rather than coordinating batch-style commands, Tcl is designed to be embedded into an application and provides a command interface for users of that application. Because the Tcl interpreter is provided as a user-level library, rather than a self-contained program, applications can link to this library to obtain a command interpreter for free.

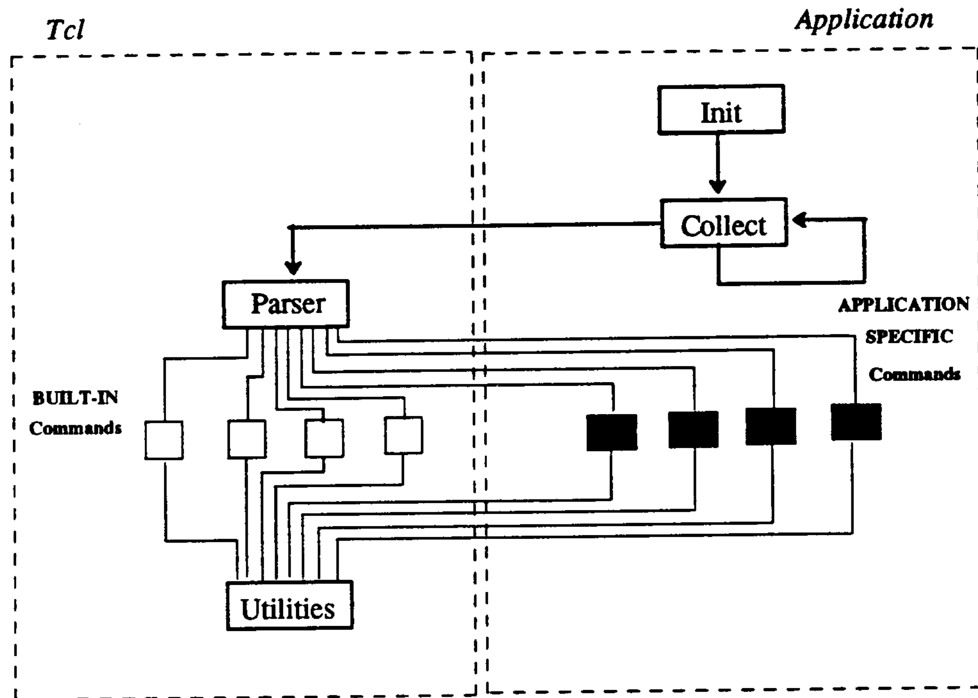


Figure 4.1 Software Structure of a Tcl-based Application

The software structure of a typical Tcl-based application is shown in Figure 4.1. In addition to the built-in commands provided by Tcl, applications can register new commands with the Tcl Interpreter. When registering a new command, the application needs to supply a command name and a function that implements that command. Registration of new commands and initialization of a Tcl interpreter occurs during the *Init* phase of a Tcl-based application. At run time, the application collects user-entered commands and passes to the parser provided by the Tcl library. When a new command is invoked, the Tcl interpreter will redirect the command to the registered function, together with any arguments. It is this *dynamic binding* capability that distinguishes Tcl from other command languages. An important reason for Tcl's success is that it provides a readily available interpreter as a library, which facilitates the acceptance of Tcl as an embedded command language. In addition, this interpreter includes an extension interface that may be

used to extend the language's basic command set. As shown in Figure 4.1, an application developer implements application-specific commands, indicated by shady boxes, and links with the Tcl library, which supplies a parser and low-level utility procedures, to form a complete Tcl-based application. The utilities include procedures for parsing and manipulating strings. TDL assumes exactly the same software architecture as described above, with several extension commands for describing and sequencing CAD tool executions.

4.2.2 Extensions

Because Tcl provides all the control constructs that is needed to describe the flow of design steps in a task, the Task Description Language only needs to focus on the description of individual CAD tools. In particular, TDL provides the following five new commands: **task, step, subtask, abort,** and **attribute.** In the subsequent syntax discussion, bold-face characters represent language reserved words, while Italics characters represent strings that can be replaced by users for their purposes. The strings enclosed by a pair of braces are optional. That is, they don't have to appear when the command is used in a task specification. This should not be confused with the brace constructs described in the last section.

Each task template is stored as a UNIX file. The **task** command declares the beginning of a task specification, and is always the first command. The command has the following syntax:

task *Task_Name* {*Task_Input*} {*Task_Output*}

where *Task_Name* is the name of the task, and *Task_Input* and *Task_Output* denote the list of input and output objects to the task. Objects in a list are separated by white space. Depending on the database model, the object names may assume a particular format such as the *cell:view:facet:version* format in OCT, or just plain ASCII file names. The task command serves two purposes. First, it sets up and initializes the data structure for the interpretation of the following task template. Second, the input and output lists in the task command are to be matched with the **subtask** commands appearing in other task

templates.

The **subtask** command allows one task template to include other task templates as subtasks. This allows the construction of complex tasks out of more primitive tasks. The command has an almost identical syntax as the task command:

[*StepID*:] **subtask** *Task_Name* {*Task_Input*} {*Task_Output*}

The *StepID* field is an optional integer argument that identifies the current step in a task template. This identifier is useful for specifying control dependencies and restart after aborts. If the input or output lists in a subtask command are not matched with their counterparts of the task command in the corresponding task template, an error occurs and the task containing the subtask command is forced to be aborted. Otherwise, the template of the subtask is expanded in-line, and is interpreted as other commands in the invoking task template. There is no limit on the nesting depth of task composition.

The **step** commands describe individual CAD tool invocations. In addition to how a tool is to be invoked, it is also necessary to specify when a tool can be started, and what to do when a design step is aborted. The command assumes the following form:

[*StepID*:] **step** {*Step_Name*} {*Input_List*} {*Output_List*} {*Invocation_Details*}
[{**NonMigrate**}] [{**ResumedStep** *N*}]
[{**ControlDependency** *StepID1 StepID2 ... StepIDm*}]

The *StepID* field is identical to that in the subtask command. The *Step_Name, Input_List,* and *Output_List* fields denote the step's name, inputs, and outputs, respectively. The *Invocation_Details* field contains the details of invoking a CAD tool, including the tool name, its options, parameters, and inputs/outputs. Normally this field specifies how the tool should be invoked, unless users choose to override this default behavior at run time.

The above four fields are mandatory, while the following three are optional. These optional fields do not have to appear in a particular order. They are self-identified by the first string. If the first string is "NonMigrate", that means that the current step is non-migratable. Because *Papyrus* supports independent CAD tool invocations in parallel on

a network of workstations, by default every step is migratable. However, in some cases users may choose to run a CAD tool on his home machine, for example, running interactive tools or tools that heavily rely on files on local disks. If the first string is "ResumedStep", the following string is an integer that designates the resumed step when the current step is aborted during a task invocation. If not specified, the default resumed step ID of a step is zero. This means that if the current step is aborted, the containing task is restarted from scratch. By specifying appropriate resumed steps, useful work can be preserved as explained in Chapter Three.

If the first string is "ControlDependency," it may be followed by one or more integer strings. These denote the steps on which the current step is control-dependent. The semantics are that the current step can be initiated only when all the steps on which it is control-dependent have completed successfully. This construct is introduced for users to impose on the steps a non-data dependent task ordering constraints. For example, after a place_and_route step, a circuit simulator can simulate the resulting layout without the design_rule checker verifying it first. However, if a design group wants to impose a design methodology that no circuit simulation should be performed on an unverified layout, then in the task specification, the circuit_simulation step can be made to be control-dependent on the design_rule_check step.

We introduce the **attribute** command to extract the properties of a design object. Most script languages provide control constructs to change the design flow according to the results of predicate evaluation. The predicates are typically based on simple variables and rarely involve the objects that are manipulated. With the attribute command, the design flow can now be based on a design object's attribute values, which could even be evaluated at run time. The syntax of the attribute command is

attribute *Object_Name Attribute_Name*

This retrieves the *attribute_name* attribute from the object *object_name*. The list of attributes that can be retrieved from a object depend on the type of the object, and also vary with the implementation of the attribute server as described in the next section. The language itself doesn't impose any limits.

The **abort** command aborts a step in a task. Depending on the resumed-step specification of the aborted step, a task may restart from a different state. The command's syntax is

abort [*Step_Identifier*]

where *Step_Identifier* can be either a step ID or a step symbolic name. If the step_identifier field is not specified, then the entire task is aborted. In that case, the task manager cleans up the intermediate side effects associated with the current task and exits.

4.2.3 Examples

A task template can be as simple as an encapsulation of a single CAD tool. For example, the following is a task template that contains a single tool, *padplace,* a placement tool that puts pads on a chip.

task Padp {Incell} {Outcell}
step Pads_Placement {Incell} {Outcell} {padplace -c -o Outcell Incell}

Here the *Padp* task contains a single step called *Pads_Placement.* The last field of the *Pads_Placement* step is in a separate line and specifies the default way the step is to be invoked. Users can override these default options through a graphical interface, as described in the next section. The following example is more complicated and illustrates a generic synthesis sequence from a structure-level description down to a physical layout, including the bounding pads.

The graphical representation of this task is shown in Figure 4.2 From the user's standpoint, only the task inputs and outputs, in this case, *Incell, Musa_Command, Outcell,* and *Cell_Statistics,* are visible. Other objects in the task are intermediates, whose naming and allocation/deallocation are handled by the task manager transparently.

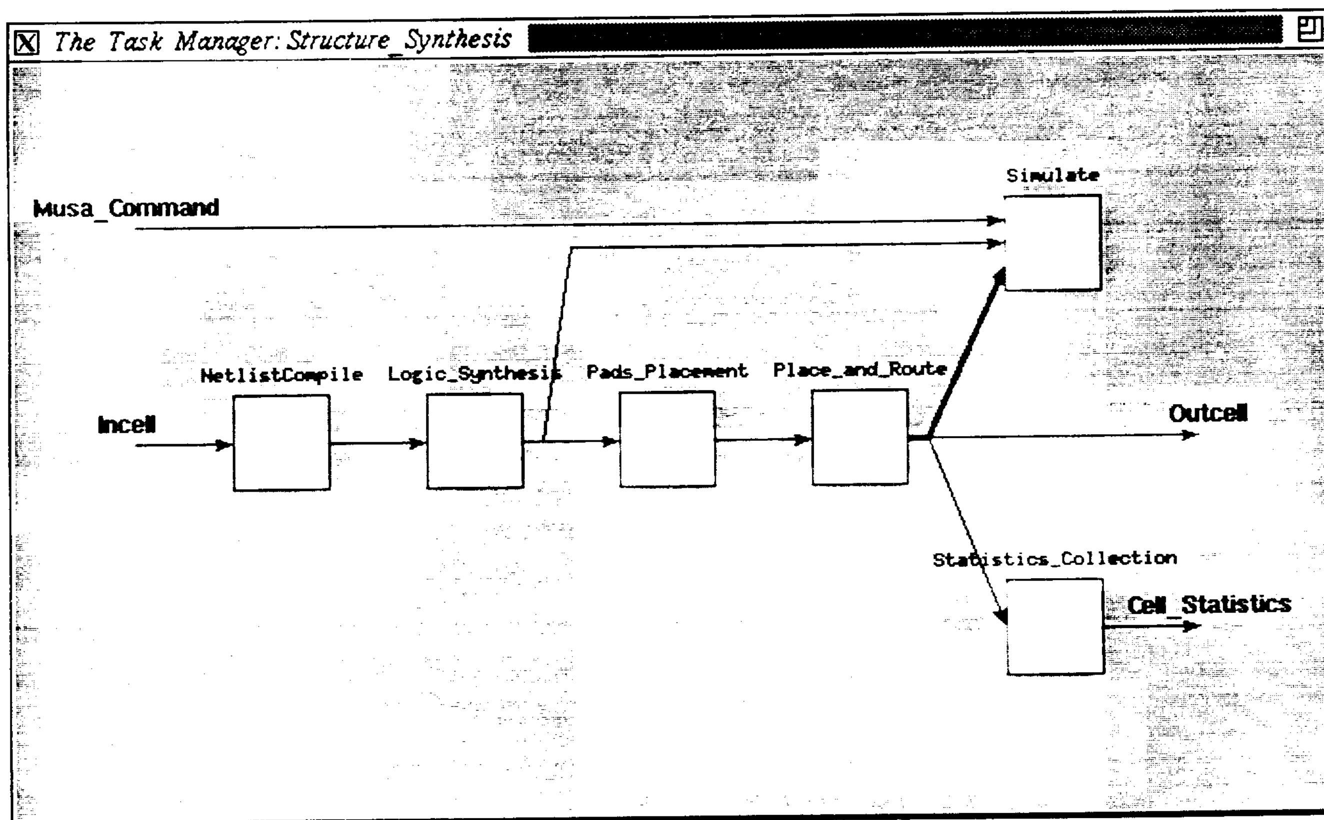


Figure 4.2 Graphical Representation of the Structured_Synthesis Task

```
task Structure_Synthesis {Incell Musa_Command} {Outcell Cell_Statistics}
   /* translate a high-level description to a multi-level logic network */
step NetlistCompile {Incell} {cell.blif} {bdsyn -o cell.blif Incell}
   /* optimize a multi-level logic network */
step Logic_Synthesis {cell.blif} {cell.logic}
      {misII -f script.msu -T oct -o cell.logic cell.blif}
   /* place pads */
subtask Padp {cell.logic} {cell.padp}
   /* place and route to obtain a physical layout */
step {1 Place_and_Route} {cell.padp} {Outcell}
      {wolfe -f -r 2 -o Outcell cell.padp}
   /* perform a multi-level simulation */
step Simulate {cell.logic Musa_Command} {}
      {musa -i Musa_Command cell.logic} {ControlDependency 1}
   /* collect performance statistics */
step Chip_Statistics_Collection {Outcell} {Cell_Statistics}
      {chipstats Outcell > Cell_Statistics}
```

are handled by the task manager transparently. Note also that the *Structure_Synthesis* task contains the *Padp* task as a subtask, which is expanded in-line in the graphical representation and becomes *Pads_Placement.* In this task, the *Simulate* step and the *Place_and_Route* step could have been performed in parallel because there is no data dependency among them. However, since the *Simulate* step designates the *Place_and_Route* step as its control dependency, the *Simulate* step can only start after the completion of the *Place_and_Route* step. Data and control dependencies are represented as arrows except that the latter are in bold arrows.

The next example illustrates some of the control mechanisms provided by TDL, and the utility of programmable abort constructs. The example is a macro-cell placement and routing tool sequence called *Mosaico* in the OCT tool suite.

The graphical representation of *Mosaico* is shown in Figure 4.3. The control flow in the *Mosaico* task is mostly in the form of a linear pipeline. Therefore only limited process-level parallelism can be exploited. Unlike other steps, an IF is represented by a diamond-shaped polygon rather than a rectangle. The *status* variable in the IF test is a global variable automatically returned by the task manager. This variable designates the

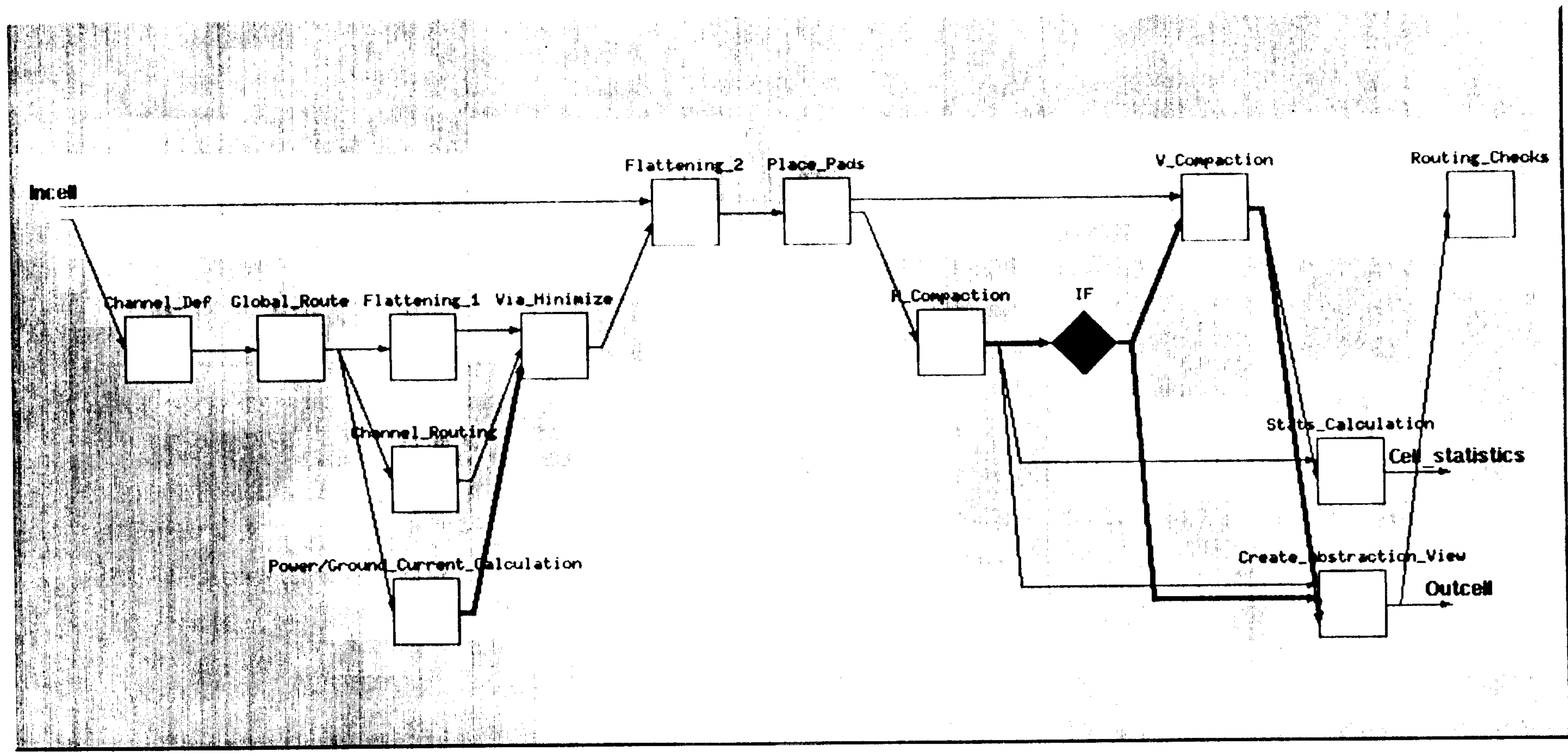


Figure 4.3 Graphical Representation of the Mosaico Task

```
task Mosaico {Incell} {Outcell Cell_statistics}
    /* define the channel ares */
step Channel_Definition {Incell} {cdOutput} {atlas -i -z -o cdOutput Incell}
    /* perform a global routing */
step Global_Routing {cdOutput} {grOutput} {mosaicoGR cdOutput -r -ov grOutput}
    /* calculate the power and ground currents */
step {1 Power/Ground_Current_Calculation} {grOutput} {pgOutput}
      {PGcurrent grOutput > pgOutput}
    /* perform a channel routing */
step Channel_Routing {grOutput} {crOutput}
      {mosaicoDR -d -o crOutput -r YACR grOutput}
    /* format transformation */
step Oct_Symbolic_Flattening_1 {grOutput} {flOutput1}
      {octflatten -r grOutput -o flOutput crOutput}
    /* minimizing the via areas */
step Via_Minimization {flOutput1} {vmOutput} {mizer -o vmOutput flOutput1}
      {ControlDependency 1}
    /* another format transformation */
step Oct_Symbolic_Flattening_2 {Incell vmOutput} {flOutput2}
      { octflatten -r Incell -o flOutput2 vmOutput}
    /* place pads */
step Place_Pads {flOutput2} {ppOutput} {padplace -f -S -o ppOutput flOutput2}
    /* compact the layout starting with the horizontal direction */
step Horizontal_Compaction {ppOutput} {Outcell1}
      {sparcs -t -w NWEL -w PWEL -w PLACE -o Outcell1 ppOutput}
    /* if not successful, compact the layout starting with the vertical direction */
if  {$status} {step Vertical_Compaction {ppOutput} {Outcell1}
                 {sparcs -v -t -w NWEL -w PWEL -w PLACE -o Outcell1 ppOutput}
                 {ResumedStep 1}}
    /* create a protection frame as a high-level abstraction */
step Create_Abstraction_View {Outcell1} {Outcell} { vulcan Outcell1 -o Outcell }
    /* Check for routing completeness */
step Routing_Checks {Outcell} {} {mosaicoRC -m 20 -c Incell Outcell}
    /* collect performance statistics */
step Statistics_Calculation {Outcell1} {Cell_statistics}
      { chipstats Outcell1 |& tee Cell_statistics }
```

exit status of the most recent completed design step. One can decide a task's control flow based on the value of the *status* variable, as shown in the case where vertical-direction-first compaction is used when horizontal-direction-first compaction fails. Also

note that when both compaction methods fail, the task can be restarted from the state right after the completion of the *Power/Ground_Current_Calculation* step, as specified in the **ResumedStep** field of the *Vertical_Compaction* step. All the side effects from the *Channel_Routing* step up to the *Vertical_Compaction* step are removed. Consequently the work performed by the first three design steps are preserved when compaction fails. After restart users can try different parameters for the following design steps to avoid subsequent compaction failures.

4.3 Implementation

The task manager's implementation is divided into two parts: the *interface* and the *execution engine*. The interface part deals with the interactions with the users and is described in the next subsection. The execution engine, described in the rest of the following subsections, controls the execution of a task across a local workstation network and performs necessary bookkeeping to record a task's operation history.

4.3.1 Tool Navigation/Encapsulation

One of the major functions of the design task manager is to navigate circuit designers through a complicated design environment. It shields them from the peculiarities of individual CAD tools. *Papyrus* achieves this goal by informing the users about the status of individual steps and the progress of a task. Most tools are invoked with a set of default options specified in the task template. Users can override these default options by entering their chosen parameters. Figure 4.4 shows a snapshot of the graphical interface presented by the task manager. This interface is built with the Tk toolkit, a Tcl-based X-window toolkit. The top window shows the progress of the current task. Each rectangle represents a step. Different colors are used to indicate the execution status of these steps. Originally the blocks are in white, green stands for completed steps, and red stands for currently-running steps.

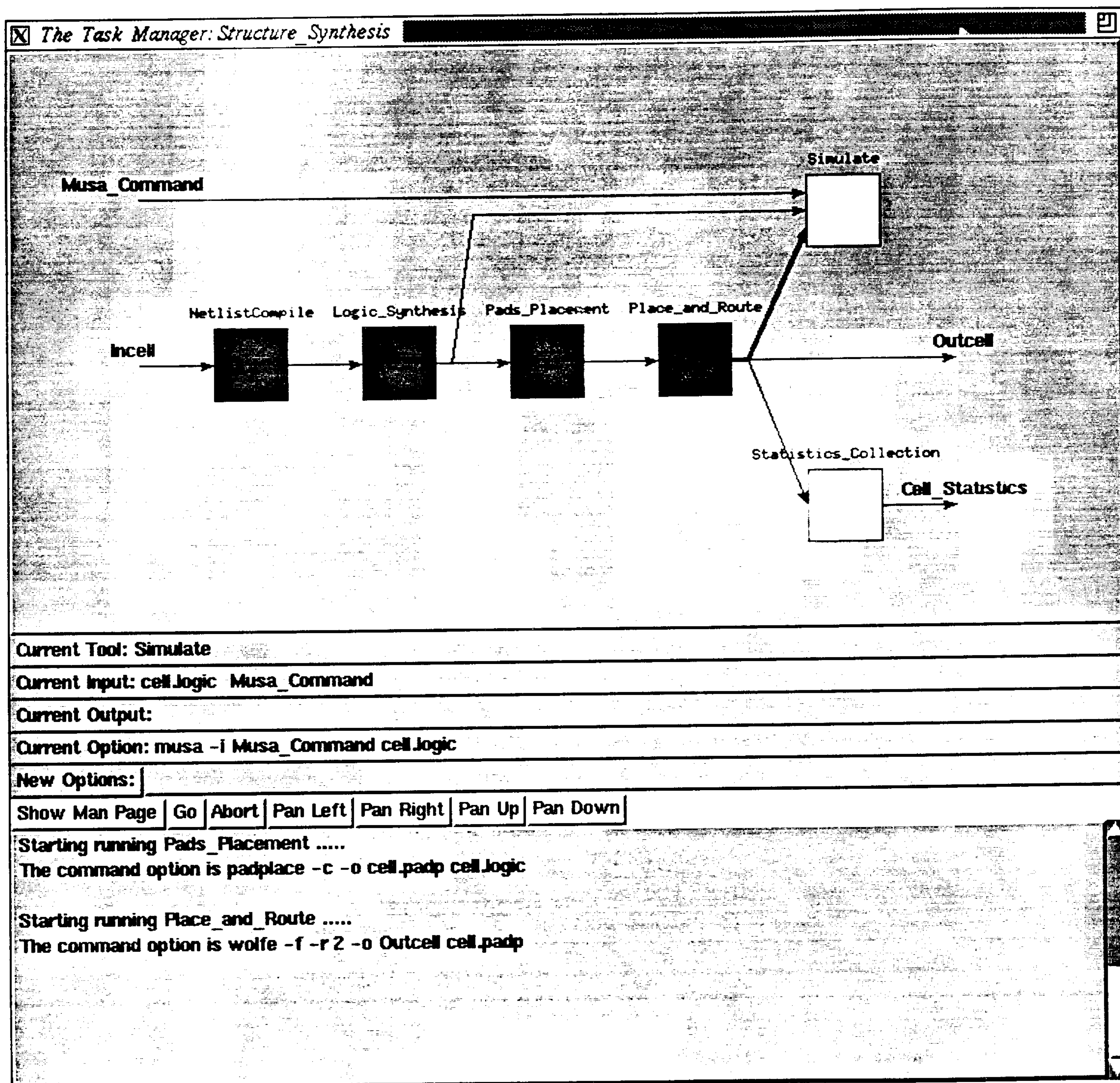


Figure 4.4 A Typical Snapshot of the Task Manager's Interface

Place_and_Route man page

Control Help Search Selection

Tx version 2.4

```
WOLFE                   UNKNOWN MANUAL SECTION                   WOLFE

NAME
     wolfe - Oct Interface to the TimberWolf Standard Cell Place-
     ment Program

SYNOPSIS
     wolfe [-m user@mach] [-o cell:view] [-r rows] [-t file] [-
     dxveifT] cell_name[:cell_view]

DESCRIPTION
     Wolfe reads an Oct symbolic view and performs placement and
     routing to generate a macro-cell in a standard-cell design
     style.  Wolfe uses TimberWolfSC(Version 4.2c) for the place-
     ment and global routing step, and uses YACR (Version 2.1)
     for the detailed routing.

     The instances in the cell are passed to TimberWolfSC to be
     placed.  All net (except for nets connected to SUPPLY pins
     or GROUND pins on a cell) are passed to TimberWolfSC to be
     routed.

     First the facet is cleared of superfluous geometry and rout-
     ing instances (including feed-thru cells inserted during a
     previous run).  This allows multiple runs to be performed on
     the same cell.

--More--
```

Figure 4.5 Man Page for Place_and_Route

Whenever a step is ready to start, i.e., all its data and control dependencies are satisfied, the entries below the task status display window will show the name of the design, the inputs, the outputs, and its current command option. In this case, it is the *Simulate* step that is to be executed next. Users can override the default setting by entering their choices in the space following "New Options:". If users need to consult with the man page to set the options, they can click the "Show Man Page" button. A man page corresponding to the current tool, such as Figure 4.3, will appear. Users can search for keywords when browsing the man page. When users feel that everything is set, they can press the "Go" button to tell the task manager to dispatch this step. The bottom window is a message window, where information related to the current status of the task is presented. In particular, the actual command options to run a design step are listed here. In converting the task template's textual description to a graphical form, we perform a topological sort based on the control and data dependencies among design steps. This is followed by a level-by-level greedy placement and route algorithm to calculate the geometrical positions of the nodes and arrows.

4.3.2 Parallelism Extraction

One of the distinguishing features of *Papyrus*'s task manager is its ability to harness the computation power of a set of networked workstations without user intervention. Moreover, transparency is maintained at both the operating system and the language levels. In other words, users don't need to locate idle machines and explicitly dispatch tool invocations to remote nodes. Neither do users need to specify in a task template the steps that are independent of one another and therefore parallelizable. Because the Task Description Language assumes a sequential computation model, the task manager is responsible for extracting the parallelism from the task specifications. In addition, since commands in a task template are interpreted one after another, the task manager has to adopt a dynamic scheduling approach.

The scheduling unit in a task template is a step (or a CAD tool invocation), therefore the task manager can only exploit the process-level parallelism. A step is said to be

ready when both its data and control dependencies are satisfied. Three data structures are involved in extracting the parallelism within tasks: *Active, Suspending,* and *Result* lists. When the task manager reads in a design step, it first checks if the step's data and control dependencies are satisfied. This check is performed by verifying whether each of the step's inputs is already on the *Result* list, and whether each of its control dependencies is one of the steps that creates the objects on the *Result* list. If they are, the task manager finds an idle workstation, forks a copy of itself to invoke the step's corresponding CAD tool on that node, and puts this step on the *Active* list. Otherwise, the set of dependencies that are not yet satisfied are recorded and the step is put on the *Suspending* list. In either case, the task manager can immediately begin to interpret the next design step in the task template. Essentially the task manager supports *out-of-order issuing* of design steps.

When a step is completed, it catches the attention of the task manager through the UNIX signal mechanism. Since different steps in a task could overlap and execute for different amounts of time, the completion of the steps may not be the same as their dispatch order. That is, the task manager also allows *out-of-order execution.* Because a completed process can only return a limited amount of information, such as its process ID, the task manager must consult the *Active* list to find the step's associated data structure, and then puts each of the step's outputs on the *Result* list. Each *Result* list entry also includes the ID of the output's creating step. A step might be waiting for the completion of a currently executing step. The task manager needs to re-activate those steps in the *Suspending* list that are data-dependent or control-dependent on this completed step. By examining each entry in the *Suspending* list, the task manager marks those dependencies that are satisfied by the completion of the current step. After this procedure, if a previously suspended step's data and control dependencies are all satisfied, the task manager deletes the step from the *Suspending* list, finds an idle workstation to execute the step, and puts it on the *Active* list. Then the task manager waits in a loop for the arrival of the next child process completion signal. The loop is terminated only when the *Active* list becomes empty.

Papyrus is built on top of *Sprite* [OUST89], an experimental network operating system that supports a kernel-level process migration facility. As a result, exploitation of

process-level parallelism becomes relatively straightforward. Sprite provides systems calls to locate idle workstations, execute a process on a designated node, and move a currently running process from one node to the other. Once the parallelism in a task template is identified, running independent design steps in parallel involves nothing more than appropriately packaging parameters and calling adequate systems services. The fact that Sprite supports a network-wide global file system name space, which is different from a conventional NFS-based system, also greatly simplifies both process migration and the implementation of *Papyrus*. Because every node in the network shares an identical view of the file system, location transparency follows naturally. This naming uniformity also makes it easier to control the visibility of design data space by appropriately restricting views against the file system.

4.3.3 Re-Migration

Sprite's process migration mechanism places a higher priority on the autonomy of individual workstations than on load balancing. As a result, only idle nodes, which are defined as nodes that don't receive inputs from mouse/keyboards for a period of time, can accept migrated processes. In other words, even if an node is only slightly loaded by an interactive user, that node is not considered to be qualified. As a result, not all requests for idle nodes could obtain one. If no idle nodes are available when a step is to be dispatched, the task manager simply executes the step locally. On the other hand, Sprite also supports the notion of *eviction*, which occurs when the owner of the previously idle workstation returns, reclaiming his/her machine by touching the mouse or keyboard. In this case, the foreign processes in that node are migrated back to their home nodes.

To more efficiently exploit a network of workstations, a more dynamic migration mechanism is needed, both for the processes that can't secure an idle node when they request one and for those that are evicted from foreign back to home nodes. This mechanism is called *re-migration*. The current implementation of Sprite does not support *re-migration*. The only design issue of implementing *re-migration* is how to identify those children processes that are migratable and yet still execute on the local node. In

our implementation, the task manager periodically examines the local kernel's process control blocks to locate those currently-executing processes whose parent process is the task manager itself. Sprite provides the following system call for querying process states.

```
Proc_GetPCBInfo(0, NUM_PCBS-1, Hid, sizeof(Proc_PCBInfo),
        infos, argStrings, &pcbsUsed);
```

From these processes, the task manager can find the migratable processes by checking their **migrate** flags. When such a process is found, the task manager attempts to dispatch it using the same procedure as described above. Of course, there is no guarantee that the process will be migrated successfully. In that case, the process has to wait for the next round. Fortunately, re-migration takes place while the task manager is waiting for the completion of its active children processes. As a result re-migration is almost continuously performed. Eventually a migratable process should be able to be successfully migrated to an idle node.

4.3.4 Programmable Abort Semantics

The other important feature of *Papyrus*'s task manager is that it supports the concept of *programmable abort.* Compulsory aborts can be issued by the task manager when it detects that the execution status of a design step is in error, or controlled by the users through an explicit *abort* command. Our mechanism is not designed to address serious failures such as client or server crashes. Each step in a task template can specify a so-called *resumed step*, whose following state is the restart state when the former is aborted. Because the Task Description language assumes a sequential execution semantics, it is relatively easy to identify such a state. However, because the task manager allows *out-of-order issue* and *out-of-order execution,* some bookkeeping must be maintained to allow reconstruction of a particular state. Because the notion of state is tied to how object names are managed, we first discuss the issue of name management.

When invoking a task, users are required to supply the names of the task's input and output objects. The names of the intermediate objects are generated by the task manager automatically. Because there can be multiple instances of the same task active

simultaneously, the name of an intermediate must be unique across multiple invocations of the same task to avoid interference. Our strategy is to append the process ID of the task manager to the formal names of the intermediates specified in the task templates. Because each task invocation is controlled by a different instance of the task manager, this scheme guarantees that intermediates' names in different invocations won't conflict with each other. Because modifications to intermediates of a task template also observe the single assignment update semantics, restoring a state is equivalent to removing the associated objects.

Because the interpretive approach used in *Papyrus*, we have to make the following assumptions. First, although a command in TDL can contain arbitrary numbers of steps, for example by separating them via semicolons, the resumed steps must be steps at the top level. In other words, a step cannot specify an embedded step as its resumed step. Moreover, all the steps contained in a top-level command share an internal ID. For example, suppose a top-level command is

```
if {$a > 5} then {step_A; step_B}
        else {if {$a > 1} then {step_C} else {step_D}; step_E}
```

Then step_A, step_B, step_C, step_D, and step_E share the same internal ID as the top-level **if** command. Second, by the same token, the steps within a subtask cannot be designated as a resumed step from the task enclosing the subtask. Because subtasks are expanded in-line, the steps within a subtask can specify their own resumed steps and the system can handle this case without difficulty.

The only complication arises when there is a conflict between the step ID's used in a subtask and the subtask's containing task. Since subtasks can be nested to arbitrary depth, a more general naming scheme is needed. Our solution is that for each subtask in a task templates, an internal ID determined by concatenating the subtask's invocation position in the enclosing task and its nesting depth is formed, and every step ID contained within a subtask is prepended with the subtask's internal ID. At the top level, the nesting depth is zero.

With these assumptions, each top-level command has an internal ID, which is essentially the sequential program ordering of the top-level commands in a task template. When a step is aborted, the task manager first examines its resumed step ID. If the ID is zero, then the entire task is to be restarted. In this case, the objects in the *Result* list are removed, all the processes in the *Active* list are killed, the three lists are reset, and the task manager starts to interpret the task template from the beginning. If the resumed step ID is non-zero, the task manager maps this step ID to its corresponding internal ID. Given the resumed step's internal ID, *J*, the *Result* list's objects that are created by steps whose internal ID's are larger than *J* are removed, and all the processes in the *Active* list that have a larger internal ID than J are killed. The entries of the *Active* and *Suspending* lists are deleted if they correspond to steps that have a larger internal ID than J. Finally the task manager restarts the task by starting to interpret the (J+1)-th top-level command.

4.3.5 History Recording

After all the steps in a task are completed, the task manager needs to do two things. First, the intermediate objects created during a task invocation are to be removed. Since tasks are supposed to be a high-level abstraction that shields the details of individual CAD tool invocations from the users, we feel it is important to maintain this abstraction by hiding the distributed nature of the underlying computation environment and the fact that individual steps may be aborted. Consequently it is imperative to remove hidden side effects when appropriate, i.e., removing the intermediate objects. The task manager recognizes intermediate objects by scanning the *Result* list and excluding the task outputs.

The second thing the task manager does is to package the operation history of the steps of a task into a history record, and passes it back to the activity manager. The operation history of a task invocation is a linear sequence of the steps that are actually executed, ordered by their completion time. Associated with each step is its parameter options and input/output objects. Since a task manager is spawned by the activity manager, the history record information is returned to the activity manager through a file

given by the activity manager. In other words, the task manager controlling a task invocation writes out the history record to the designated file before termination, which in turn, via a signal, notifies the activity manager to load that file to its maintained design history data structure.

4.3.6 Attribute Management

Objects and attributes are stored separately. There is a central attribute database associated with each thread workspace. The implementation of this attribute database is described in Chapter Five. Attribute values are either retrieved directly or dynamically computed. An object's attribute consists of three parts: *attribute name*, *attribute value*, and *attribute computation tool*. For those attributes that need to be computed dynamically, the specified tools are used to compute the values of these attributes. The task manager interacts with the attribute database through a UNIX *db* library. If there is a need to compute attribute values dynamically, it spawns a child process to do it.

Unlike the execution of design steps, the computation of attributes is synchronous, i.e, the task manager waits for the return of attribute values before moving on to the next command. This design decision is made on the observation that attribute computations are usually embedded within **if** commands. Therefore, the attribute values must be available for **if** commands to proceed. Upon return, the task manager caches the computed results in the corresponding entries in the attribute database. It is possible to optimize the computation and storage of attributes. These mechanisms are described in Chapter Six.

4.4 Summary

In this chapter, we introduce the task description language (TDL) and its implementation. The language is based on Tcl with a set of extensions for describing and sequencing VLSI CAD tool executions. Because of this, most of the facilities of Tcl are readily available to the TDL. The extensions to Tcl are implemented as procedures that can be called by the Tcl Interpreter. The features of TDL includes the support of conditional

design flows, programmable aborts, and a limited class of While-loops. TDL assumes a sequential semantics for two reasons. First, users are not required to express parallelism explicitly; they can be extracted by the task manager automatically. Second, a sequential execution semantics entails a well-defined task execution state, which makes error restart easier. The task manager can exploit the process-level parallelism in a task by running independent steps in parallel on a network of workstations. This is made possible by the kernel-level process migration mechanism provided by *Sprite*. In addition, we describe in detail how the task manager implements parallelism extraction, re-migration, and atomicity guarantees. The task manager also records the operation history of an individual task. When a task is completed, its history record is passed to the activity manager, which maintains the history records in such a way that the context of a design entity is preserved. In the next chapter, we will describe the implementation details of the design activity manager.

Chapter 5

Design Activity Management

5.1 Introduction

The design activity management subsystem maintains the design threads' history and supports the *rework* mechanism by resolving object names in appropriate contexts. The design activity manager and the design task manager are implemented as two separate UNIX processes. Users invoke a new instance of the activity manager when they start a new activity. They interact with the design activity manager to invoke a task, at which time they are required to supply the object names of the invoked task's arguments. If the version number of an input object is not specified, the design activity manager will match the object name against the *data scope*, that is, the thread state of the current cursor, and returns the most recent version of that object in the data scope. In the case of an output object, the design activity manager returns the next version number of the object and preserves the single assignment update semantics. When the versions of input/output objects are resolved, the design activity manager instantiates an instance of the invoked task by forking a child process to run the design task manager with the given task and its inputs/outputs. In the interim, the design task manager navigates the users through the

task templates and coordinates the execution of individual steps within a task, as discussed in Chapter Four. When a task is completed, the design task manager packages the invocation history associated with the task into a history record, and sends an exit signal to notify the design activity manager of the history record. With a simple interface and modular software architecture, the development of these two subsystems can be kept reasonably independent. In fact, to make *Papyrus* entirely Tcl-based, the current version of the design task manager completely re-implements the one originally described in [KING89]; the design activity manager is largely unaffected by this decision.

In this chapter, we will present the software architecture and the data structures used in the implementation of the design activity manager. In Section 5.2, the design and implementation of the activity manager's graphical interface are presented. In Section 5.3, the algorithms for managing the design history and computing the data scope are explained in detail. Storage management techniques to reduce the storage overhead associated with the single assignment update semantics are discussed in Section 5.4. Section 5.5 concludes this chapter with a summary of major design issues and solutions in implementing the design activity manager.

5.2 User Interface

Most of the efforts associated with the activity manager's user interface are devoted to the graphical presentation of the design history, specifically the control streams of design threads. A typical snapshot of the activity manager's window is shown in Figure 5.1. Each oval block denotes the history record of a completed design task, with the task's name attached under the oval block. The arrows represent the temporal order of the *completion* of the tasks. When a task is completed, the activity manager receives a history record from the corresponding task manager and appends a block at the appropriate place of the thread's control stream. For simplicity, the prototype implementation assumes that the display is decomposed into a set of square grid cells. Each oval block is assigned a grid cell, whose lower-left corner uniquely determines the placement of this oval block.

To invoke a new task, users click the *Invoke A Task* button. A list of task templates is displayed, as shown in Figure 5.2, from which users select a task to invoke by double-clicking the desired entry. Users are then required to enter the object names of the arguments to the invoked task, as shown in Figure 5.3. These names take the following three forms: an hierarchical file-system path name, such as /user/chiueh/Multiplier; a plain object name concatenated with a version number, such as ALU.logic@1; or a plain file name, such as ALU.logic. A '@' character separates the object names from their version numbers. Input objects to a task can assume the above three formats while output objects typically take only the third form.

The first format is used when the object in question is not currently in the thread workspace. As a result, the task manager needs to physically copy the object into the thread workspace, and updates the workspace's directory accordingly. This is similar to the check-in operation in version management systems except that in this case check-in is implicit. The second format is used when users want to bypass the default version resolution mechanism. For example, although the most recent version of an object is the third version, it might so happen that users insist to use the first version. Users can explicitly use this naming format to get at the desired version. The third format is the most common. In the case of input objects, the most recent version of that object in the data scope is returned. In the case of output objects, a new version is created to preserve the single assignment update semantics.

The *stagger* sign in Figure 5.1 represents the current cursor, whose thread state, called the *data scope*, is the default context in which the inputs/outputs names of the next invoked task are resolved. Users can move the current cursor by clicking the *Move Cursor* button first and then clicking on the history record to which the current cursor intends to move. By controlling the position of the current cursor, users control the context in which new tasks are invoked, or the portion of the thread workspace that is visible. Before invoking a task, users may want to browse the thread state associated with the current cursor. This can be done by clicking the *Show Data Scope* button. An example of the data scope is shown in Figure 5.4. In this case the current cursor is at a "Structure_Synthesis" task. The number beside the current cursor is its recording time.

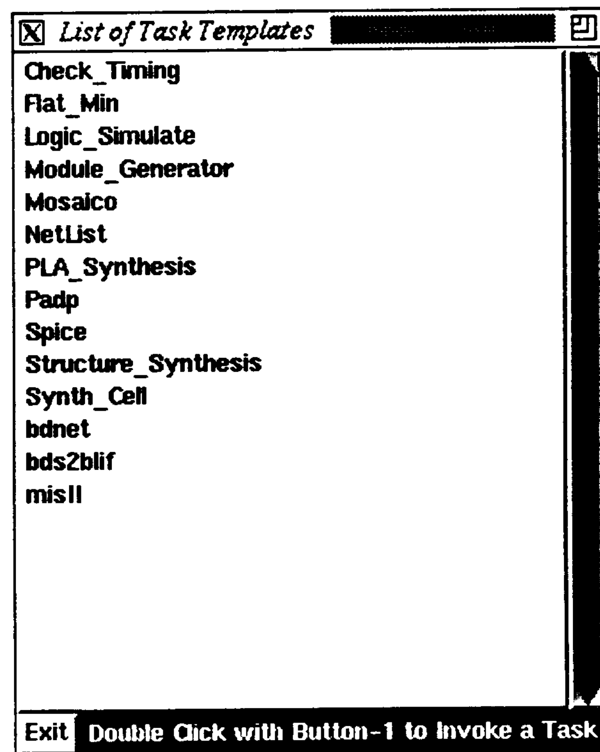


Figure 5.2 List of Task Templates

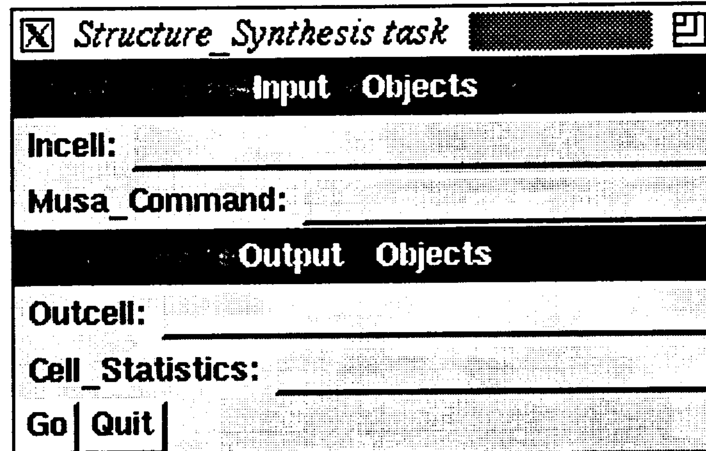


Figure 5.3 Invoking a Task

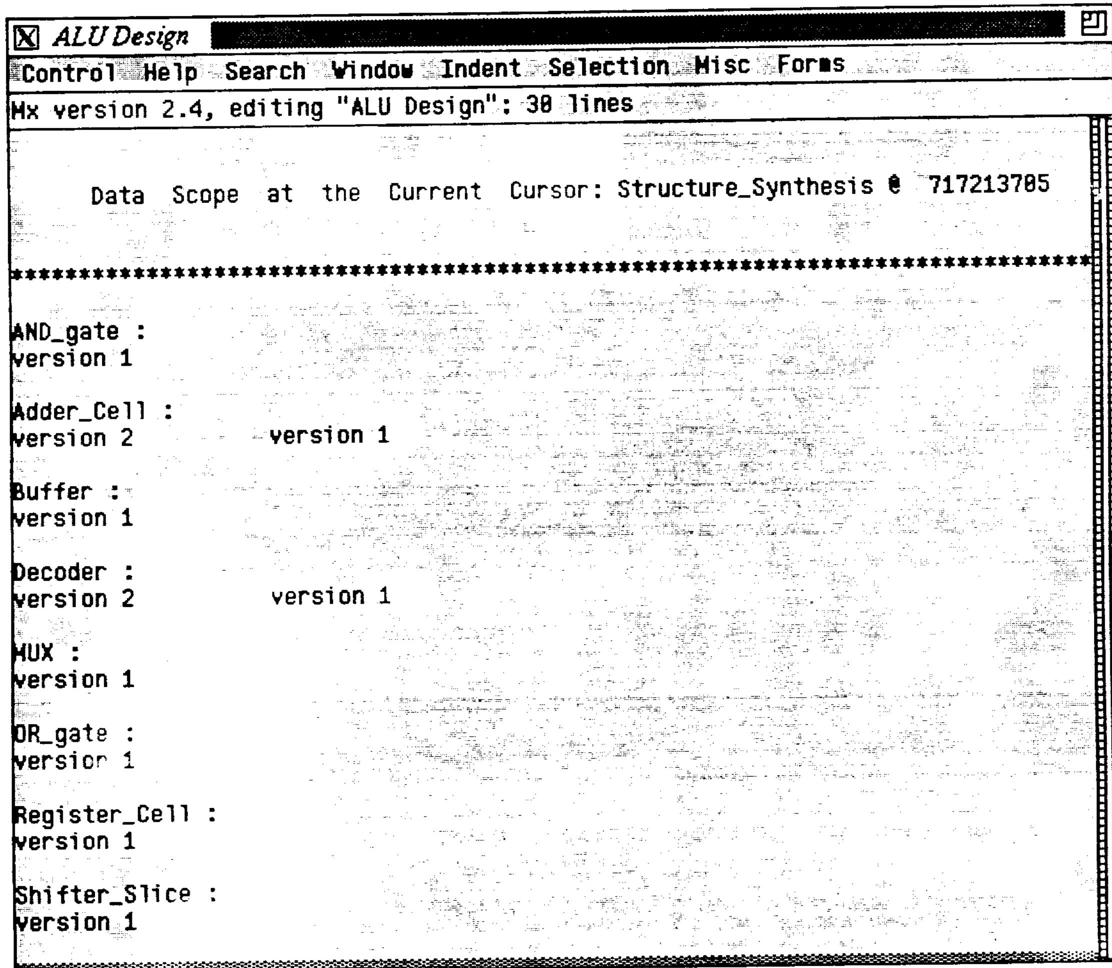


Figure 5.4 An Example Snapshot of the Data Scope

This is used to distinguish different instances of the same task. Alternatively users could view the set of all objects associated with the design thread, in this case, *ALU Design*, by clicking on the *Show Thread Workspace* button. To get a better perspective when choosing the new current cursor, users could browse the control stream by panning the display in up/down and right/left directions, and/or zoom in/out.

As in the design task manager, the graphical user interface of the activity manager is built on top of the Tcl/Tk library. Because this library provide rather high-level interfaces for common user-interface widgets, the implementation of the activity manager's graphical interface is made easy. Unfortunately, the support for structural graphics in the current version of Tcl/Tk, such as those needed for presenting the design history, is relatively limited. In particular, there is no facility to query the geometrical coordinates of an graphic item on a *canvas* widget[1], which is designed to implement structural graphics. One consequence of this lack of support is that the applications need to keep track of these items' coordinates when they are being panned and zoomed. In our prototype, pans and zooms are implemented as modifications to the physical coordinates of the graphical items. Therefore, when new graphical items are to be inserted and displayed in the canvas, the activity manager must have the coordinates of existing items to present them in a harmonious fashion. Unfortunately maintaining these geometrical information in the applications could lead to performance problems when there are a lot of history records in the control stream (and therefore a lot of graphical items), because each pan/zoom will cause a complete traversal of the existing history records.

Our solution is to log the changes due to pans/zooms in an intelligent way and lazily apply these changes when new history records are to be appended. Let's use an example to illustrate this idea. Suppose a sequence of pans and zooms produce the following sequence:

[50, 0] {2} {2} [100, 0] {0.5} [-20, 0] [0, 50]

1. A canvas widget is a graphical area on which structured graphics items such as polygons, lines, and text strings can be displayed.

where a brace encloses a magnification factor, while a bracket contains a pair of numbers denoting a 2-D translation vector. Instead of applying them one at a time, we combine them and apply the resulting effect only when new history records are added. The central observations are the following:

[1] Consecutive translations and magnifications can be merged by addition and multiplication, respectively.

[2] Magnifications that are separated by translations can still be merged through multiplication.

[3] Translations that are separated by magnifications can be merged only when the translation vectors are normalized by the inverse of the accumulated magnification factors.

Consequently the above sequence can be decomposed into a translation and a magnification sequence as follows:

[50, 0] [25, 0] [-10, 0] [0, 25]   {2} {2} {0.5}

For example, the third translation vector, [-10, 0], is obtained by multiplying the original, [20, 0], by the inverse of the accumulated magnification factor, $\frac{1}{2 * 2 * 0.5}$. By applying the first observation, these two sequences could be further compressed into [65, 25] and {2} respectively. When a new history record is to be appended, the coordinates of the corresponding oval block is first translated by the compressed translated vector and then magnified by the accumulated magnification factor. With this method, new history records can be intermixed with existing history records in a graphically consistent fashion. This is accomplished without explicitly keeping track of the physical coordinates of their associated oval blocks.

In addition to pans/zooms, the design activity manager also provides a random access facility to access the design history. In particular, users can access specific history records based on time and/or annotation. When a history record is appended, the current time is also recorded. An example time-based access query is "Go to the design point as of 4:PM 9/17/92". However, for efficiency reasons, the resolution of temporal access is restricted on an hour-by-hour basis. In other words, a new history record is inserted in the

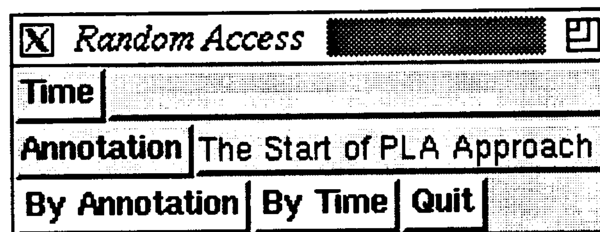


Figure 5.5 Random Access to Design Points

temporal index only when its recording time is at least an hour away from that of the last entry in the index. As a result, given a designated hour, the first history record within that hour is returned, if one exists; otherwise the next closest history record after that hour is returned instead. Users can also annotate history records with textual strings such as "The Start of PLA Approach". With annotation attachments, it becomes possible to access specific history records using annotations. Figure 5.5 shows an example annotation-based query that accesses the history record with an annotation attachment "The Start of PLA Approach." A random access interface allows users to move the current cursor to the desired history record without extensive browsing.

5.3 History Management

There are three data structures associated with the history records of a design thread's control stream. The first is for the graphical display of history records. The second maintains the structure of a control stream in main memory and computes the thread state of the current cursor. The third is a persistent version of the second data structure, for inter-process communication and crash recovery. The first and second design history data structures and their related operations are discussed in this section.

The main memory data structure for maintaining the control stream of a design thread mimics their graphical structure, i.e., a tree. However, there are two complications. First, a control stream becomes a graph when two control streams are merged, as discussed in Chapter Three. Second, even if a control stream is a tree, the structure of this tree is dynamically changing. In particular, the number of children of a tree node is not statically fixed because users can create an arbitrary number of branches while they explore the design space. In other words, the data structure for history records must accommodate a variable number of children AND parents. We use the following data structure to represent history records.

```
struct HistoryRecord {
  int HRNumber;
  boolean CacheFlag;
  char *TaskName;
  int X;
  int Y;
  long Time;
  char *Input;
  char *Output;
  int ParentCount;
  union Previous {
       struct Entry *ParentChain;
       struct HistoryRecord *Parent;
  };
  int ChildCount;
  union Next {
       struct Entry *ChildChain;
       struct HistoryRecord *Child;
  };
};

  struct Entry {
      int path;
      struct HistoryRecord *hrPtr;
      struct Entry *Next;
  };
```

The *HRNumber* field is a unique identifier for each history record. The *X* and *Y* fields are the coordinates of the lower-left corner of the history record's enclosing grid cell. The *Time* field records the time at which the history record is appended to the control stream. *Input* and *Output* fields store lists of names for input and output objects, respectively. When the *ParentCount* field is one, the *Previous* field contains a pointer to the record's parent. Otherwise, the *Previous* field contains a pointer to a chain of pointers to the history record's parent nodes. Similar interpretations are true for the *ChildCount* and *Next* fields.

When a new history record is appended, the *Next* and *Previous* fields of the associated history records are manipulated accordingly. However, there is the question of where in the control stream should the new history record be appended. Because history records are appended to the control stream after the corresponding tasks are completed, it

is not possible to determine the "appending" point when a task is first invoked. The complication arises from the fact that while a task is being executed, users may change the current cursor to another design point of the control stream to invoke new tasks, or other tasks might be completed during this period, which advance the current cursor. Consequently, the current cursor at the time when a task is invoked is not necessarily the same as the one when this task is completed. Therefore it is not correct to append a new history record wherever the current cursor is located.

We solve this problem by imposing the following convention: a history record should be appended to the logical path of the current cursor at the time when the corresponding task is invoked. Consequently we can represent the appending point of a task's history record with a pointer to the current cursor when the task is invoked (called the *invocation cursor*), and a path number. When a new history record is to be appended, the activity manager traverses the control stream starting from the invocation cursor until reaching the end of the path denoted by the path number, to which the new history record is appended. If, during the traversal, a branch is encountered, the history record is instead inserted before the branching history record. As shown in Figure 5.6, the newly completed history record is inserted into the control stream at the design point marked the *insertion point*. The rationale of the latter design decision is that there should not be branches between where a history record is actually inserted and its invocation cursor.

An important service provided by the design activity manager is *rework*. Users move the current cursor to existing design points to experiment with different design operation sequences against a particular snapshot of the design database. Because of the single assignment update semantics, implementation of the rework mechanism is reduced to a name management facility, specifically a mapping from object names to their most recent versions in the data scope, the thread state of the current cursor.

The data scope could change either because a new history record is added or because users move the current cursor. When a history record is added to the control stream, its inputs and outputs are added to the data scope if the current cursor is the same as its insertion point. Otherwise some cache entries may need to be updated, as discussed below. When users move the current cursor, the new data scope can be computed by a

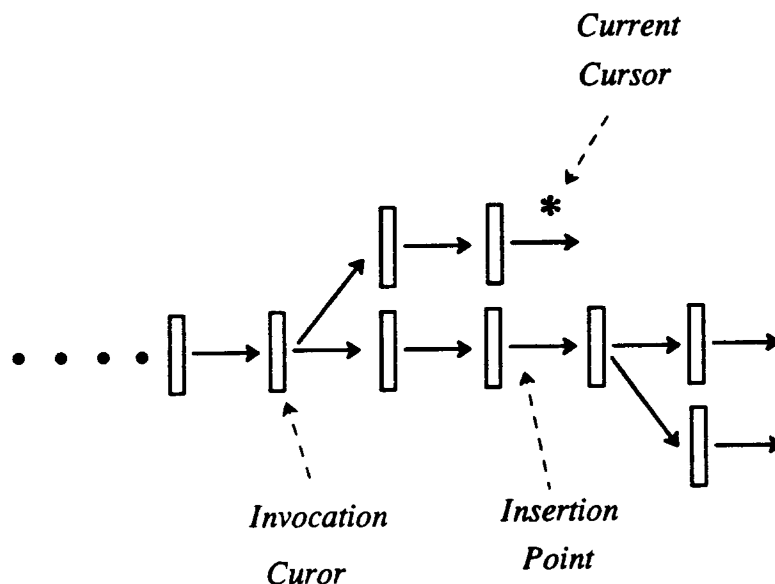


Figure 5.6 Inserting a Completed History Record

simple backward traversal of the control stream starting from the new current cursor. During this traversal, if a node has more than one parent (because of thread merges), all of them need to be visited. On visiting each history record, the inputs and outputs of the history record are collected and ordered according to the lexical order of their names and version numbers.

The activity manager optimizes the computation of the data scope by caching the thread states of certain design points in the control stream. As a result, the computation of new data scopes can stop as soon as the backward traversal of the control stream reaches a design point whose thread state has been cached. New history records can attach to a design point that is a logical predecessor of another design point whose thread state has been cached. Consequently the insertion of new history records may lead to modifications of the cached thread states. If a design point's thread state is cached, the *CacheFlag* field of the associated history record is turned on. Thus, when a new history record is inserted, rather than appended, to the control stream, the activity manager must traverse the following history records. It must update the cached thread states when it

finds history records whose *CacheFlag* is on. Updating the cached thread states involves inserting the output objects of the new history record.

One important feature that distinguishes *Papyrus* from other design management systems such as *VOV* is that it provides facilities for users to manipulate design threads as first class objects. With this capability, users are free to choose the granularity of a design thread based on specific needs. The semantics of forking a new thread from an existing thread is to copy the latter's *thread workspace, control stream,* and *frontier cursor set* to the former. Since these data structures are stored persistently, implementation of forking is relatively straightforward. A special case of forking is to only copy the thread state of a particular design point in another design thread. In this case, only the portion of the control stream involved in the computation of the design point's thread state is copied and the specified design point is the current cursor.

To join (or cascade) two threads to form a new one, the thread workspaces of the original threads are unioned and copied. The control streams of the original threads are duplicated and connected at the specified connector design points. Because connector design points are required to be frontier cursors, cached thread states from the original threads are still valid even in the merged new thread. Therefore they can be reused. This is not true when cascading two threads. In this case the cached thread states of the following thread must be recomputed by incorporating the thread state of the connector design point of the preceding thread. In both cases, the new frontier set is obtained by unioning the original threads' frontier cursor sets minus the connector design points.

5.4 Storage Management

One fundamental assumption made by *Papyrus* is the single assignment update semantics: updates to an object are not performed in place but rather create new versions. Although this assumption simplifies the conceptual model and many implementation issues related to the *rework* mechanism, it drastically increases the storage overhead if appropriate measures are not taken. We call the techniques used in *Papyrus* to address this problem *object reclamation*. This is provided by a process independent of the

activity manager. It communicates with the activity manager through the persistent version of the design history.

The idea of reclamation is that the entire thread workspace is treated as a */tmp* file system in UNIX. Intuitively, the disk space of versions of design objects are reclaimed if they are not accessed for a specified period of time. By "reclaimed," we mean object versions are either deleted or archived to a tertiary storage subsystem such as a tape device. The current implementation simply deletes the reclaimed versions. But the interface is general enough so that automatic archiving is possible when a tertiary storage device becomes available. On the other hand, *Papyrus* doesn't base the reclamation decision solely on a single *last-time-to-access* attribute. Instead it analyzes the design history maintained by the activity manager and attempts to reclaim those object versions that are least likely to be needed in the future.

Because users are allowed to move the current cursor, there are conflicting considerations as far as version reclamation is concerned. On the one hand, the system should allow users to move the current cursor to arbitrary design points in a design thread. On the other, to provide more focused contextual information to designers and to make efficient use of the available storage space, it is necessary to prune away unneeded or useless history records and their associated object versions whenever possible. The issue then is how to maintain the balance between keeping the most relevant portion of a design thread's history and supporting "meaningful" reworks. In the following, design points in a design thread that can create branches in the control stream are said to be *reworkable*. Currently we use three mechanisms in *Papyrus*: filtering, aging, and garbage collection.

Filtering

Users can specify a set of tasks to be monitored by the activity manager. In other words, any task invocations not in the set will not be maintained by the activity manager. For these, the history records passed from the task manager are simply discarded. Example tasks not typically maintained by the activity manager include "facility" tasks, such as printing or displaying something on the screen. These usually have no influence on the

design. Since they are not an integral part of the design process, the activity manager can get away with maintaining them in the design history.

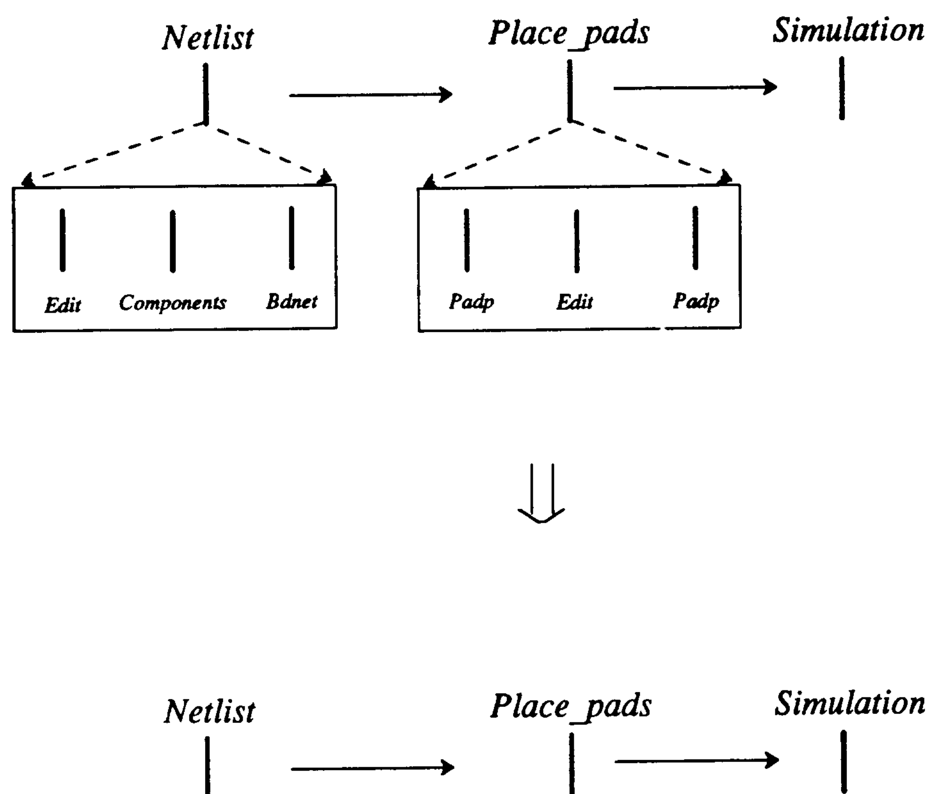


Figure 5.7 The Effect of Vertical Aging

Aging

From the user's standpoint, it is generally true that the relevance of history records to the current design context decreases with their age. Old history records should be abstracted away so that only sufficient historical details are preserved. There are two ways to exploit the aging mechanism to reduce the amount of historical information: vertical and horizontal. *Vertical aging* refers to level compaction based on the ages of task invocations. In particular, the details of past task invocations can be progressively "forgotten" as they get older. For example, in Figure 5.7, both *Netlist* and *Place_Pads* are composite tasks that contain subtasks. As they become old, their internal details are abstracted away as

shown in the other side of the arrow. *Horizontal aging* refers to the process of reclaiming the part of the design history that are too far back in time. For example, in Figure 5.8, *Netlist* and *Place_Pads* are are pruned away from the design history when they get old. The "*" sign shows that there are history records before that point but they have been reclaimed by the design activity manager. All of these "forgetting" operations (vertical or horizontal) involve deleting intermediate data objects and the associated history records. Currently the way the design activity manager "forgets" the history is by *actively* reminding users that some part of the design history will to be pruned away. Only when users approve the activity manager perform the pruning operations.

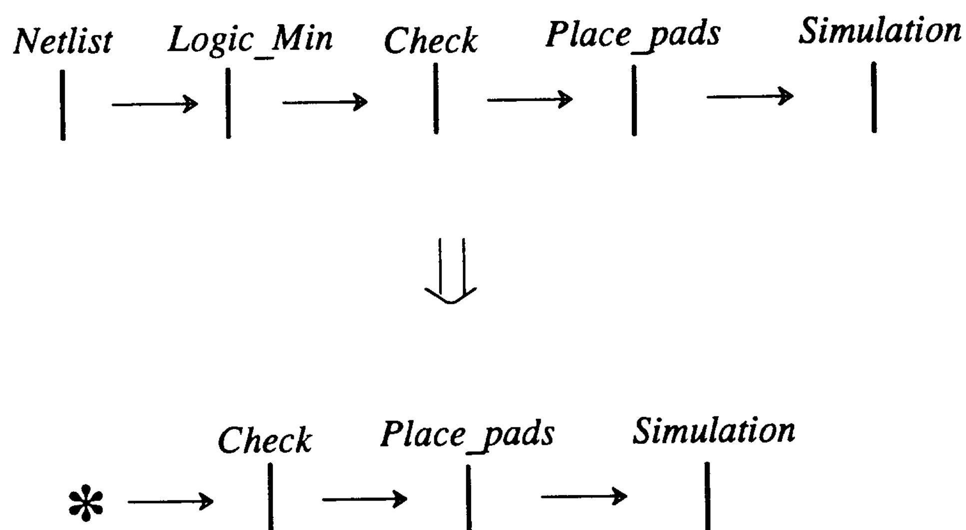


Figure 5.8 The Effect of Horizontal Aging

Garbage Collection

The term *garbage collection*, as used here, refers to a general procedure for finding "abandoned" history items. There are two possible types of abandoned history items. In an iterative refinement process, a sequence of task invocations is iterated several times. Typically only the results of a small subset of the iterations are used later. The garbage collector abstracts the entire iterative process with the small subset that is actually used, and eliminates the history elements associated with the remaining iterations. The current

implementation of *Papyrus* is not intelligent enough to discover iterative processes from the history. The user must provide explicit hints identify sequences of task invocations corresponding to iterative processes. For each such iterative sequence, the garbage collector finds all iterations (typically only one) whose outputs are used by later task invocations. Other iterations are pruned away. In Figure 5.9, refinement of a layout object by iterative layout edits and circuit simulation is abstracted as a representative round (the third round in this case). The numbers below the history records denote the round numbers. They are shown only for the purpose of illustration.

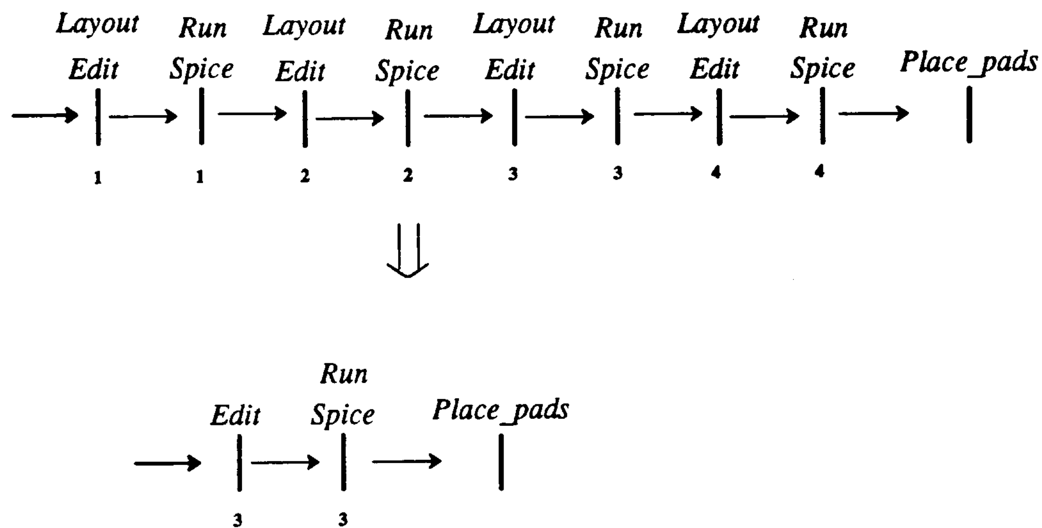


Figure 5.9 Abstraction of An Iterative Process

The other type of abandoned history elements is related to *dead-end branches*. A design thread typically has branching structures that represent alternative development paths. Some of these are no longer needed because the corresponding design alternative is found to be inadequate. The garbage collector recognizes these kinds of branches by maintaining a list of branches that contain at least one of the frontier cursors, together with their last access time. A frontier branch is marked as a dead-end when the difference between the last access time and the current time exceeds a certain threshold. Again, for safety reasons the activity manager will actively ask for users' approval when attempting

to eliminate these abandoned history elements.

5.5 Summary

In this chapter, we described the implementation issues of the design activity manager and the solutions developed in *Papyrus* are described. The design activity manager has to present a graphical view of the control stream of design threads for users to manipulate the design history. In particular, both browsing and query mechanisms are provided to examine the design history and to access specific design points. It also needs to maintain the internal structure of the design history and computes the data scope to support the rework mechanism. The computation of data scope is facilitated by caching intermediate thread states, which in turn lead to cache consistency problems when new history branches are created. Lastly, through object reclamation, the design activity manager attempts to reduce the storage overhead due to the single assignment update semantics. Because the history of design threads are stored persistently, it is possible to apply the design history information for other purposes than supporting rework. In the next chapter, we will describe a particular application of design history: automatic inference of design metadata.

Chapter 6

Automatic Metadata Inference Based on Design History

6.1 Introduction

Engineering design data is comprised of *internal* and *external* structures. Internal structure are the data structures and data formats used to represent design objects, e.g., ASCII texts for source codes and graphs for VLSI layout. External structure denotes the abstract attributes extracted from internal representations, and the interrelationships among design objects. First-generation engineering design databases [HARR86] focused on the storage and retrieval of the internal structures of design objects while second-generation design databases [CHAN89] provided a richer set of primitives to define, create, and manipulate inter-relationships among design objects. The next evolutionary step along this line has been called *object management systems* (OMS). The central notion of an OMS is an *object* abstraction, whose granularity is independent of files in a file system. Objects are logical entities as seen from the applications while files are

physical storage units. Users or applications access objects through access interfaces provided by an OMS rather than traditional file system calls.

Two central data structures supported by an OMS are *type* and *relationship*. A type system allows categorization of object behavior, and provides certain degrees of protection depending on how "strong" the typing mechanism is. The type of an engineering object is an intrinsic property of that object and typically can be determined at its creation time. For example, a layout object, when created from an automatic placement and route tool, is an object of type "layout", which in turn implies that only a specific set of tools can be applied to it. A relationship management system of an OMS treats inter-object relationships as first-class objects and manipulates them directly, rather than indirectly through objects involved in the relationships. First-class relationship objects offer more powerful modeling capabilities than previous systems, as well as opportunities for physical storage optimization, such as object clustering [CHAN89]. In summary, *typing* is a powerful mechanism to abstract the semantics of individual objects and *relationships* are particularly useful in capturing the interaction among objects.

Current object management systems [CLEM85] [SKAR86] [HUDS88] [DUEX90] [KIM90] support these two concepts to a varying extent, with the assumption that object types and inter-object relationships have somehow been established. However, it is not clear how type and relationship information are created in the first place. We use the term *meta-data* to refer to object types, per-object attributes, and inter-object relationships. Users of the above systems are required to supply meta-data themselves, which seems to be both disruptive and error-prone [CHAN89]. To solve this problem, we propose a novel design data management paradigm based on a design's history, from which design meta-data can be deduced and maintained in a completely transparent fashion.

This approach can be seen as a generalization of the recent trend in data management systems development: more and more application-specific semantics associated with data are captured and exploited by the system. In VLSI design, most work is performed through invoking CAD tools. Accordingly VLSI design database systems evolve to manage both design data and design processes. As a result it becomes possible for the underlying object management system to provide better services by exploiting the

semantics of data objects and design tool executions. We build on this observation and propose an automatic approach of building up design metadata by capturing and exploiting how design objects are created and manipulated.

The rest of this chapter is organized as follows. Previous research efforts that are related to this work are reviewed in the second section. Section 6.3 introduces two different approaches towards representing design history, and sets up the stage for explaining meta-data construction algorithms. The algorithms for constructing design meta-data based on this framework are presented in Section 6.4. Examples will be shown to illustrate the power of the proposed paradigm. Section 6.5 concludes this Chapter with a summary of major research developments.

6.2 Related Works

In this section, we summarize some previous attempts at building up meta-data automatically. Xerox's Cedar [SWIN86] has an automatic *make* facility called *MakeDo* that can determine the dependencies among source code objects by examining the files contents. In other words, the makefile for a software system can be transparently synthesized according to the internal references among software modules. Moreover, *MakeDo* has a limited capability of inferring the re-build procedure from the dependency relationships among design objects. IDEAS [MEHM87][TAYL87] offers a design file tracking mechanism in which the system records a set of attributes during creation of a design object. Among these are the type, the *source,* and *destination* attributes. The *source* attribute records the input files and the operations involved in creating the object, thus implicitly constructing a re-build procedure for this object. The *destination* attribute maintains where the object is used, thus providing information about all the downstream objects that will be affected by a change of this object. By combining these two attributes, the system can locate all objects affected by a modification and automatically rebuilds them transparently. VOV [CASS90] is an automatic design manager that is based on the concept of *design traces,* which are operation records left by circuit designers. By recording design operations in a bipartite acyclic graph, a "retracing" facility can re-

execute portions of recorded operations when a modification is made, thus bringing related design configurations into a consistent state. A *design trace* is basically a data dependency graph, augmented with the operations that are involved in creating these dependencies. These three systems aim at maintaining the consistency among design modules across modifications by tracking dependency relationships. No other meta-data are built automatically.

Language-directed editing environments like the Cornell Synthesizer Generator [REPS83] incrementally construct the internal structures of a program. The central formalism used in syntax-directed editing is *attribute grammar*, which captures language-specific semantics through semantic equations associated with context-free grammar rules. The internal representation of language-directed editors is an annotated program parse tree. An optimal attribute evaluation algorithm [REPS83] was developed to incrementally update the attribute values associated with the nodes of a parse trees that are affected by a modification. Cactis [HUDS88] allows specifications of attribute evaluation rules in terms of local attributes within an object and remote attributes carried through inter-object relationships. The system implements an incremental evaluation algorithm for handling the attribute evaluation dependency graph

In essence, our work applies the same idea used in syntax-directed editors at a higher level of abstraction to incrementally build up useful information for design objects. Because the meta-data inference algorithm is based on the notion of design history, the history representation model will be discussed first in the next section.

6.3 Models of Design History Representation

Recording what users did in a form of history has been known to be a useful mechanism for providing operation REDO/UNDO and other services [LEE92]. Some command interpreters provide some sort of history mechanism for reusing earlier actions, possibly with some modifications, e.g., the *history* facility in C-shell. Unfortunately the power of the history mechanism is not fully explored in these systems. One of the major contributions of this chapter is to show that it is possible to infer useful meta-data from

recorded design operation history. In this section, two representation models of design operation history --- *operation-oriented* and *data-oriented* --- are presented and compared.

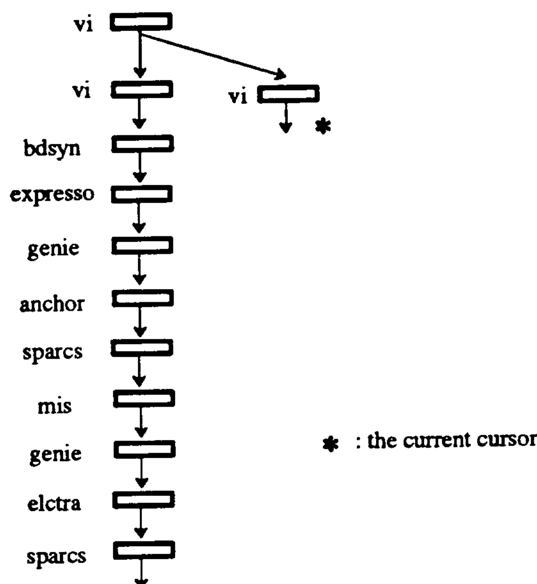


Figure 6.1 The *Operation-oriented* Design History Representation

The *operation-oriented* model focuses on the performed design operations, the records of which form the control stream of a design thread. C-shell's history facility is an example, but it only allows linear control streams, where commands are ordered by temporal sequencing of the operations. As shown in Figure 6.1, a thread's control stream (for a session of VLSI design) consists of an sequence of interleaved **history records** (rectangle) and **design points** (arrow). A history record corresponds to a design operation including the tool name and its execution options. Because of the rework mechanism explained in Chapter Three, a thread control stream can actually have a branching structure. Consequently history records in the operation-oriented representation model are normally stored in the temporal order in which their corresponding design operations are executed. Users can override that order by moving the current cursor to create branches. *Temporally* adjacent design operations in a thread don't imply data dependency

relationships among them, although they usually do. This history representation preserves high-level design development patterns as branching design operation sequences. More importantly, it supports the idea that a design can be rolled back to an earlier state, providing a powerful mechanism for exploring the design space.

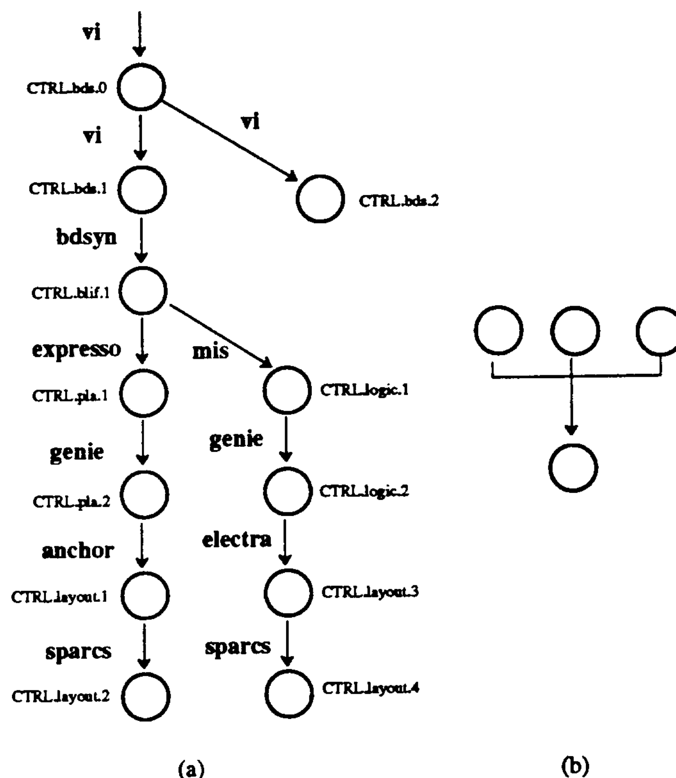


Figure 6.2 The *Data-oriented* Design History Representation

The *data-oriented* design history focuses on the *use* and *used-by* relationships, i.e., data dependency relationships, among design objects, supplemented by the design operations involved in creating these data dependencies. The data-oriented history representation is independent of the temporal order of their execution. We call the data-oriented history representation an *augmented derivation graph* (ADG). Figure 6.2(a) is the corresponding ADG of the activity control thread in Figure 1. Each circle in an ADG is a design object and each arrow represents an instance of a CAD-tool invocation, which includes the control parameters and auxiliary files. It is a graph because an object can

have more than one input and it itself can be used as an input in more than one place. Figure 6.2(b) shows the graphical representation of the case when a design operation requires more than one input. VOV's design trace [CASS90] is an explicit form of ADG while IDEAS's source/destination attributes [MEHM87] implicitly contain all the information in an ADG.

In contrast to most history database systems, these two representation models are based on operation logging rather than on snapshots of previous database states. We believe that the paradigm of capturing and analyzing operation history is much more powerful than simply maintaining history snapshots. This is particularly true in situations where the semantics of operations can be precisely characterized, as in a VLSI design environment.

It is interesting to contrast these two design history representations with data/control flow graphs used in language compilers. Thread control streams exhibit high-level development patterns just as control flow graph shows how control is passed around in a program. Augmented derivation graphs maintain data dependency relationships among design objects as data flow graphs, plus the operations creating the dependencies. It should be emphasized that activity control threads and augmented derivation graphs are really separate logical concepts, although they can be implemented in one physical data structure. Both are essential from a design management system point of view. the properties and usage of thread control streams have been examined in Chapter Three. In the next section we will focus on how meta-data can be inferenced from the augmented derivation graph of a design's history.

6.4 Incremental Meta-data Construction

Building the external structure of design objects while users create and manipulate them has been pioneered by language-directed program editors [REPS83], in which abstract program representations are computed as users edit the program sources. This *immediate computation* paradigm not only eliminates unnecessary declaration and manipulation on the part of users, but also makes efficient use of the otherwise unused

hardware cycles. Based on the same idea, we propose a new design data management paradigm. Instead of requiring users to explicitly supply meta-data, the system observes what users did and automatically infers meta-data without user intervention. Underlying this new paradigm is the augmented derivation graph of a design's history. Our approach infers and constructs meta-data incrementally from an ADG, taking into account such domain knowledge as the semantics of design objects and CAD tool executions.

One can draw an analogy between an augmented derivation graph and an annotated parse tree representation of a program, with each tool execution corresponding to an instantiation of a grammar rule, and the outputs and inputs of a tool execution to the left-hand-side and right-hand-side of a grammar rule, respectively. In Figure 6.3(a), **b, C,** and **d** are syntactic elements that are aggregated into **A** when the grammar rule is instantiated; in Figure 6.3(b) the tool execution takes three input objects, **b, C,** and **d,** and produces an output object **A.** Just as the semantics of a language are encoded in the semantic equations associated with attribute grammar rules, the semantics of tool executions are captured in terms of how abstract information about design objects are evaluated as a side effect of tool executions. In language-based editors, attributes are evaluated as a result of the instantiation of grammar rules during the incremental parsing process. In our approach, meta-data such as attributes and relationships, are established as a by-product of CAD tool executions.

Figure 6.3 The Analogy Between A Tool Application and A Grammar Rule

6.4.1 Type Inference and Attribute Evaluation

The first kind of information about a newly-created design object is its *type*. In VLSI design, an object's type can usually be inferred from the tool that creates the object. For example, the output of a behavior-to-logic translator is an object of type *logic*. With the type information, the system can detect incompatible tool applications, e.g., invoking a layout compaction tool on a logic object. Specifications of the type of a tool's output are encapsulated in a data structure associated with each CAD tool, which we call the *tool semantics description* (TSD). An object of a certain type typically has a specific set of attributes that are useful in abstracting the object's behavior. For example, an object of type *layout* can have attributes such as the critical path delay and area, and a logic object can have attributes such as the number of minterms and worst-case delay. These type-specific attributes of an object provide a concise representation for that object.

For practical purposes, there are three kinds of attributes: *administrative, intrinsic,* and *propagated*. *Administrative* attributes refer to those attributes that are conventionally supported by the file system such as the owner, time of last modification and access control information. We won't consider these attributes any further in this chapter. *Intrinsic* attributes refer to those attributes whose values are evaluated by applying some measurement tools, e.g., the power consumption of a layout object, the area used by a logic object implemented in Programmable Logic Arrays (PLA). These attributes are *intrinsic* because their values depend only on the object itself. The values of *propagated* attributes, in the most general form, depend on both the object itself and other objects that have certain relationships with the original object. A typical example is the area attribute of a composite layout (an object has other objects as submodules) is the sum of the areas of contained objects and that of its own, including the interconnections.

When an object is created (it is called the *triggering* object hereafter), the system examines the TSD of the creating tool to determine the object's type. By consulting with the type specification associated with the object's type, a set of attributes are automatically *attached* to the object. The type specification also contains the procedures to compute intrinsic attributes. These procedures are deposited into the object when the

attributes are attached. In our system, the values of intrinsic attributes can be either evaluated explicitly using the measurement tools or inherited from the inputs used in creating the triggering object. For example, running a two-level logic minimizer on a logic object implemented in PLA form could change the number of minterms (the length of PLA) but not the number of inputs and outputs (the width of PLA). Included in each tool's TSD is an *inherit* list, which specifies the set of attributes that can be propagated from inputs to outputs through the tool execution.

Values of the intrinsic attributes that are not in the inherit list of the creating tool's TSD need to be re-computed. Computation of these non-inheritable attributes are completely transparent to users and can take place either *lazily* or *immediately*. Lazy evaluation means demand-driven, i.e., attribute values are computed only when they are actually needed. This mechanism is useful because attribute computation could consume substantial resources and therefore should be performed only when absolutely necessary. Immediate evaluation means data-driven, i.e., attribute values are computed when data arguments becomes available. This is typically used in constraint attributes, where constraint violation should be detected as early as possible, or in index attributes, whose values are needed to put the triggering object in the index. The computation mode of intrinsic attributes is part of the type specification.

Figure 6.4 shows the Tool Semantics Description for the two-level logic minimizer *espresso*. This TSD declares that the output of *espresso* can be either a logic object of algebraic equation format or a logic object of PLA format, depending on the parameter setting when the tool is invoked. The *inherit* list enumerates the attributes that won't be affected by this tool, i.e., they can be propagated from inputs to outputs. In this example, the number of inputs and outputs won't be affected. The composition tool flag indicates whether or not this tool is a composition tool. The execution semantics vector specifies the transformation semantics of this tool. We will discuss composition tool flags and execution semantics vectors next.

Values of *propagated* attributes depend on *local* (attributes of the triggering object) as well as *remote* (attributes of other objects) attributes. Typically remote attributes belong to objects that are related to the triggering object through certain relationships. In

composition tool flag : NO

execution semantics vector :

behavioral : *1*

logic : *0*

physical : *0*

output object type/format :

-o eqntott: logic/equation

-o pleasure: logic/PLA

inherit list :

number of inputs

number of outputs

•

•

Figure 6.4 Tool Semantics Description of *espresso*

practice, propagated attributes are almost always used to propagate semantic information *up* and *down* the configuration hierarchy. In other words, the remote objects on which an object's propagated attribute depends either contain as components, or are components of the latter. An example of propagating information up the design hierarchy is that the power consumption of a composite layout object is the sum of those of the component objects, plus its own. An example of propagating information down the design hierarchy is that the I/O specification (e.g. pin mask layer) of a component should conform to part of the I/O specification of its composite object.

Unlike Cactis [HUDS88], where propagated attribute evaluation rules are encapsulated with design objects (Figure 6.5(a)), our model associates propagated attribute evaluation rules with the relationships, as shown in Figure 6.5(b), where the bold arrow indicates that it is the evaluation rules associated with the physical type that are triggered. That is, each relationship can have a default set of semantic equations that evaluate the

propagated attribute values of the objects that are involved in this relationship. Moreover, the evaluation rules are tailored to the type of objects involved in the relationship. For example, a configuration relationship among layout objects may have a different set of evaluation rules than those of configuration relationships among logic objects.

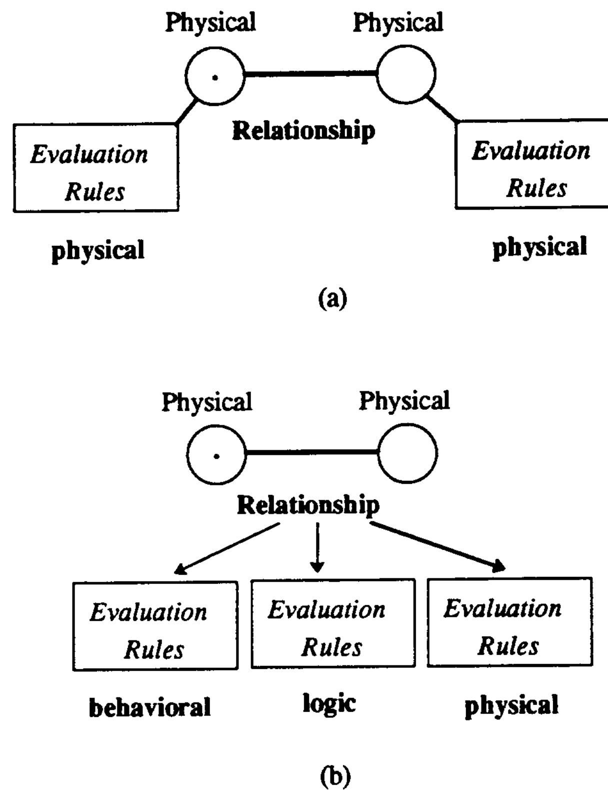


Figure 6.5 Encapsulating Evaluation Rules with (a) Individual Objects, or (b) Relationships

Encapsulating evaluation rules within relationships objects rather than within design objects has the following advantages. First, specifications of evaluation rules can be shared among all objects that have the same kind of relationship with others. Second, it becomes possible to transparently compute propagated attributes without the user's registration of evaluation routines because there is always a default set of evaluation rules available. Third, by keeping evaluation rules with relationships, one can essentially treat a design configuration hierarchy as a program parse tree, with each configuration relationship corresponding to a derivation step in syntax-driven editors. Consequently, the

incremental attribute evaluation algorithm developed in [REPS83] becomes readily applicable to re-compute the attribute values in response to modifications to design objects. The disadvantage of this approach is that the evaluation rules can not accommodate object-specific attribute evaluation rules, and thus is less expressive. However, those attributes probably should be managed by individual CAD tools anyway, rather than by the underlying object management system.

In summary, when an object is created, the object's type is determined based on the creating tool's TSD. The set of attributes associated with that type are then attached to the object. Values of intrinsic attributes are evaluated in the background, either lazily or eagerly. Propagated attribute values are evaluated only after the object's relationships with other objects have been established. Once established, the relationships together with the object's type determine the appropriate set of evaluation rules for computing the propagated attributes. The important point is that *all these computations take place completely transparently to the users*. From the user's standpoint, once an object is created, s/he can assume that its type information and associated attributes are automatically established.

6.4.2 Relationship Establishment

The *Version Server* [CHAN89], developed at U.C. Berkeley, supports the creation and maintenance of three kinds of relationships: *version, configuration,* and *equivalence.* Unfortunately these relationships must be made known to the system *by users* through procedural interfaces or interactive commands [CHAN89]. Because it is the user's responsibility for relationship declarations, the design process tends to be interrupted by these bookkeeping operations when new objects are created. Moreover explicit relationships declaration has the disadvantage of being error-prone because users may forget to declare these relationships. Therefore an automatic way of establishing inter-object relationships is very desirable. In this section, we show how these relationships can automatically be inferred from the augmented derivation graph of a design's history when supplemented with tool semantics descriptions.

Let us define the meanings of version, equivalence, and configuration relationships as used in this context. As in the Version Server, we assume an object's name is of the form *Entity_Name.Representation_Format.Version_Number*, e.g., ALU.logic.1. However, our algorithm applies to other naming formats. Version numbers are automatically generated by the system when an object is created. Version relationships relate ancestors and descendents. An object is a *descendent (ancestor)version* of the other if the former is derived from the latter (or vice versa), and they are of the same entity-name and representation. Two objects are *equivalent* if they are just different representations of the same logical entity and one is directly derived from the other. For example, a behavior description of a circuit module is equivalent to a logic equation description of the same module that is derived from it, e.g., ALU.logic.1 and ALU.layout.2. The difference between version and equivalence relationships lies in whether the transformation that relates two objects changes the underlying data representation. For example, an object created by applying logic minimization on a logic equation object is a version of the latter, whereas an object created by applying a PLA layout generator on a logic equation object is an equivalence. Note that *version* is a non-associative relationship while *equivalence* is an associative relationship. *Configuration* relationships express is-a-component-of and is-composed-of relations among design objects. For example, a register-file object is composed of a decoder object and an array of register objects.

The goal is that when a new object is created, the version, equivalence and configuration relationships in which this object participates are automatically extracted from the augmented derivation graph. For equivalence and version relationships, the basic algorithm, shown in Figure 6.6, performs a backward breadth-first-search traversal of the augmented derivation graph starting from the triggering object. When visiting each node, the algorithm checks whether this object has the same name and representation as the triggering object. If so, the algorithm finds the triggering object's immediate ancestor version, establishes a version relationship between this object and the triggering object, and stops. If the visited object has the same name but with a different representation, then this object is an equivalence of the triggering object. The algorithm stops in the following situations:

```
Start_Breadth_First_Traversal(Triggering_Node);

While (Not Stop_Criterion()) {

   Node = Next_Breadth_First_Traversal();

   Check_For_Version(Node, Triggering_Node);

   Check_For_Equivalence(Node, Triggering_Node);

}

If (Exist (Anode = Ancestor(Triggering_Node))

      Establish_Inherited_Equivalence(

         Anode, Triggering_Node);
```

Figure 6.6 Version and Equivalence Relationships Algorithm

[1] When the algorithm reaches an arrow of the ADG that doesn't have inputs. This means that the tool represented by the arrow doesn't have inputs, e.g., an object created from an editor from scratch.

[2] When the immediate ancestor version is found, then there is no need to perform further traversal. Depending on the execution semantics of the tool execution sequences between the triggering object and its immediate ancestor, the former may be able to *inherit* some of the equivalence relationships from the latter.

[3] When an arrow has multiple inputs, i.e., the tool executes a "composition" operation. For example, a layout module can be constructed from several submodules through a macro-cell placement and route tool. In this case, the algorithm stops, knowing that the triggering object does not have an immediate ancestor because it is composed from lower-level modules.

Note that the inheritance mechanism used in [2] is more general than those proposed in [CHAN89] because it takes the tool execution semantics taken into account. As shown in Figure 6.4, the execution semantics of a tool execution is represented as a vector, with each bit corresponding to a particular type of representation in the system. In our system, only three types **behavioral, logic,** and **physical** are supported. The algorithm described below, however, is applicable irrespective of the type system. The *execution semantics vector* of a CAD tool specifies whether this tool changes the semantics in the respective representation. For example, a logic minimizer "espresso" will not change the behavioral-level semantics, but will change the logic-level and physical-level semantics. Therefore, its execution semantics vector is 100. In general, the execution semantics of a tool execution sequence S that contains $T_1, T_2, .. , T_n$ is defined as

$$\mathbf{ESV(S)} = \underset{\mathbf{i=1}}{\overset{\mathbf{i=n}}{\mathbf{AND}}} \mathbf{ESV(i)}$$

where ESV(i) and ESV(S) are the execution semantic vectors of the i-th tool and the whole tool execution sequence, respectively. A 1 entry in an ESV means that the tool execution sequence preserves the semantics along the corresponding representation between two objects, and therefore allows equivalence relationship inheritance in that representation. Given that the ESV of the tool execution sequence between the triggering object and its immediate ancestor, the triggering object can inherit the equivalence relationships of its ancestor for those types whose corresponding bits in ESV are 1. In our example, assuming that ESV(S) is 100, then the triggering object can inherit its ancestor's equivalence relationships along the behavioral domain, but not logic or physical domains.

For practical reasons, it is possible to selectively trigger the relationships establishment procedure. Users can customize their environment by declaring beforehand that the relationships establishment procedure will be initiated only when the newly created object's data format belongs to a particular set. The same idea can be used to reduce the number of equivalence relationships associated with an objects. That is, only certain kinds of objects can have equivalence relationships with others. This selectivity mechanism is important in a VLSI design environment, where a great variety of data formats are

used and most of them are intermediate temporaries, so it is not necessary to keep track of their relationships.

Simply examining the augmented derivation graph is not enough to infer the configuration relationships. The problem is that in addition to the class of "composition" tools, interactive editors can also include submodules, which are not observable from the augmented derivation graph. The system uses the following approaches to establish configuration relationships.

[1] If the creating tool is a "composition" tool, i.e., the compositional tool flag is set to "yes," then a configuration structure object is created with the inputs to this tool as components.

[2] If the creating tool can potentially build up configuration relationships, e.g., an editor, then the system will actively ask users to explicitly specify the component objects included during the editing session. This is the only place in our system where user intervention is needed to build up meta-data. Configuration relationships need not be built up all at once. They can be constructed incrementally as the corresponding design object evolves.

As an illustration of relationships establishment, Figure 6.7 shows part of the meta-data constructed out of the augmented derivation graph shown in Figure 6.2. Note that the equivalence relationship is transitive except when the objects in question have the same name and representation. Therefore CRTL.pla.1 is equivalent to CRTL.bds.1, but CTRL.pla.1 and CTRL.pla.2 are not equivalent. Also note that CTRL.layout.2 inherits the equivalence relationship from its immediate ancestor CTRL.layout.1 because the ESV of the compaction tool "sparcs" is 110, meaning that it is legal to inherit the equivalence relationships along the behavioral and logic domains. CTRL.layout.3 and CTRL.layout.4 form a separate version derivation path because they are completely independent of the path formed by CTRL.layout.1 and CTRL.layout.2. Other relationships can be similarly deduced.

In summary, when an object is created, the system consults with the creating tool's TSD to establish the associated configuration relationship, and a backward traversal of

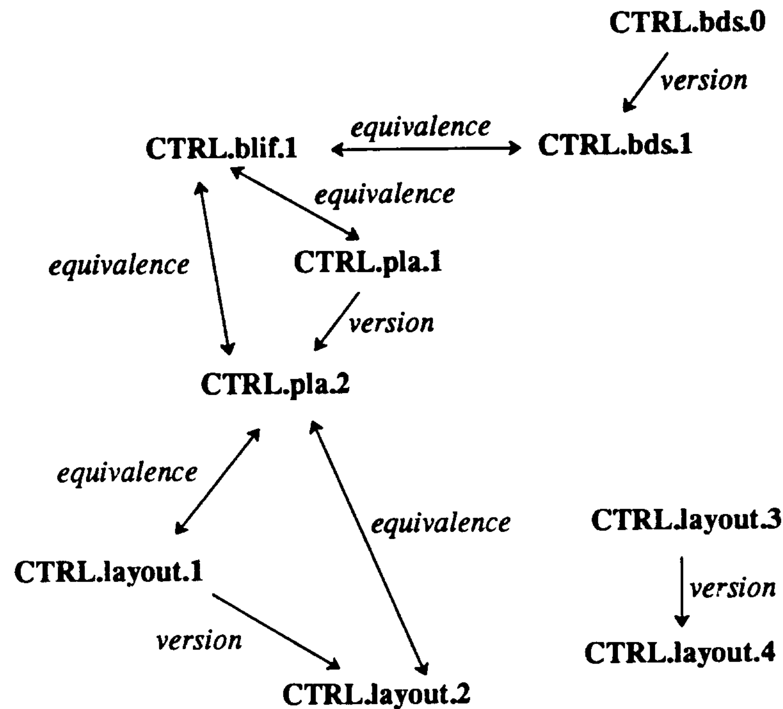


Figure 6.7 Established Relationships for ADG in Figure 6.2

the augmented derivation graph starting from the triggering object is initiated. During the traversal, associated version and equivalence relationships are created by checking each visited node. Finally, the equivalence relationships that could be inherited from the triggering object's immediate ancestor, if any, are located and established.

6.4.3 Discussion

In a broader sense, the idea of using history information to predict systems behavior has been around for a long time. The various virtual memory replacement policies such as least-recently-used and FIFO are just an example. Except in those cases, the history of the objects in question is compressed in a form that can speed up the processing, perhaps at the expense of prediction accuracy. In our case, since we need to maintain the design history for rework and reuse anyway, the additional effort to reap the benefit of

extracting useful information from design history is considered well justified.

On the down side, one of the limitations of this paradigm is that the system has virtually no knowledge of the behavior of interactive CAD tools because of the difficulty of specifying the semantics and of tracing the I/O behavior of interactive tools. However, as VLSI design evolves towards higher levels of abstraction, this limitation is expected to be less of a problem. Alternatively, one can use the so-called *embedded tool encapsulation* [KRAF91] technique to intercept the interactions between interactive tools and the file system, and records the data dependency relationships among design objects. However, this technique requires modifications to the operating system kernel, and is thus less portable.

The other criticism about this approach is that the system needs extra knowledge to analyze the design history. In our case, they are tool semantic descriptions and data type specifications. However, as we indicate earlier, there seems to be a trend to endow on the data management system more and more domain-specific semantic knowledge. It seems to us that the question is really *how easily* this knowledge can be embedded into the system, rather than *how much*. That is, it is more of an interface issue than an architectural one. We also found that this paradigm can be applied to solve some of the old problems encountered before. For example, based on the history information, a version management system can make a more intelligent decision when it needs to restrict the extent of constraint and change propagation, e.g., by analyzing their dependency relationships. As another example, automatic integrity validation can be performed on a design object that is to be checked into an archive, based on the object's derivation history. It would be interesting to explore the possibility of structuring the history mechanism in a more modular architecture so that it can serve as a unified framework for various high-level design management services.

6.5 Conclusion

Because of extensive usage of CAD tools in VLSI design, it is possible for the design management system to record the design history and infer useful information from it, taking into account the tool execution semantics. We show that there are two kinds of design history representation: *thread control stream* and *augmented derivation graph*, and argue that both are essential for high-level design management. The activity control thread offers a clean conceptual model for users to manage the design process. The augmented derivation graph provides a unified framework from which useful meta-data can be inferred. We apply the *immediate computation* paradigm as advocated by language-directed editor researchers to engineering object management systems by treating each tool execution as an editor command and the whole object base as an interrelated network of objects. Based on this idea we are able to infer the meta-data of design objects both incrementally and automatically. We examined three kinds of relationships supported by the Version Server in greater detail. The semantics of design operations, captured by *tool semantic specifications*, are used to determine the type of a tool's outputs, the set of attributes that can be inherited from the inputs by the outputs, to infer the existence of configuration relationships, and to decide whether equivalence relationships can be inherited from the ancestor by the descendent version.

In conclusion, in this chapter we have described an interesting application of design history information. This approach offers a novel design data management paradigm: Instead of requiring users to tell the system what meta-data are, the system simply looks at what the users did and infer meta-data from the recorded history. During the inference process, domain knowledge in the form of tool execution semantics are taken into account to resolve potential ambiguities. To the author's knowledge, this is the first attempt at exploiting history information in ways other than providing operation REDO/UNDO. The power of this paradigm scales with the amount of tool/object semantics that can be specified in advance. As more refined semantic specifications of tools and/or objects are made known to the system, we believe the proposed approach could build up more varieties of sophisticated meta-data in an automatic fashion.

Chapter 7

Conclusion and Future Works

7.1 Research Contributions

With the advent of powerful computer-aided-design tools and increasing complication of VLSI systems, the notion of circuit design evolves towards managing complexity rather than manipulating electronic devices. The complexity in question includes those inherent in design data as well as in the process of creating the design data. The latter is attributed to the growing variety of CAD tools that are optimized for specific technologies and design methodologies, and may very well come from distinct developers with different interface styles. Based on the belief that the key to further enhancing the designer productivity does not lie in better CAD tools but a more responsive infrastructure, this thesis represents the first attempt at alleviating the design complexity associated with design data and processes from an *environment* perspective. Therefore this thesis is not about specific CAD tools that can optimize a specific aspect of VLSI design. It is about support mechanisms that allow composition of a set of potentially heterogeneous tools into a coherent design system, and that facilitate the integration of design data and process management.

From the very beginning of the project, we have recognized the fact that there is a fundamental tradeoff between design generality and degree of design automation. This is evidenced by the history of high-level synthesis research, where significant success can only be secured by focusing on a specific class of design applications. As a result, we adopt an *assistance* approach to solve the design complexity problem. That is, the system helps circuit designers to accomplish certain design tasks, but it is the circuit designers who have to decide what these tasks are. The rationale of this decision is the belief that deciding what to do is what creativity is all about and cannot be delegated to machines in the foreseeable future; however, automating or facilitating the *how-do-do* process is where computers are superior to humans.

For circuit designers, the design complexity manifests itself in two forms: which tools are most adequate to achieve a given objective, and what data objects are relevant to a given task. The first problem is exacerbated by the unfortunate fact that a CAD tool typically has dozens of command options that allow customization of a specific aspect of the tool's behavior. This makes correctly invoking a CAD tool a non-trivial matter. The second problem is related to version management, but in a different light. The issue is how to identify a set of object versions (or a configuration) during exploration of the design space, or more specifically, how to relate configurations to the high-level design alternatives they embody.

To address these two problems simultaneously, we take a *semi-structured* view towards the VLSI design process: There are portions of a design process that are well-understood and therefore can be specified in advance, and then there are the other portions of a design process that involves exploration and refinement of various design alternatives. Based on this design process model, a set of support mechanisms and an implementation of these mechanisms are described in thesis. More concretely, the research contributions of this thesis are:

- We developed a design process support model called the *Light Weight Transaction* (LWT) model. A design task construct is devised to automate or to navigate the users through the well-understood parts of the design process. A design thread

construct reinforces the notion of contexts and provides the basis for exploring the design space. The central notion of LWT is *data visibility*: users can operate on a data object only when it is visible. By controlling the portion of the design data space that is visible, this model provides a unified conceptual abstraction for exploring the design space and for coordinating group work.

- We proposed a conceptually intuitive and semantically powerful history mechanism to organize VLSI design processes. The unique feature of this approach is that the structure of the design history need not be linear; it can have branches. Operations are provided for users to organize their design history in such a way that reflects the high-level development patterns of various design alternatives. This history mechanism adds a new dimension to design process management in particular, and graphical user interface in general. Just as the desktop serves as a spatial metaphor for organizing graphic objects on the workstation display, the branching history offers an interesting temporal metaphor for users to manage their computation activities.
- We demonstrated the ideas proposed in the *Light Weight Transaction* model with a prototype implementation that is built on top of the *Sprite* operating system, the Tcl/Tk facility, and the Berkeley OCT CAD tool suite. This implementation features an transparent load balancing scheme to exploit the computation power of networked workstations, an atomicity-guarantee mechanism to preserve the high-level abstraction of the design task construct, and a set of storage management techniques for supporting the single assignment update semantics.
- Based on the design history, we proposed a novel design management paradigm. Rather than requiring users to supply design meta-data, the system maintains and analyzes the design history to deduce the metadata, in particular, object attributes and inter-object relationships, according to a suite of domain-specific knowledge and inference procedures. This paradigm actually represents a generalization of the approach used in syntax-directed editors. However, we apply the idea in the context of design database management systems. Instead of using abstract syntax trees, a special representation of design history called *augmented derivation graph* is used

as the basis for design metadata inference. This paradigm opens a new way of thinking about creating information that are interesting to the system, be that a user, an operating system or a database system. The difference lies only in the suite of domain knowledge and inference algorithms.

Although the idea of using history as a unified mechanism in *Papyrus* is unique among contemporary systems that intend to address the same problem, in retrospect it seems to be just a simple combination of C-shell history facilities, version management, and the generalization of history-based prediction techniques (e.g., LRU). Taken together, however, it presents a clean abstraction to the users and a useful mechanism for the system to perform metadata inference. Of course, it is foolish to claim that a single piece of research such as this can address all the problems in managing design complexity. Some advancements have been made, but there is still work to be done. The following section documents some personal thoughts about how a future VLSI design system will evolve and other lines of research inspired by this work.

7.2 Future Directions

7.2.1 What's Next in VLSI Design Systems?

If one is to review the history of electronic design systems, there is a distinct pattern: VLSI CAD systems is always one generation behind what is available in software engineering environments. From storing design objects, managing versions, to supporting processes, computer-aided software engineering systems seem to be invariantly one or two steps ahead than their electronic contemporaries. Therefore the natural thing to do when speculating the next generation of electronic CAD systems is to see what are the research issues in today's software engineering environments.

Until recently, most of the CAD systems research focused on the *design* aspect of the problem. After all this is where the D in CAD comes from. But if one takes a life-cycle view about an electronic product, it becomes clear that design is only one phase of the cycle. There are also other phases such as maintenance and improvement. Whereas

supporting design activities emphasizes the management of data and processes, maintenance and improvement stress how to make it easier to understand a design. And that calls for an electronic design system to provide support for documentation. To provide documentation support is to help users create and to view documents. In the broadest sense, design objects are themselves documents. The process of documentation involves creating documents and *associating* them in such a way that it is possible to traverse from one to the other and vice versa. The process of viewing documents involves choosing a particular presentation order out of the set of related documents. Recent developments in hypertext systems have provided the necessary technology to make it possible to embed a documentation system into a VLSI design environment. Initial research in this direction has been reported in [SILV92]. A related issue to documentation is about design library and archive management. Once facilities for documenting designs become available, it is natural to apply these facilities to increase the possibility of reuse at various granularities, i.e., from cells to macro-blocks to chips.

The other possibility is to focus on the managerial aspect of VLSI design, i.e., tools and techniques developed to facilitate the management of a VLSI project rather than to design circuits. In principle there is no difference between managing a VLSI design project and other technology-oriented development projects. In practice, however, there are two differences. First, most VLSI design teams are relatively small, i.e., less than ten members. This implies a flat management hierarchy, e.g., only one level deep. Also most teams tend to be self-managing. Computer support becomes more effective in this environment because there is a close interaction between managers and team members. Second, in VLSI design most of the work is carried out by running CAD tools. The project progress is much easier to monitor and predict. In particular, the design history mechanism can serve as an monitor aid for keeping track of current projects, and as a useful hint for similar projects in the future. Although project management tools are commercially available, there are few, if any, that actually exploit these two characteristics.

7.2.2 Other Lines of Research

One of the most important design decisions made in the *Light Weight Transaction* model is the single assignment update principle: No updates are performed in-place; new versions of the given object are created instead. While most researchers view this approach as an expensive way of spending disk storage, it is actually rather feasible considering a 4 GByte helical scan tape only costs about $10 and a tape drive costs less than $2000. This assumes an automatic migration daemon that keeps only the most recent version of data objects on the disks. With this approach come a set of interesting possibilities, for example, time travel between different states, automatic versioning, and log-structured storage layout. There is no need for doing separate backup. The entire disk is just a cache of the tertiary system based on tapes. This concept in essence requires the file system to provide versions as first-class objects. There is another reason for supporting first-class versions. In environments where clients are not connected to the file server all the time, e.g., portable computers without wireless connections, creating new versions is the only logical alternative to allow continuous operations. The systems-supported versioning mechanism is actually available in earlier systems such as TENEX and VMS. The difference, though, is that these systems do not perform automatic migration.

The other interesting system research direction is to explore exactly how useful operation history is and how expensive to maintain them inside the kernel. As we discussed earlier, operation history contains potential information that may be useful for various parts of an operating system. Moreover, if user applications also need to access operation history for various reasons as outlined in [LEE92], it seems more reasonable for the operating system to provide history as a built-in service, rather than to let each application duplicate the effort. The main thrust of this research direction is to explore the potential ways in which operation history can be productively put to use, and to devise a flexible interface to a history mechanism supported directly by the operating system.

7.3 Final Words

A piece of PhD-thesis research is about a vision. The vision underlying this work is that it is possible to use a simple script and history mechanism to help circuit designers manage their design process. As all visions that are worthwhile, it is almost impossible to completely embody a new vision in one dissertation. I hope this work reported here represents a new beginning of an exciting research endeavor.

REFERENCES

[ALLE90] W. Allen, K. Fiduk, and D. Rosenthal, "Distributed Methodology Management for Design -in-the-Large," Proceedings of International Computer Conference on Computer Design '90, November 1991, pp. 346-349.

[BANC86] F. Bancilhon, W. Kim, H.F. Korth, "A Model of CAD Transactions," Proceedings of 11th International Conference on Very Large Data Base, August 1985, pp. 25-33.

[BANN83] L. Bannon, A. Cypher, S. Greenspan, M. L. Monty, "Evaluation and Analysis of Users' Activity Organization," Proceedings of the CHI'83 Human Factors in Computer Systems, pp. 54-57, Boston, Massachusetts.

[BATO85] D.S. Batory, W. Kim, "Modeling Concepts for VLSI CAD Objects," ACM Trans. on Database Systems, V.10, No.3, (September 1985), pp. 322-346.

[BUSH89] M. Bushnell, S.W. Director, "Automated Design Tool Execution in the Ulysses Design Environment," IEEE Transactions on CAD", Vol. 8, No.3, (March 1989), pp. 279-287.

[CARD87] S. K. Card, D. A. Henderson Jr., "A Multiple, Virtual-Workspace Interface to Support User Task Switching", Proceedings of the CHI'87 Human Factors in Computer Systems, pp. 53-59, Toronto, Ontario.

[CASO90] A. Casotto, R. Newton, A. Sangiovanni-Vincentelli, "Design Management Based on Design Traces," Proceedings of 27th Design Automation Conference, June 1990, pp. 136-141.

[CHAN89] E. Chang, D. Gedye, R. Katz, "The Design and Implementation of a Version Server for Computer-aided Design Data," Software-Practice and Experience Vol. 19

No. 3, (March 1989), pp. 199-222.

[CHIU89] T. Chiueh, R. Katz, V. King, "Managing the VLSI Design Process," UCB/CSD 89/538, University of California, Berkeley, Computer Science Division, (November 1989).

[CHIU90] T. Chiueh, R. Katz, V. King, "A History Model for Managing VLSI De sign Process," Proceedings of 1990 International Computer Conference on Computer Aided Design, Santa Clara, November 1990, pp. 358-361.

[CHIU92a] T. Chiueh, R. Katz, "Incremental Meta-Data Construction for VLSI Design Databases ," Proceedings of European Design Automation Conference, Brussels, Belgium, March 1992, pp. 399-403.

[CHIU92b] T. Chiueh, R. Katz, "Intelligent VLSI Design Object Management," Proceedings of European Design Automation Conference, Brussels, Belgium, March 1992, pp. 410-414.

[CHOW88] P. Chow, M. Horowitz, "The Design and Testing of MIPS-X", Proceedings of 1988 Conference on Advanced Research in VLSI, March 1988, pp. 95-114.

[CHU 83] K. Chu, et al. "VDD -- A VLSI Design Database," Proceedings of ACM SIGMOD Conference on Engineering Design Applications, San Jose, January 1983.

[CHRO90] G. Chroust, H. Goldmann, O. Gschwandtner, "The Role of Work Management in Application Development," IBM Systems Journal, Vol. 29, No. 2, 1990, pp. 189-208.

[CROF84] W.B. Croft, L.S. Lefkowitz, "Task Support in an Office System," ACM Trans. on Office Information Systems, Vol. 2, No. 3, (July 1984), pp. 197-212.

[DANI89] J. Daniell, S. Director, "An Object-Oriented Approach to Distributed CAD Tool Control," in IEEE Proc. 26th Design Automation Conference, Las Vegas, June 1989, pp. 197-202.

[FROM90] B.D. Fromme, "HP Encapsulator: Bridging the Generation Gap," Hewlett-Packard Journal, (June 1990), pp. 59-68.

[FRYD89] C. Frydman, N. Giambiasi, M. Gatumel, P. Bayle, "DeBuMa: Description, Building and Management of Applications," Proceedings of 26th Design Automation

Conference, (June 1989), pp. 203-208.

[GOLD88] K. Goldman, T. Stout, "A Design Automation Environment," In VLSI Systems Design, (June 1988), pp. 46-49.

[HARR86] D. Harrison, P. Moore, R. Spickelmier, R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment", Proceedings of 1986 International Computer Conference on Computer Aided Design, Santa Clara, November 1986, pp. 20-24.

[HOFF85] G. Hoffnagle, W. Beregi, "Automating The Software Development Process," IBM Systems Journal Vol. 24, No. 2, (1985), pp. 102-120.

[HUDS88] S. Hudson, R. King, "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", TR 88-20, Department of Computer Science, University of Arizona.

[JOHN89] W. Johnson, "Bringing Design Management to the Open Environment," High Performance Systems (June 1989), pp. 66-70.

[JANN85] A. Janni, "A Monitor for Complex CAD Systems," Proceedings of 23th Design Automation Conference, (June 1985), pp. 145-151.

[KAIS90] G. Kaiser, "A Flexible Transaction Model for Software Engineering," Proceedings of 5th International Conference on Data Engineering, pp. 560-567.

[KATZ87] R. Katz, R. Bhateja, E. Chang, D. Gedye, V. Trijanto, "Design Version Management," IEEE Design and Test, V01. 4, No. 1, (February 1987), pp. 12-22.

[KACH87] R. Katz, E. Chang, "Managing Change in a Computer-Aided-Design Database," In 13th Very Large Database Conference, September 1987, pp. 455-462.

[KING89] V. King, "Task Specification and Management in the VLSI Design Process," UCB/CSD 89/533, Computer Science Division, U.C. Berkeley, September 1989.

[LEBL84] D. Leblang, R. Chase, "Computer-Aided Software Engineering in A Distributed Workstation Environment," Proceedings ACM SIGPLAN/SIGSOFT Conference on Practical Software Development Environments, April 1984, pp. 104-112.

[LEE 88] A. Lee, "Use of History for User Support," Technical Report CSRI-212, Computer Systems Research Institute, University of Toronto.

[LEE 92] A. Lee, "Investigations Into History Tools for User Support", Ph.D. Thesis, CSRI-271, Computer Systems Research Institute, University of Toronto.

[LINX86] C. Linxi, A. Habermann, "A History Mechanism and UNDO/REDO/REUSE Support in ALOE," CMU-CS-86-148, Department of Computer Science, Carnegie Mellon University.

[MEHM87] Z. Mehmood, D. Singer, A. Singhal, K. Wu, "A Design Management System for CAD," 1989 International Computer Conference on Computer Aided Design, Santa Clara, November 1989, pp. 220-223.

[MITS80] T. Mitsuhashi, et al., "An Integrated Mask Artwork and Analysis System," In 17th Design Automation Conference, Minneapolis, June 1980, pp. 277-284.

[OCT 89] OCT Manual, U. C. Berkeley EECS Department Technical Report,

[OSTE87] L. Osterweil, "Software Processes are Software Too," Proceedings of 9th International Conference on Software Enginnering, Monterey, California, March 1987, pp. 2-13.

[OUST88] J. Ousterhout, A. Cherenson, F. Douglas, M. Neilson, and B. Welch, " The Sprite Network Operating System," IEEE Computer 21(2), pp. 23-26 (1988).

[OUST90] J. Ousterhout, "Tcl: An Embeddable Command Language," Proc. USENIX Winter Conference, January 1990, pp. 133-146.

[OUST91] J. Ousterhout, "An X11 Toolkit Based on the Tcl Language," Proc. USENIX Winter Conference, January 1991.

[PUKA90] C. Pu, G. Kaiser, N. Hutchinson, "Split-Transactions for Open-Ended Activities," Proceedings of 14th International Conference on Very Large Data Base, August 1988, pp. 26-37.

[RADI85] R. Radice, N. Roth, A. O'Hara, W. Ciarfella, "A Programming Process Architecture," IBM Systems Journal Vol. 24, No. 2, pp. 79-90.

[REPS83] T. Reps, T. Teitelbaum, A. Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," ACM Trans. on Programming Languages and Systems, 5(4), pp. 449-477, (July 1983)

[ROBE81] K. Roberts, T. Baker, D. Jerome, "A Vertically Organized Computer-Aided Design Database," Proceedings of 18th Design Automation Conference, Nashville, June 1981, pp. 595-602.

[SEQU83] C. Sequin, "Managing VLSI Complexity: An Outlook," Proceedings of the IEEE, Vol. 71, No. 1, January 1983, pp. 149-166.

[SILV92] M. Silva, T. Chiueh, R. Katz, "Active Documentation for VLSI Design," UCB/CSD 92/670, Computer Science Division, University of California, Berkeley, February 1992.

[TAKA89] T. Takala, "Design Transactions and Retrospective Planning: Tools for Conceptual Design," In *Intelligent CAD Systems II - Implementation Issues,* ed. V. Akman, P. ten Hagen, and P. Veerkamp, pp. 262-272, Spring-Verlag, Berlin, 1989.

[TAYL87] S. Taylor, N. Srinivas, J. Liu, L. Noronha, J. Kane, K. Wu, " Automating an IC Design Methodology," 1989 International Computer Conference on Computer Aided Design, Santa Clara, November 1989, pp. 224-227.

[TBOW87] R. Taylor, D. Baker, F. Belz, B. Boehm, L. Clarke, D. Fischer, L. Osterweil, R. Selby, J. Wileden, "Next Generation Software Environments: Principles, Problems, and Research Directions," CS-CU-370-87, University of Colorado, Boulder, July 1987.

[VANH89] E. vanHorn, R. Rezac, "Experience with The D-BUS Architecture for A Design Automation Framework," In 26th Design Automation Conference, Las Vegas, pp. 209-214, June 1989.

[VIDO89] N. Vidovic, D. Siewiorek, D. Vrsalovic, Z. Segall, "Towards A Consistent View of The Design Tools and Process in Distributed Problem Solving Environment," 1989 Hawaii International Conference on Systems Science, pp. 29-38, January 1989.

[WEIS86] S. Weiss, K. Rotzell, T. Rhyne, A. Goldfein,"DOSS: A Storage System for Design Data," Proceedings of 23th Design Automation Conference, July 1986, pp. 41-47.

[WIDY86] I. Widya, T. Leuken, F. Wolf, "Concurrency Control in A VLSI Design Database," 25th ACM/IEEE Design Automation Conference, June 1988, pp. 357-362.

[WONG79] S. Wong, W. Bristo, "A Computer-Aided Design Database," In 16th Design Automation Conference, San Diego, June 1979, pp. 398-402.

[YAMA87] Y. Yamaguchi, F. Kimura, and P. ten Hagen, "Interaction Management in CAD Systems with History Mechanism," Proc. of Eurographics '87, pp. 543-554, North-Holland, Amsterdam, 1987.

[ZINT81] G. Zinte, "A CODASYL Computer-Aided Design Database," In 18th Design Automation Conference, Nashville, June 1981, pp. 589-594.

Errata

There is no page 83 in this technical report.