

Copyright © 1984, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

AN ELECTRONIC CIRCUIT CAD FRAMEWORK

by

K. H. Keller

COVER

Memorandum No. UCB/ERL M84/54

6 July 1984

AN ELECTRONIC CIRCUIT CAD FRAMEWORK

by

K. H. Keller

Memorandum No. UCB/ERL M84/54

6 July 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

An Electronic Circuit CAD Framework

By

Kenneth Howard Keller

B.S. (Carnegie Mellon University) 1979

M.S. (University of California) 1981

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science


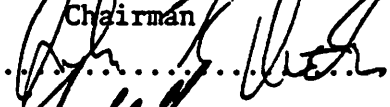
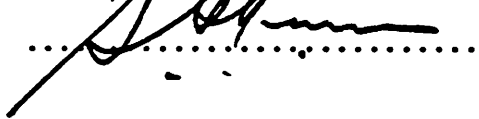
in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved:

	4/15/83
Chairman	Date
	4/16/83
	4/18/83

.....

AN ELECTRONIC CIRCUIT CAD FRAMEWORK

Kenneth Howard Keller

Ph.D.

Computer Science Division

Sponsors: Hewlett-Packard
Fannie and John Hertz Foundation

Signature 
A. Richard Newton
Committee Chairman

ABSTRACT

The requirements for a *framework for circuit computer-aided design* are presented, including the information and operations that circuit CAD tools require. An experimental framework has been developed that consists of the *data structure package* Squid, the device-independent graphics package MFB, and the user interface Hawk. Squid is object-oriented so it is efficient relative to general-purpose DBMS and design description language approaches. By calling Squid, circuit CAD tools are provided with storage for *hierarchical* connectivity, parameter, and layout information and provided with search operations. Squid uses a new data structure, OSL, for 2-dimensional searching so that searching layouts is efficient. OSL falls into the class of bin data structures, but is the first data structure to exploit the aspect ratio of shapes explicitly. Through Hawk, the user can have Hawk call tools and draw Squid information in color and multiple windows on the screen of a raster graphics device by calling MFB. Other contributions include tools for laying out arrays, routing schematic diagrams, and for the abstraction of layouts. The research on layout abstraction is particularly important and introduces the concept of *protection frames*. Protection frames speed drawing, extraction, layout rule checking, and process-independent compaction by reducing the number of shapes that each tool must process.

ACKNOWLEDGEMENTS

First and foremost I thank my research adviser, Prof. Richard Newton.

I thank Rick Spickelmier, Prof. John Ousterhout, and Prof. Stephen Penman for reading this thesis.

I thank the Berkeley students who contributed software to the framework— their names are listed in the appendices.

I thank Peter Moore for NLP algorithms and for building the mask operation package that made the protection frame computation program possible.

I thank Prof. Ousterhout's MAGIC group for exchange of technical information and for keeping their code readable.

I thank the first users of Hawk: the designers of the CMOS SOAR chip, Prof. Newton, and Res Saleh.

I thank my backers: Fannie and John Hertz Foundation, Hewlett-Packard, and the late Mr. I. Keller.

I thank my family who originated the sanity-preserving chant, "Love my thesis—feelin' good."

I thank my office-mates—Jeff Burns, Mark Hofmann, and Carl Sechen—for keeping the Flag flying high.

LIST OF FIGURES

1.1 A design hierarchy.	1
1.2 An ideal framework.	6
1.3 A real circuit CAD framework.	7
2.1 An instance hierarchy.	21
2.2 The dependency hierarchy of the resistor example.	22
2.3 Dependency propagation.	24
2.4 Four layers.	32
2.5 Outlined shapes with control points shown.	33
2.6 Examples of geometric functions.	40
2.7 A 1 kilohm polysilicon resistor.	42
2.8 Geometric operations.	47
4.1 An OSL data structure.	83
4.2 Pruning behavior of OSL data structure.	86
5.1 Screen controlled by Hawk.	91
6.1 NibbleReg is a homogeneous row vector of BitRegs.	120
6.2 Legend for nMOS process.	125
6.3 Layout view of a depletion-load inverter.	126
6.4 Bounding box view of a depletion-load inverter.	128
6.5 Doughnut view of a depletion-load inverter.	131
6.6 Pins of abstract view of a depletion-load inverter.	137
6.7 Frames of abstract view of a depletion-load inverter.	138

6.8 Layout view of address decode PLA.	139
6.9 Pins of abstract view of address decode PLA.	140
6.10 Frames of abstract view of address decode PLA.	141
6.11 Pins of array.	142
6.12 Frames and active area of array.	143
6.13 Pins of abstract view of array.	145
6.14 Frames of abstract view of array.	146

TABLE OF CONTENTS

Chapter 1. INTRODUCTION	1
1.1 The Problem	1
1.2 The Structure of Tools	3
1.3 Frameworks in General	5
1.4 An Experimental Framework	6
1.5 Design Methods	8
1.6 Terminology	8
1.7 Summary of Results	9
Chapter 2. REQUIREMENTS OF A CIRCUIT CAD FRAMEWORK	10
2.1 Mechanism and Policy	10
2.2 View Data Model	10
2.2.1 Naming	11
2.2.2 Protection and Locking	16
2.2.3 Revision Control	17
2.2.4 Crash Recovery	18
2.2.5 Other Operations	19
2.2.6 Stranger Views and Circuit Views	20
2.3 Circuit View Data Model	20
2.3.1 Instances	20

2.3.2	Dependencies	21
2.3.3	Parameters	26
2.3.4	Connectivity	27
2.3.5	Regularity in Connectivity	28
2.3.6	Geometric Information	31
2.3.7	Shapes	31
2.3.8	Geometric Functions	39
2.3.9	Placed Instances	43
2.3.10	Geometric Regularity	43
2.3.11	Geometric Operations	45
2.4	Database	49
2.5	Hardware Independence	51
2.6	User Interface	52
2.7	Summary	52
Chapter 3. THE SQUID PACKAGE		53
3.1	History	53
3.2	File System	54
3.3	Path Mechanism	55
3.4	Protection	56
3.5	Dependency Hierarchy	58
3.6	Revision Control	59
3.7	The Package from the Public Point of View	60
3.7.1	Terminology	60

3.7.2	Communication	61
3.7.3	SQ	62
3.7.4	Switching Contexts	65
3.7.5	The Resistor Cell of Chapter 2	67
3.7.6	Range Queries	70
3.7.7	A Demon	71
3.8	The Package from the Implementation Point of View	72
3.8.1	The Data Structure	72
3.8.2	Circuit View File Format	76
3.8.3	Transformation	79
3.9	Conclusions	80
Chapter 4. ORTHOGONAL SCAN-LINES		81
4.1	Background	81
4.2	Motivation	81
4.3	Principles of Operation	82
4.4	Algorithm Analysis	85
4.5	Measurements	87
Chapter 5. HAWK		89
5.1	Initialization	91
5.1.1	Layers	91
5.1.2	Commands	94
5.1.3	More Customization	97

5.1.4	Desktops	97
5.2	Scheduler	99
5.2.1	Keyboard Command	99
5.2.2	Pressed Button Number One	99
5.2.3	Pressed Any Other Button	100
5.3	Demons	101
5.4	Selection	102
5.5	User Interface	103
5.6	Measurements	104
5.7	Multiple Windows	105
5.8	Redisplay	106
5.9	Instance Transformation	109
5.10	Clipping	109
5.11	Window to Viewport Transformation	110
5.12	Ellipses	111
5.13	Bounding Boxes	112
5.14	Suppressing Detail	113
5.15	Obscured Pins	114
Chapter 6. NEW CIRCUIT CAD TOOLS		115
6.1	Logic Design Tools	115
6.1.1	Schematic and Block Diagram Capture by Beaver	115
6.2	Layout Design Tools	118
6.2.1	Drawing Layouts	119

6.2.2	Array	119
6.2.3	Protection Frames	123
Chapter 7. CONCLUSIONS		150
REFERENCES		152
Appendix A. SOURCE CODE		157
Appendix B. PACKAGE CONTRIBUTORS		158
Appendix C. HAWK TUTORIAL		160
10.1	Introduction	160
10.1.1	Assumptions	160
10.1.2	Hawk	160
10.1.3	Disclaimer	162
10.1.4	Getting Started	162
10.2	Basic Hawk	164
10.2.1	Screen Layout	164
10.2.2	Graphical Input	166
10.2.3	Invoking Commands	166
10.2.4	Manipulating a Single Window	171
10.2.5	Selecting Layers	172
10.2.6	Getting Help	172
10.2.7	Editing an Object	172
10.2.8	Pan & Zoom	173
10.2.9	Shape Drawing in General	173

10.2.10	Multiple Windows	174
10.2.11	More on Editing	174
10.2.12	Desktops	174
10.3	Tools	175
10.3.1	Selected Set	175
10.3.2	Hardcopy	176
10.3.3	Slide & Figure Making	177
10.3.4	Leaf Cell Layout	177
10.3.5	Floor Plan Layout	179
10.3.6	Schematic Capture	179
10.4	Wish List	179
10.5	Bug List	181
10.5.1	Fatal	181
10.5.2	Annoying	181
10.6	Installing a Command	181
10.6.1	Simple Command	182
10.6.2	Advanced Command	185
ACKNOWLEDGEMENTS		i
LIST OF FIGURES		ii
TABLE OF CONTENTS		iv

CHAPTER 1

INTRODUCTION

1.1. The Problem

Circuit design times are increasing dramatically [1-3], because circuit technology is changing and circuit complexity is increasing [4] while designer productivity is not keeping pace. A design is termed *hierarchical* if it can be modeled as a supervisory system composed of subordinate subsystems as in Figure 1.1.

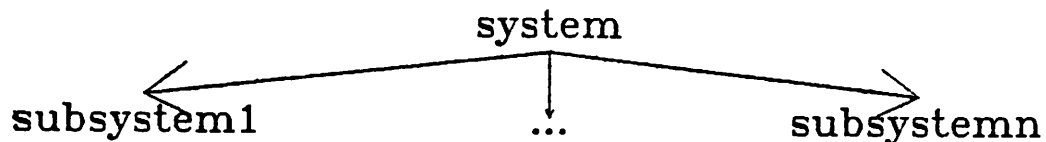


Figure 1.1: A design hierarchy.

Each subsystem can in turn be modeled as a system so that the design hierarchy is more than two levels deep. A microprocessor hierarchy could contain register file, ROM, data path, pad, and control subsystems. A design is termed *regular* if the frequency of re-use of the same subsystems is high or there is a pattern to the use of the subsystems. A ROM could be laid out as a 2-dimensional matrix of "1" and "0" subsystems in which there is a high frequency of each present. The solution to this problem is to use high-productivity design methods that exploit both design hierarchy and regularity [5] and to use Computer-Aided Design (CAD) software [6] which can exploit this structural information. CAD software is known also by the terms computer aids, CAD tools, design automation (DA) software, and

design aids. Henceforth, the favored term will be simply *tools*.

With a hierarchical design method, each subsystem is easier to understand, less error-prone, and more likely to be designed swiftly than the entire system. Because of the volume of design data involved, if one does not use a hierarchical design method, not enough computer power may be available and affordable to designers. Because subsystems are designed independently, the resultant system may not be as efficient— as measured by speed, power consumption, and cost—as the system designed as a whole. Thus, efficiency is traded for productivity. One objective of hierarchical design is to make minimum compromises in efficiency while maintaining minimum design time. Once a hierarchical design method is chosen, specific design methods must be selected for the subsystems and their combination into a single system.

There are many different design methods in use today [7], but regular design methods are gaining popularity. In a hierarchical design method, both regular and random design techniques can be applied in the design of the same system. In a regular design method, much functionality can be implemented quickly. For example, a ROM or PLA that implements a particular function can be designed by a synthesis tool in a matter of seconds while a random circuit for the same function may take days to design. Regular design methods also help to minimize the data volume problem mentioned earlier. For example, a ROM can be represented compactly. The layout of the ROM is similar in form to a 2-dimensional, Boolean matrix in which the value of an element selects one of two types of circuit. To conserve space, the layout can be represented as this sort of matrix. Designing subsystems by a synthesis tool, using subsystems taken from a library of pre-designed subsystems that are to be re-used, and designing with arrays that require little or no routing are all regular design methods.

It is *not* clear what the best design methods and technologies are for complex circuits. It is

likely that new ones will be discovered continuously. For that reason, it is important that a CAD system be able to adapt to new design methods as they evolve.

1.2. The Structure of Tools

There are many ways to build the tools that make possible the use of a computer-oriented design method including *design languages*, *database management systems* (DBMS), and *data structure packages*. Each has been successful in certain applications.

In the design language approach, there must be *at least one* design language that represents all the information that tools need. Each tool has a *parser* for each language it reads and a *generator* for each language it writes. The situation at Berkeley is a good example of this widely used approach. Various dialects of CIF [8] represent layout information. BLT [9] was developed to represent net list information for all tools that process net lists. Though the SPLICE [10] class of simulators uses BLT, the other simulators including SPICE [11], the net list comparator, and the extractor do *not* use BLT. Rather, they each have their own input language.

STIF [12] and SLL [13] were developed to represent net list and layout information, and the relationships between them, but they were never really used for design. However, the data models they represent are very similar to the data model presented in Chapter 2.

The disadvantages of multiple design languages are:

1. It takes time to learn them all.
2. The same information is replicated in different forms. This wastes space and leads to confusion and consistency problems.
3. Constructs that are not used by a tool must be parsed by it anyway even when the language can be separated into independent *sections*.
4. For ease of use, design languages are often human-readable, but this makes them bulky and may lead to inefficient parsers.

In the DBMS approach, a special-purpose database for the tools used by a specific design method is created by writing a schema for a general-purpose database such as relational, network, or hierarchical database [14-18]. Some of the advantages of the DBMS approach are:

1. Single, shared data structure that is easy to change.
2. Single, shared query language or set of operations.
3. No parsing.

But the DBMSs of today also have some problems:

1. They are slow relative to the other approaches [17].
2. Each DBMS supports *one logical* data structure that must be "twisted" to represent the information needed by tools.
3. Each DBMS supports *physical* data structures designed to make its standard operations on its one logical data structure fast. However, if the data structure of the design method does not fit the logical data structure, the physical data structures will not make the standard operations fast.
4. Standard operations are often not the right ones for the design method.
5. There is no revision control and it is hard to implement a revision control mechanism on top of a DBMS.
6. Features such as protection and crash recovery that are critical to business data processing cannot be disabled easily to recover the performance lost due to their presence.

In the data structure package approach [19-23], the best data structure for the tools can be built and used, because there are few constraints on the structure. If the data structure is created and maintained by invoking subroutines, procedures, or macros, then tools can be insulated from many changes to the data structure and tools can share common operations. Sometimes this approach is termed *language embedding*. When the data structure and operations are designed so that they serve *many* design methods, they become a *framework*—the topic of the next two sections.

1.3. Frameworks in General

The dictionary [24] defines a framework as *a structure for supporting something, or a basic system or design*. Since the class of regular, hierarchical design methods has been identified as most promising and new design methods within this class are being discovered continuously, it is reasonable to seek a framework for this class of design methods. The framework can be thought of as a tool box with common tools at the bottom and drawers on top to keep special, custom tools. From this point on, common tools will be termed *services* and special tools simply *tools*. If a circuit designer uses the same framework for two design methods, chances are he will learn the second one faster. The same is true for a CAD programmer.

To make it as easy as possible to implement a design method, all services that are bound to be common to design methods should be implemented and made easy to use by *tools and designers* alike. As illustrated in Figure 1.2, services include:

1. A data model that expresses not just the *objects* to be dealt with, but also the *operations* to be performed on them.
2. A database that implements the model. It is impossible to make an efficient database without taking into account the operations of the data model.
3. Inspecting and displaying the contents of the database.
4. Synthesizing and editing the contents of the database.
5. A user interface for scheduling services and tools.

It may also be important for the framework to be independent of particular hardware choices.

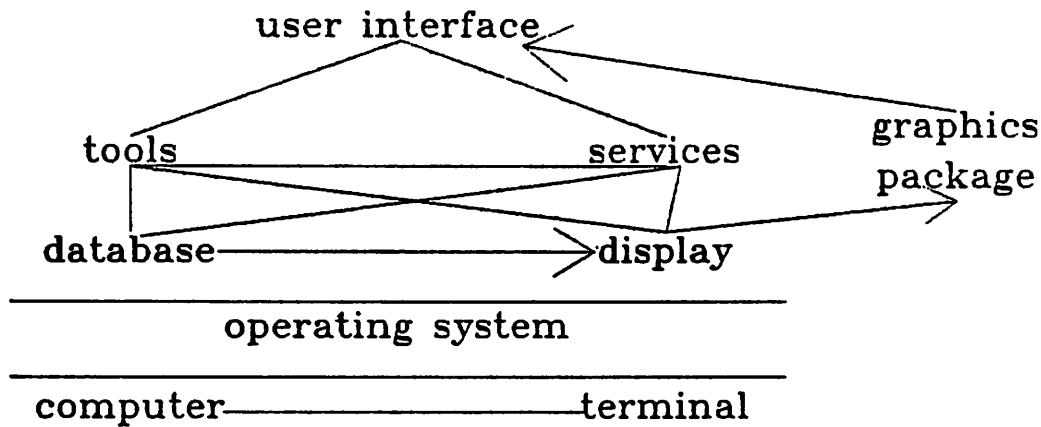


Figure 1.2: An ideal framework.

Thus far, the framework has been presented independent of an application. The requirements of a framework for circuit CAD in particular are described in Chapter 2.

1.4. An Experimental Framework

To support the propositions of Chapter 2, I wrote a framework to serve as a laboratory in which to execute experiments. At present, the framework is being used to design integrated circuits (ICs). Because of the diverse range of terminal hardware in many design environments and the recent developments in the work-station area, the framework must be as independent of hardware as possible. There are three main parts of the framework—the Squid data model and database package, the MFB device-independent graphics package [25], and the Hawk graphics editor. Squid and Hawk are the subjects of Chapters 3 and 5 respectively and their relationship is illustrated in Figure 1.3.

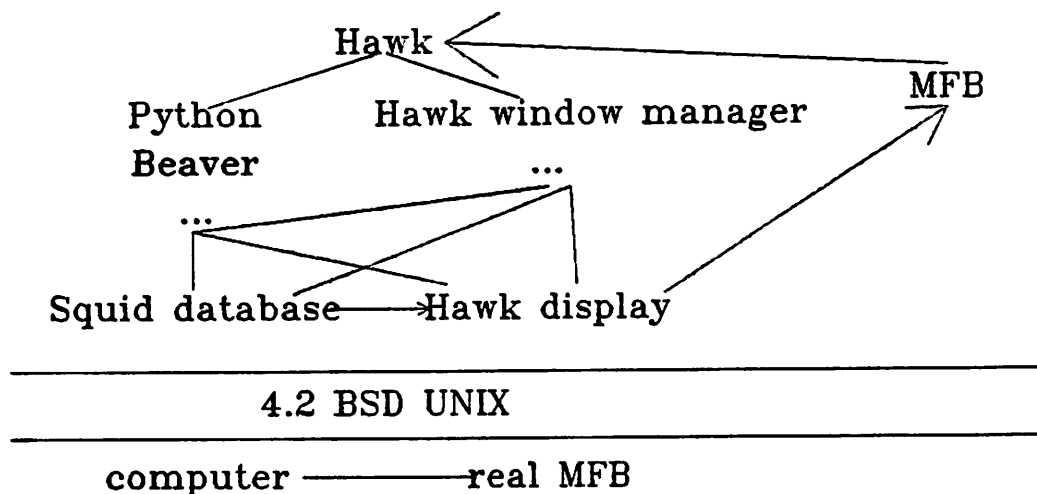


Figure 1.3: A real circuit CAD framework.

The framework is as independent of hardware as possible because it models the computer as the 4.2 Berkeley Software Distribution (BSD) dialect of the widely-used UNIX operating system and C programming language, and models the graphics terminal as a *model frame buffer* (MFB) that is mapped to a *real* frame buffer by the MFB package.

The Squid package has two parts presented in Chapter 3. The first part is a set of conventions for storing multiple views of a circuit in the UNIX hierarchical file system and the second part is a C package for locating the views and operating on the contents' of special views termed *circuit*. The Squid package defines an *object-oriented* [23] data model.

Hawk completes the framework by serving as the common user interface, scheduler, and display engine for all tools in the framework.

1.5. Design Methods

A framework cannot be judged accurately without applying it to a number of design methods and thus incorporating a variety of tools into the framework. To be fully successful, new tools must be easy to make, easy to install, easy to use, and efficient. The first three criteria are largely subjective while the last can be measured quantitatively in terms of computer resources required including storage and execution time.

In Chapter 6, a number of new tools are described that execute on top of the framework. The new tools include a schematic capture tool (Beaver), a procedural array making tool (Array), a procedural painting tool (Program), a compaction tool (Python), an alphanumeric terminal emulator and UNIX process window manager (WiSh), a parasitic extractor (Ibex), a Net Listing tool (NLP), and a GRaPH drawing tool (GRF).

1.6. Terminology

All measurements included in this thesis were made on a VAX-11/780 with a floating point accelerator executing 4.2 BSD UNIX.

The notation:

$$\langle e_1, e_2, \dots, e_n \rangle$$

is used to denote a finite sequence or n-tuple of length n whose elements are e_i .

The term *client* denotes either a person or a computational object that uses a service while the term *user* refers to a person that uses a service.

When a program is executed, UNIX passes it a *command line* made up of a sequence of *arguments*. By UNIX convention, any argument that begins with a minus sign but is not a number is termed a *switch* and is the name of a parameter of the program. By convention,

an argument after a switch that is not also a switch is the value of the parameter named by the switch.

Algorithms and code fragments are described using a variant of the "C" programming language [26].

1.7. Summary of Results

The requirements for a *framework for circuit computer-aided design* are presented, including the information and operations that circuit CAD tools require. An experimental framework has been developed that consists of the *data structure package* Squid, the device-independent graphics package MFB, and the user interface Hawk. Squid is object-oriented so it is efficient relative to general-purpose DBMS and design description language approaches. By calling Squid, circuit CAD tools are provided with storage for *hierarchical* connectivity, parameter, and layout information and provided with search operations. Squid uses a new data structure, OSL, for 2-dimensional searching so that searching layouts is efficient. OSL falls into the class of bin data structures, but is the first data structure to exploit the aspect ratio of shapes explicitly. Through Hawk, the user can have Hawk call tools and draw Squid information in color and multiple windows on the screen of a raster graphics device by calling MFB. Other contributions include tools for laying out arrays, routing schematic diagrams, and for the abstraction of layouts. The research on layout abstraction is particularly important and introduces the concept of *protection frames*. Protection frames speed drawing, extraction, layout rule checking, and process-independent compaction by reducing the number of shapes that each tool must process.

CHAPTER 2

REQUIREMENTS OF A CIRCUIT CAD FRAMEWORK

2.1. Mechanism and Policy

[27] popularized the distinction between mechanism and policy in the context of computer security, but the distinction is especially useful in the context of frameworks. The dictionary [24] defines a mechanism as *a process by which something is done or a machine*. The framework is the mechanism. Policy is defined as *a guiding procedure*. The design methods are policies for using the mechanism. A mechanism can constrain the policies that use that mechanism. In the remainder of this chapter, some of the requirements of an ideal efficient and flexible framework are presented.

2.2. View Data Model

In a hierarchical design approach, the a system is broken down into subsystems, then each subsystem is further reduced to subsystems, and so on. A system or subsystem is termed a *cell*. A *view* is a particular way of looking at a cell. To a test system, a complete representation of a circuit is its test vector set. That is the only information about the cell that the test system is concerned with. A cell can have a variety of views. Example view types are:

- Documentation.
- Performance graph.
- Test vector.
- Schematic diagram.
- Mask layout.

- Net list extracted from mask layout.
- Symbolic layout.
- Compacted symbolic layout.
- Expose instructions for an optical pattern generator.

In the following sections, each of the important characteristics of such a data model is described.

2.2.1. Naming

To name a view *of a cell*, the name of the view *and* the cell to which it belongs must be given. Thus, the name of a view of a cell is a pair:

< cell's full name, view's name >

In this section, various ways of producing such names are presented.

If only a single *name space* is used, then to name a cell, the cell's full and unique name must be given as in:

Full Cell Name
hopper'sALU'sbuffer
zap'sPC'sbuffer

Each name is cumbersome and composed of short names concatenated together.

If there are multiple name spaces or, equivalently, the one name space is partitioned, then the full name of a cell has the form:

< name space's name, cell's relative name >

Having more than one name space is more powerful than having only one provided each client has a current name space and he can use relative or full cell names. In this way, relative cell names can be used more than once. Different clients or the same client can use

the same relative cell name in different name spaces as in:

Name Space's Name	Relative Cell Name
hopper'sALU	buffer
zap'sPC	buffer
hopper'sBIE	—

If the current name space is **hopper'sALU** and the cell **buffer** is searched for, then **<hopper'sALU, buffer >** will be found, but if the current name space is **zap'sPC** and the cell **buffer** is searched for, then **<zap'sPC, buffer >** will be found. Designs must be merged with care. If two cells have the same relative cell name and they are merged into the same name space, either one will destroy the other or an ambiguity will result, unless one is renamed.

A list of names of name spaces is sometimes called a *search rule*, a *path*, or an *import list* [21]. Even more powerful than having a current name space is to have a current path. Given a relative cell name, its full name is computed by searching each element of the path until a cell is found whose name matches it. There is not a current name space. Paths make cell libraries possible. Consider:

Name Space's Name	Relative Cell Name
library	buffer
hopper'sALU	buffer
hopper'sBIE	—

If the current path is **(hopper'sALU library)**, then **buffer** is found in **hopper'sALU**. If the current path is **(hopper'sBIE library)**, then **buffer** is found in **library**, because **hopper'sBIE** does not contain any cells. The problem with this mechanism is that the current path must be assigned manually.

The current path and current name space schemes can be combined into a single mechanism. Associate a path with each name space. The current path is the path of the current name space. Let the current name space be the name space being searched. The special path ele-

ment "●" (dot) denotes the name space in which the current path is contained. The search algorithm follows. It returns **false** if search fails. Otherwise, it returns **true** and **nameSpace's** value is the full name of where **relativeCellName** was found.

```

BOOLEAN FUNCTION SearchP(
  currentNameSpace, relativeCellName, nameSpace)
STRING currentNameSpace, relativeCellName, nameSpace;
BEGIN
currentPath = PathOfNameSpace(currentNameSpace);

FOREACH element in currentPath,
  IF(element = ●)
    IF(InNameSpaceP(relativeCellName, element))
      nameSpace = element; /* Found directly. */
      return(true); ENDIF
    ELSE IF(SearchP(element, relativeCellName, nameSpace))
      return(true); ENDIF ENDIF /* Found indirectly. */
END

```

Consider:

Name Space's Name	Path	Relative Cell Name
library	(buffers ●)	
buffers	(●)	buffer
hopper'sALU	(● library)	buffer
hopper'sBIE	(● library)	—
hopper'sALU2	(library ●)	buffer

If the current name space is **hopper'sALU**, then **buffer** would be found in **hopper'sALU**.

If the current name space is **hopper'sBIE**, then **buffer** would be found in **buffers**, because **buffer** is not in **hopper'sBIE**, **buffers** which is searched because of **library's** path which is searched because of **hopper'sBIE's** path. If the current name space is **hopper'sALU2**, then **buffer** would be found in **buffers**, because **buffer** would be found from **library**, before **hopper'sALU2** could be searched.

In *lexical scoping* or *definition hierarchy*, name spaces can contain name spaces as in:

Root	Child	Grandchild	GreatGrandchild
hopper			
hopper	goldSoar		
hopper	goldSoar	buffer	
hopper	goldSoar	ALU	buffer
hopper	goldSoar	BIE	—

In this table, all occupied entries except those containing **buffer** are name spaces— **buffer** is a cell. The notation:

root/child/grandchild

denotes the name space **grandchild** contained in the name space **child** contained in the name space **root**. Paths contain just two elements. The first is the ● from above and the second is the name of the space that encloses it—sometimes called the *parent scope* and denoted "●●" (dot dot). If the current name space is **hopper/goldSoar/BIE**, then **buffer** is found in **hopper/goldSoar**. **buffer** is not in **hopper/goldSoar/BIE**, so the parent scope **hopper/goldSoar** was searched which does contain **buffer**. Presumably, this name space is serving as a library. If the current name space is **hopper/goldSoar/ALU**, then **buffer** is found in **hopper/goldSoar/ALU**.

Definition hierarchy is quite different from the hierarchy of view *uses* or *instances* described in Section 2.2.

If paths can contain other name spaces besides ● and ●●, then lexical scoping and paths can be combined. Consider:

Path	Root	Child	Grandchild	GreatGrandchild
(library)	hopper	goldSoar		
(● ●●)	hopper	goldSoar	ALU	
	hopper	goldSoar	ALU	buffer
(● ●●)	hopper	goldSoar	BIE	
(● buffers)	library			
(● ●●)	library	buffers		
	library	buffers	buffer	

If the current name space is **hopper/goldSoar/ALU**, then **buffer** is found in

hopper/goldSoar/ALU. If the current name space is **hopper/goldSoar/BIE**, then **buffer** is found in **library/buffers**, because the search fails in **hopper/goldSoar/BIE**. The search then considers the path (**library**) of the parent scope **hopper/goldSoar**. The name space **library's** path is (**● buffers**). The search of **library** itself fails so the path of **library/buffers** is considered. The search of **library/buffers** itself succeeds.

So far, a cell cannot have more than one view of the same type, because the view component of a view name has been regarded as the name of the view's type. However, for a cell sometimes it is useful to have multiple views of the same type. Thus, it is reasonable for a data model to extend the name of a view of a cell to be a triple:

< cell's name, view type's name, view's name >

Multiple views are equivalent to multiple representations, but this leads to the notion of multiple *alternatives*— a notion with artificial intelligence overtones that is not presented in depth here. Each cell may have multiple alternatives each of which has multiple views. Suffice it to say that the names a client chooses for his name spaces, cells, and views may encode considerable meaning— a cell may have low and high power alternatives, a radiation-hard low power alternative, and so on. A design method may decree that name-space names are drawn from a hierarchical taxonomy of knowledge about circuits. For example, a name space name may have the form:

designer/chip/macro-cell/cell

as in:

hopper/goldSoar/ALU/buffer

Such names map onto a hierarchical name space nicely, but such a name space is not necessary. A data model may have a name space name be a set of pairs:

< property's name, property's value >

with a powerful pattern matcher so that name-spaces can be named easily. This defines a hypercube in which each dimension corresponds to a property. Hierarchical names map onto a hypercube well, because, in a pattern, the order of the components does not matter. For the properties designer, chip, macro-cell, and cell, the name:

chip = *, designer = *, cell = buffer

would match:

chip = goldSoar, designer = hopper, macro-cell = ALU, cell = buffer
 chip = library, designer = librarian, macro-cell = buffers, cell = buffer

Henceforth, when the term *view of a cell* is used, the reader may think of it either as a triple or a pair. Sometimes, the term *view* will mean view of cell and at other times, it will mean a type of view. If the sense is not clear from context, the term *view of cell* or *view type* is used.

2.2.2. Protection and Locking

A mechanism that controls who can access an object and how it can be accessed by him is termed an *access control, protection, or security mechanism*. A mechanism that also controls when an object can be accessed is termed a *locking mechanism*. The grain of locking and protection can be the cell, the view of a cell, or even finer grain. A data model should protect and lock at least at the level of the view, because unless a multiple process computation is updating a view, it only makes sense for one process to update a view at a time.

A *demon* [20, 23] is a procedure that is associated with a type of event and is called to act when an instance of that type of event occurs. The demon may change the outcome of the event or it may perform an independent task. The range of possible protection strategies is bounded by *unrestricted access* at one extreme and by the use of demons at the other extreme. *Access control lists* belong in the middle of the spectrum. If there is no chance

that information will be used by more than one client, unrestricted access is the most efficient approach. If this is not the case, access control lists that can control who can write an object are critical. If a demon is associated with each object in a database, an arbitrary protection policy can be implemented. For example, a demon could permit an object to be edited for no more than a certain period of time.

The range of possible data locking is bounded at one extreme by no locking at all and at the other extreme by one-client-at-a-time-per-object locking. Multiple readers but only one writer allowed at a time per object is in the middle of the spectrum in this case. A data model should permit multiple readers but only one writer at any time if more than one client is responsible for, or can modify, an object. If libraries are always copied from development computers to production computers and each client is solely responsible for his own objects, locking is unnecessary except at the time of copying.

2.2.3. Revision Control

Each object can change with time—each has a history. If a view of a cell is extended to be a triple:

< cell's full name, view's name, timeRange >

or four-tuple:

< cell's full name, view's type's name, view's name, timeRange >

the *latest* or *current* view in a view's history is well-defined, and there are operations on the history, then history is captured at the granularity of the view. For each view, each value of timeRange names a *revision* or *version*. To implement this mechanism, the latest n versions of a view can all be stored [28], or a differential representation can be stored and the versions derived from it [29]. In the former case, more space is traded for less time. In the latter case, more time is traded for less space in theory. In practice, it may be that more

time is traded for more space, because two versions may be very similar semantically, but differ greatly in form. To date, research on differential representations has focused on form rather than meaning [30].

2.2.4. Crash Recovery

Most interactive editors store the current version of a design in both secondary storage *and* in primary storage. The secondary storage representation is as consistent as this store itself, but there is no guarantee that the two representations are consistent except at the time the cell is written back to secondary storage— *save* time. The process of saving makes the secondary storage representation conform to the primary representation. The process of re-editing does the inverse—it makes the primary storage representation conform to the secondary one.

If the computer or the framework crashes before the client has invoked the save command, the client will lose the changes made to the primary storage. To be safe, he invokes the save command periodically. The *crash recovery* capability makes it unlikely that changes will ever be lost, because crash recovery insures that if a command completes and then the computer crashes, the current version will contain what the client expects it to contain.

The problem with crash recovery is that it is useful to clients that the secondary and primary storage representations of the current version can differ. Before a client makes a substantial change that does not warrant creation of a new version, he invokes the save command. He then makes the change and if he does not approve of it in retrospect, he invokes the re-edit command. Thus, the re-edit command can be regarded as a form of *undo* command. Of course, the client must still remember to invoke the save command at the right time.

If changes to the current version are logged by the database and each change can be inverted or undone, then a new scenario is possible. Any command can be undone or re-done. Thus,

a client can explore many changes without having to worry about invoking the save command. Undoing all of the commands invoked since the last invocation of the save command is equivalent to invoking the re-edit command. If the change log is kept in the secondary store, then crash recovery can be performed by re-doing all of the changes to the current version which will still be consistent after the crash. Because the log can become very bulky, the DBMS may truncate it at save time. This will trade space for being able to undo beyond a save.

The change log capability effectively captures history at a finer granularity than the version. This can also be implemented by insisting that version-creation be a *function* of the operation on the objects belonging to a view. This is unacceptable because it leads to many artificial versions that do not correspond to substantial changes.

2.2.5. Other Operations

A data model should have a full set of operations for manipulating cells and their views. These operations include copying, renaming, creating, and deleting cells and their views. If revisions and multiple views of the same type are to be supported, then the operations are the same, but their operands are bulkier.

Sometimes a client must be able to associate client-specific information with an object. Without this capability, whenever a client had to associate information with an object that is not part of the object the client would have to invent a new data structure package! A *property list* is defined as a set of pairs:

< property's name, property's value >

Each cell, view of it, and all of the objects in the views should have an arbitrarily-long property list associated with it for storing client-specific information. At least integer, real, Boolean, string, and time constants should be valid data types for property values.

2.2.6. Stranger Views and Circuit Views

A data model for the contents of views termed *circuit* views is described in the remainder of this section. Views whose contents are unconstrained by this data model are termed *stranger* views. A stranger view's contents are treated as a sequence of bytes termed a *stream*. Standard stream operations—read, write, rewind, and seek—should be provided for stranger views. When the information required by a tool just cannot be represented in a circuit view even with the aid of property lists, the tool must resort to stranger views. For example, special documentation, source code for a functional model of a cell, and tester output may be stored as stranger views. All of the facilities presented thus far are applicable to both stranger views and to circuit views.

2.3. Circuit View Data Model

The information carried by the objects in a circuit view is:

- *Instances* of child views of cells (which make possible hierarchy).
- *Property lists* associated with each object to make it possible to extend the data model in a limited way.
- *Net lists* which represent physical and logical connectivity.
- *Layers* which represent masks and inks, and which are populated by shapes.
- The *geometric functions* of *shapes* which relate net list and geometric information.

This information is presented in detail in the following sections. A resistor cell named *resistor* will be used as an example to illustrate each concept.

2.3.1. Instances

An *instance* is the *use* of one view inside another view. The view that is used is termed the *master* [31], because it is similar to a master copy. The instance depends on its master

and, by transitivity, the view that contains the instance depends on the master. The view that contains the instance is termed the *dependent* view or simply the dependent.

The dependent is the source of an un-weighted, single-source, directed, acyclic graph (DAG) whose arcs are the instances in the dependent and whose nodes are the masters of the instances. The arcs are directed from the source to the nodes. Because masters can also be dependents, the longest path's length may be greater than one. Such graphs are often termed *instance hierarchies* even though they are not trees, because a master node may have in-degree greater than one and the arcs are directed. An example of an instance hierarchy is illustrated in Figure 2.1.

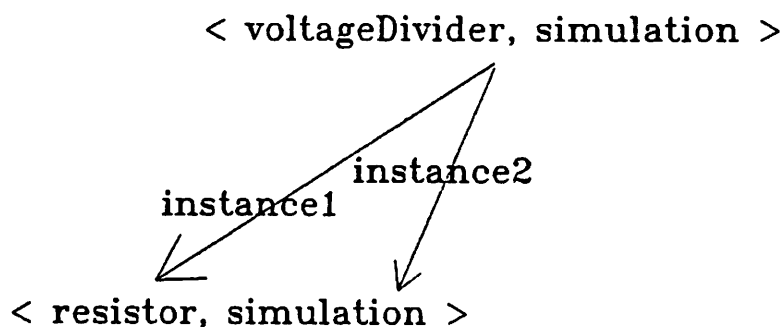


Figure 2.1: An instance hierarchy.

Instances are analogous to calls, masters to subroutines, and instance hierarchy to call graph in programming languages.

2.3.2. Dependencies

Sometimes a client must determine the dependents of a master in order to process the dependents, because of changes to the master. State-of-the-art programming environments provide

tools such as MASTERSCOPE [32] that enable the programmer to display and edit calls of subroutines. The use of such tools is termed the *deferred* approach, because when a subroutine is changed, the calls to the subroutine are not immediately checked for validity. Some tools, notably Hawk's display service, *automatically reflect* or *propagate* changes of masters to their dependents. This is referred to as the *automatic* approach. Each particular design method and its supporting tools determine which approach is most suitable.

A second graph can be constructed that provides the information that makes a dependency-analysis tool possible. To compute this graph, let the master, from which changes are to be propagated, be the source and reverse the direction of the arcs in the instance hierarchy graph. Sometimes, these reversed arcs are termed *pointers up the instance hierarchy*. The graph is termed the *dependency hierarchy* and a data model should have it by associating a *dependency set* of:

< instance, dependent >

pairs associated with each master. The dependency hierarchy of the resistor example is illustrated in Figure 2.2.

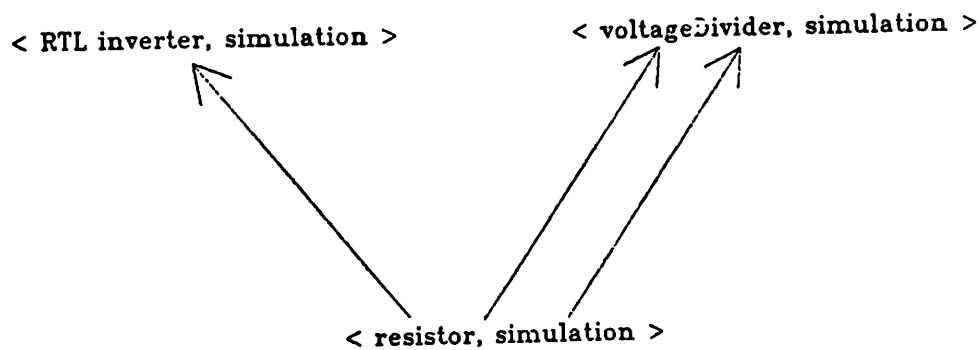


Figure 2.2: The dependency hierarchy of the resistor example.

To make possible an efficient implementation of the deferred and automatic approaches, each

view should have a set of times— termed *time stamps*— associated with it. The view's property list is a convenient place to put time stamps.

Assume that all tools can be interrupted as they execute. Each view has a time stamp whose value is when the view was last updated. For each view and each analysis tool, the time of the tool's last *complete* analysis of the view should be a member of the time set. Thus, an analysis tool can be applied to a view and it can determine what it must analyze by comparing the times of its last analysis to the times of the last updates *of the view itself and all masters that can be reached from it in the instance hierarchy graph!*

It is insufficient to examine only the masters of instances of the view as shown in the following example illustrated in Figure 2.3.

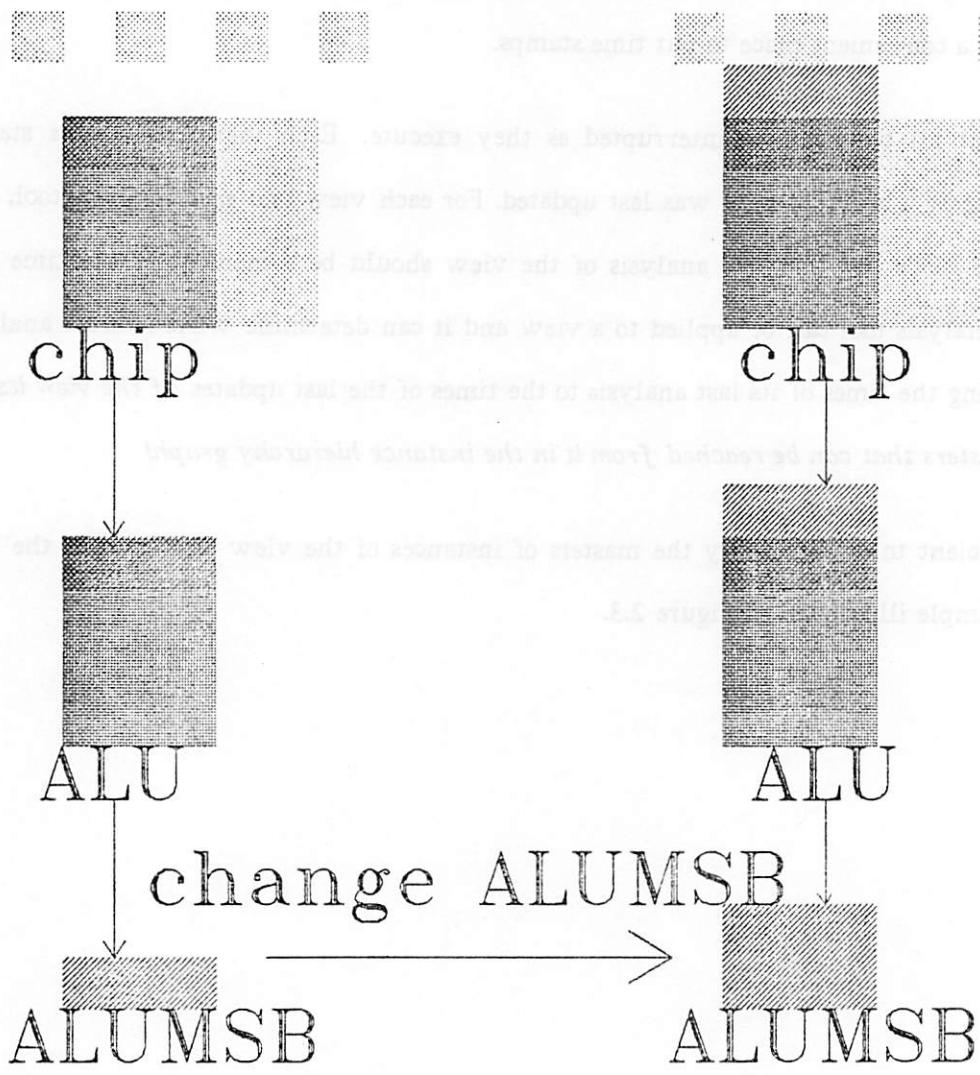


Figure 2.3: Dependency propagation.

It is useful to associate a bounding box with each view—the box approximates the area that the view occupies. Suppose **chip** contains an instance of **ALU** which contains an instance of **ALUMSB**. The time stamp on **chip** is later than the time stamp on **ALU**, because **ALU** was stable when the instance of it was created in **chip**. A client increases **ALUMSB**'s layout area so that the area of **ALU** increases too. Now, the client examines **chip** with his graphics

editor. The graphics editor will show a bounding box that is too small for ALU unless it examines the instances in ALU.

For efficiency, it should be possible to access an entire instance hierarchy and its time stamps *without* accessing the other objects—such as shapes—in the masters. It takes care to implement a database that separates instances, time stamps, and other objects. It is insufficient to have the time of the last update of each view only *unless* a view cannot be updated if any of its masters have been updated more recently than it has been updated, and it has not been completely analyzed.

The amount of time information in the data model can be traded for efficiency of the implementation. The information carried by the time information can also be represented differently. Consider a mixed approach. When a master is changed, a Boolean property named *re-analyzeP* is automatically asserted on the property list of the instance components of the master's dependency set, but the analysis itself is deferred. When the analysis tool is applied to one of the dependents, it searches for all instances whose property list contains a true-valued property named *re-analyzeP* and analyzes such instances.

If the view or master is renamed, there are obvious problems. Creating an instance of a *version* of a view rather than a view itself and then deleting a view if and only if the view's dependency set is empty would prevent these problems from arising.

Often a client is not quite using a *view*—rather he is using a *cell* and the view of that cell he chooses to create an instance of is just a *preferred* view for purposes of display. For example, an instance of a schematic-symbol view is created in a schematic-diagram view, but at net listing time, the symbol view is replaced by its underlying diagram view. For example, an instance of a bounding box view is created in a layout view, but the bounding box view is replaced by its underlying layout view when the user zooms in enough or when layout analysis tools are applied to it. Thus, a data model may have cells be the

masters instead of the views of cells. In addition to the master cell's name, a set of equivalent views and a preferred one could be associated with the instance.

An instance has an optional name. If an instance in a view is being selected by via a graphics editor, then the instance can be found without reference to its name by pointing to it. Thus, the instance's name is irrelevant except for documentation and as a means for selecting the desired instance among many which overlap. Some tools must report results and diagnostics non-interactively. These tools must have a way of referencing an instance uniquely. Thus, an instance that is not given a name by the user should be given a unique name by the graphics editor. A demon performs this service for Hawk. In contrast, if an instance in a view is being referenced in a design description language, it must have a unique, user-defined name.

2.3.3. Parameters

Parameters are analogous to programming language parameters. A parameter is similar to a variable in that it has a name and a scalar value such as a *string*, *integer*, *Boolean*, or *real*. A simulation view of a resistor named `<resistor, sim >` would have a parameter named `R` to represent its resistance. Because `R` is a variable, this view represents a *class* of views. If `R`'s default value is 1000, then its default resistance is 1000 ohms assuming the unit of resistance is ohms in all tools. When an instance of this view is created, all instances will have a resistance equal to the default value unless an *overriding* value is associated with the instance by naming `R` and making its value the overriding one. An overriding parameter is termed an *actual parameter* and the default parameter on the master is termed a *formal parameter*. These terms come from the programming language analogy.

2.3.4. Connectivity

A *terminal* is a circuit object that may have a voltage-valued variable associated with it, among others. Here, voltage is abstract such as **high** or 0.3 millivolts. Like a parameter, a terminal has a name. The simulation and layout views of a resistor have two terminals named + and -. These two terminals are functionally interchangeable in this case and a data model should represent this fact. In a hypothetical design description language, the simulation view might have the form:

```
CELL resistor
VIEW TYPE simulation

TERMINAL +
TERMINAL -
CAN INTERCHANGE + -

FORMAL PARAMETER R 1000
```

When an instance of one of these views is created, the instance inherits the terminals of its master. As in the case of parameters, terminals of the master are termed *formal* terminals and those of the instance are termed *actual* terminals.

A *net* represents a "signal" or a set of terminals that are "logically" connected such that the terminals all have the same voltage. The physical implementation of the connection determines how different the terminal voltages are. A net has an optional name. To make the simulation view of a voltage divider cell, two instances of the simulation view of the resistor would be made in it:

```
CELL voltageDivider
VIEW TYPE sim

COMMENT instanceName masterCellName masterViewType
INSTANCE instance1 resistor sim
INSTANCE instance2 resistor sim
```

A net would connect either of the two actual terminals on one of the instances with either of the two actual terminals on the other instance.

```

CELL voltageDivider
VIEW TYPE sim

INSTANCE instance1 resistor sim
INSTANCE instance2 resistor sim

COMMENT <instanceName, terminalName >
NET net1 <instance1, - > <instance2, + >

```

This would leave two unconnected actual terminals—one on each instance. Three formal terminals could then be created named **ground**, **voltage**, and **dividedVoltage**. **ground** would be connected to either of the two unconnected actual terminals by making a second net and **voltage** would be connected to the other by making a third net. **dividedVoltage** would be merged in the first net.

```

CELL voltageDivider
VIEW TYPE sim

INSTANCE instance1 resistor sim
INSTANCE instance2 resistor sim

TERMINAL ground
TERMINAL voltage
TERMINAL dividedVoltage

NET net1 dividedVoltage <instance1, - > <instance2, + >
NET net2 ground <instance2, - >
NET net3 voltage <instance1, + >

```

2.3.5. Regularity in Connectivity

A bus is a sequence of nets or terminals termed bits that it is convenient to treat as a unit. For example, an operand bus of an ALU may be implemented as 32 separate bit terminals, but it is inconvenient to specify a voltage for each one. Rather, it is convenient to apply a decimal value to the operand bus that may be used directly by a functional simulation tool or partitioned into bit voltages by a circuit simulation tool.

Some data models allow sequences of nets, but not sequences of terminals. However, a sequence of nets in a view can be thought of as a sequence of terminals that *might* be

formal terminals of the view. Not having sequences of terminals make instances bulkier than they need to be, because they force each bit to be represented as a separate terminal.

If terminal sequences can be represented by single terminals, then the problem is solved. The schematic-symbol view of an ALU would have a terminal named `operandA<0-31 >` and the same view of a shifter would have a terminal named `in<0-31 >`. A net connecting these two terminals represents 32 bits by the naming convention as in:

```
INSTANCE alu ALU schematic-symbol
INSTANCE sh shifter schematic-symbol

NET <sh, in<0-31 >> <alu, operandA<0-31 >>
```

If bit 7 has to be extracted from a bus terminal, then the terminal is connected to a *tap*, *merger*, or *ripper* instance that has at least two terminals named `32<0-31 >` and `i`, and a parameter named `i` whose value is the number of the bit to be tapped:

```
INSTANCE alu ALU schematic-symbol
INSTANCE sh shifter schematic-symbol
INSTANCE tap 1From32 schematic-symbol
COMMENT Tap off bit 7.
ACTUAL PARAMETER tap i = 7

NET <sh, in<0-31 >> <alu, operandA<0-31 >
NET <alu, operandA<0-31 >> <tap, 32<0-31 >>
COMMENT Connect operandA<7 > to ...
NET <tap, i > ...
```

In a real design description language, the tap instance could be implied by notation that is compiled into the database:

```
INSTANCE alu ALU schematic-symbol
INSTANCE sh shifter schematic-symbol
NET <sh, in<0-31 >> <alu, operandA<0-31 >
NET <alu, operandA<7 >> ...
```

The schematic-drawing view of the ALU would have 32 separate terminals named:

```
CELL ALU
VIEW TYPE schematic-drawing

TERMINAL operandA<0 >
...
```

TERMINAL operandA < 31 >

The mapping between the operandA terminals on the drawing view to the terminal on the symbol view is straightforward.

Rather than having to parse terminal names to access the bits of a sequence of terminals representing a bus, a data model may have the name of a terminal be a triple:

<string name, number of LSB, number of MSB >

If there are no tap instances in a schematic-drawing view, then there is no need to number bus terminal names at all, because the number of bits in the bus can be determined at net listing time by examining the schematic-drawing views associated with instances of schematic-symbol views.

Often in logic families, one has a part that processes nBits bits and wishes one had a part that processes

multiple*nBits

bits. To represent the desired part, the client—a user or tool—creates a new cell with schematic-diagram and schematic-symbol views. The client instances in the new diagram view *multiple* instances of the nBits symbol view, connects them together if desired, copies each formal terminal on the nBits symbol view to the new diagram view, adjusts the bit range on any bus formal terminals, and connects the formal terminals to the actual terminals. Bus formal terminals must be connected to actual terminals through tap instances. If subscripts are not being used on bus formal terminals, then the new and old symbol views are the same. If not, then the client takes the nBits symbol view, copies it to the new symbol view and adjusts the bit range on any bus formal terminals.

As a second alternative for representing the part, the client just instances the original symbol view, but makes the value of the instance's property named SIZE the value of **multiple**

[33]. The new diagram view is *implied* by the original symbol view and the value of the **SIZE** parameter, but it is not created. It is up to the net list processing tools to implement this implication.

There are additional properties that net list processing tools may wish to recognize. The **ConnectSingletonsP** property directs the tools to assume that all terminals that are not buses are to be connected. This arises when control signals are represented by terminals that are not buses.

The **Cascade** property on a terminal **in** directs the tools to assume that **in** on an instance connects to **out** on the next—in the sense of next bit—instance. This arises when one wishes to connect bits of a shift register or **carryOut** to **carryIn** on an adder. Other properties that assert regularity are undoubtedly desirable.

2.3.6. Geometric Information

There are a variety of techniques for describing geometric data [34, 35]. The issues of prime importance here are to simplify the description of the most common shapes while not excluding more complex ones. Hints, such as "this shape is a rectangle", can often be used to improve processing efficiency later. In the following sections, a particular choice of shape types is described which illustrates the main issues involved.

2.3.7. Shapes

Each separate view possesses a set of 2D, Cartesian coordinate systems that are superimposed. Each coordinate system is termed a *layer* as illustrated in Figure 2.4.

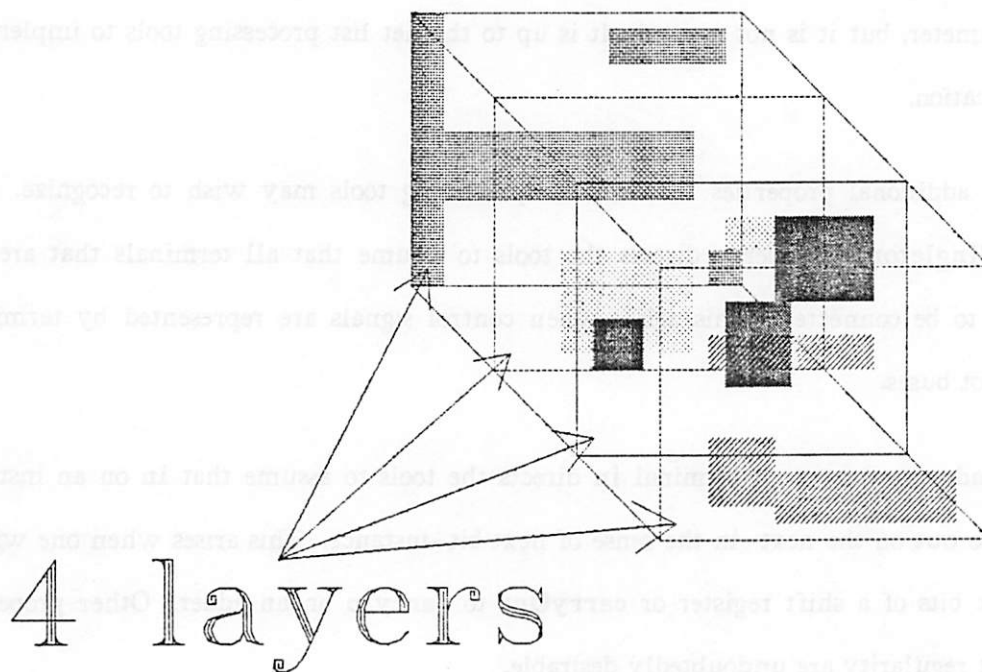


Figure 2.4: Four layers.

The meaning of layers is application-dependent. A view of a cell that contains integrated circuit masks would possess shapes on layers that represent masks and a schematic view of a cell would have shapes on layers that represent inks. Different inks could be used for pins and wires, symbols, and labels.

A *shape* is a set of curves on one layer. The curves forming a single shape need not be closed. If a shape is *filled*, it becomes a surface bounded by the shape's curves. If a shape is *outlined*, it remains a set of curves. Examples of outlined shapes are illustrated in Figure 2.5.

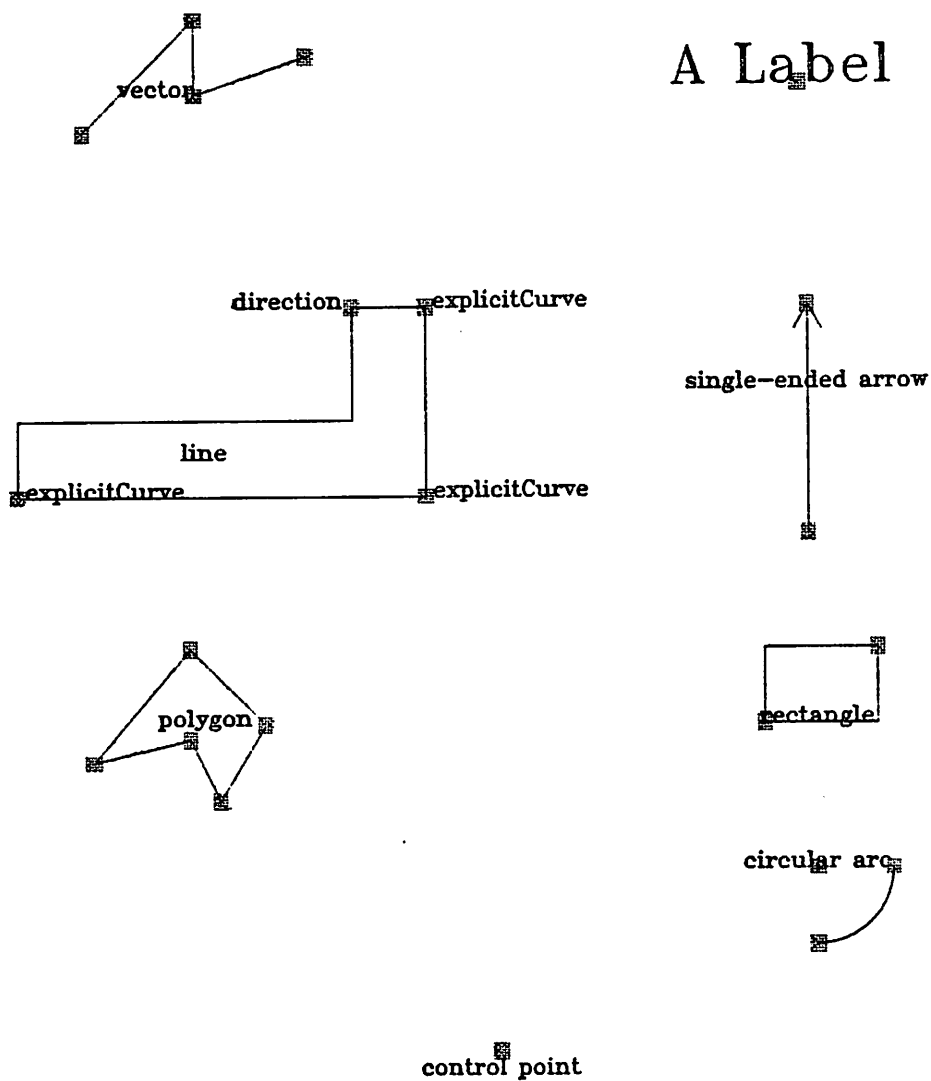


Figure 2.5: Outlined shapes with control points shown.

A shape is termed *Manhattan* or *orthogonal* if each of its curves are perpendicular to one of the two axes. A shape can be defined by a set of *control points* and parameters. A *spline* is a curve that is defined by a sequence of control points that indicate the curve's desired shape. The term control points is used in the literature of *splines* [31].

Magnetic tapes for machines that make masks and wire-wrap boards are expressed in real

coordinates. For a particular machine, real coordinates can be re-expressed as integral multiples of a real-valued unit of distance termed *lambda* [8] that is equal to the smallest distance the machine can resolve. Multiples of *lambda* can be processed more swiftly than real coordinates can be processed. Expressing real coordinates as multiples of *lambda* also defers the selection of *lambda* to the last possible moment so the designer can re-scale his design if he wishes. For these reasons, control points must be integral and their units are *lambda*. A silicon wafer six inches in diameter is 15.24×10^8 angstroms or a little more than 10^9 angstroms across. Since 10^9 can be encoded in 32 bits, an acceptable precision for an integer is 32 bits. Fortunately, current microprocessors process 32-bit data.

Unless a shape is a point, in the general case all of the points on the shape cannot be represented by integral coordinates. Tools must deal with this fact and it may cause problems. For example, if display code clips a non-Manhattan polygon in window coordinates, then visible edges may be displayed with the wrong slopes. The solution is to clip in viewport coordinates. For example, an operation that clips a non-Manhattan polygon to a rectangle may yield polygons whose control points cannot be represented by integral coordinates. The solution is to let the user beware or let him draw control points on an integral *drawing grid* that is coarser than the integral grid. For example, if each circular arc is defined by three control points on the arc, then an arc with a slow rate of curvature can only be drawn on a quite coarse drawing grid.

A *rectangle* is defined by a lower left corner and an upper right corner. In the hypothetical design description language, a rectangle might have the form:

```
SHAPE
  LAYER NP
  LOWER LEFT 10 10
  UPPER RIGHT 100 100
  COMMENT All rectangles are Manhattan.
```

Filled rectangles occur very frequently in mask layout while unfilled rectangles often

occur in block diagrams and schematic diagrams. Every shape defines an implicit rectangle that is termed its *bounding box*. A shape's bounding box is the rectangle with minimum area that encloses all of the points on the shape's curves.

Refer to Figure 2.5. A filled *line*, or *path*, is defined by a sequence of control points and a width, *width*. Each successive pair in the sequence defines a line segment. The curve defined by this sequence of line segments is termed the center-line:

```
SHAPE
  LAYER connect
  CENTER LINE 0 0 100 0 100 100
  WIDTH 20
```

```
COMMENT The bounding box for this line is implied by the center-line.
LOWER LEFT 0 -10
UPPER RIGHT 100 110
```

```
COMMENT By inference, this line is Manhattan.
IS MANHATTAN
```

The line itself can be thought of as a thick center-line with *width* governing the thickness. More formally, the line is the locus of points that are $\text{width}/2$ from the line segment sequence with the semicircles at each end cut away so that the first and last control points correspond to where the line begins and ends respectively. The problem with this representation is that often a line of odd width must end on the edge of a rectangle, but then the line's control points do not lie on the integral grid.

The center-line of a line divides the closed curve that bounds the line into two open curves. In a representation that solves the aforementioned problem, the control points define one of these two curves. The curve defined by the control points is denoted by *explicitCurve* and the other by *implicitCurve*. *implicitCurve* is defined by an additional control point *direction*. The direction vector whose tail is the first control point of *explicitCurve* and whose head is *direction* defines the direction to travel in to intersect *implicitCurve*. The direction vector must be perpendicular to *explicitCurve*. *width* is the distance to travel.

```

SHAPE
LAYER connect
EXPLICIT CURVE 0 -10 110 -10 110 100
DIRECTION 0 0
WIDTH 20

```

```

COMMENT The bounding box for this line is implied by the center-line.
LOWER LEFT 0 -10
UPPER RIGHT 110 100

```

```

COMMENT By inference, this line is Manhattan.
IS MANHATTAN

```

Filled lines with non-zero-width on mask layers could serve as physical interconnect in an IC. Filled lines with non-zero-width on PCB-side layers could serve as wire traces. Lines with zero-width--termed *vectors*-- on ink layers could serve as wires in a schematic diagram. Note that there is no logical, or connectivity information, expressed explicitly by the line. It is reasonable to have a *vector* shape so that a shape processing tool does not have to check width. It is useful to have two special lines-- *double-ended arrow* and *single-ended arrow*-- that represent arrows, because they are used often in schematic symbols-- for example diodes--and schematic diagrams--for example buses. Arrows can be made up of lines and polygons, but then selecting an arrow with a graphics editor becomes difficult, because the arrow is really a set of separate shapes that each must be selected.

A *polygon* is defined by a sequence of control points termed its *vertices*:

```

SHAPE
LAYER CD
VERTICES 0 0 100 0 50 100

```

```

LOWER LEFT 0 0
UPPER RIGHT 100 100

```

```

COMMENT A triangle cannot be Manhattan.

```

A *circle* or doughnut slice is represented by a point defining its center, and integer constants defining its inner and outer radii, and beginning and ending angles:

```

SHAPE

```

```
LAYER CD
INNER RADIUS 0
OUTER RADIUS 100
BEGINNING ANGLE 0
ENDING ANGLE 90
CENTER 0 0
```

```
LOWER LEFT 0 0
UPPER RIGHT 100 100
```

COMMENT Such a shape cannot be Manhattan unless it has trivial radii.

The angles are in degrees and are measured counterclockwise from the x-axis. To be exact, the data type of an angle must be real. There are four implicit control points corresponding to the beginnings and ends of the inner and outer circular arcs. These control points do not necessarily lie on the integral grid. Technically, a circle whose diameter is odd cannot be represented. Practically, if typically control points are placed on a drawing grid, but the user can also place control points between drawing grid points, then a circle whose diameter is odd can indeed be represented.

A *Manhattan* transformation may be any sequence of flips about the axes, rotations about the origin by multiples of 90 degrees, and translations. The above, angle-oriented representation of a circle can be transformed by a Manhattan transform very quickly by translating the center point and calculating the transformed angles trivially.

A representation that does not use angles or radii may be better:

```
SHAPE
LAYER CD
OUTER ARC'S BEGINNING 100 0
OUTER ARC'S ENDING 0 100
INNER ARC'S BEGINNING 0 0
INNER ARC'S ENDING 0 0
CENTER 0 0
```

```
LOWER LEFT 0 0
UPPER RIGHT 100 100
```

COMMENT Such a shape cannot be Manhattan unless it has trivial radii.

The problem with this representation is that a center point and two other control points

define a circular arc only if the distance from the center point to each of the other two control points is identical. It is also possible to eliminate the center point by representing a circular arc by three control points that lie on the circular arc. A problem with this representation is that three random points in the plane do not lie on a circular arc if they are colinear, but such an invalid definition can easily be detected. Even if they do, the middle point may not lie on the integral grid. For example, the circular arc in the first quadrant of the unit circle cannot be represented by three control points.

The first representation is preferred, because it is the only representation that does not have to be verified to represent a circle. However, a graphics editor should have three different commands for drawing circular arcs:

1. The user picks two points: the center and a point on the arc to define a full circle.
2. The user picks three points: the center, the beginning of the arc, and the ending of the arc. The distance between the first two points is the arc's radius. The coordinate of the ending of the arc is only used to calculate the angle that the arc sweeps out so the coordinate does not have to lie on the arc. This command is used to draw pie charts and arcs that sweep out a multiple of 90 degrees.
3. The user picks three points: the beginning and ending of the arc, and another point that is supposed to be on the arc. The editor fits this last point to obtain an arc that is as close to the desired one as possible. This command has a spline flavor and would be used to draw the three arcs on the schematic symbol for an OR gate.

In order to represent a schematic diagram that contains classical logic gates, the drawing grid must be several times coarser than the integral grid or else the tips of the input pins for a NOR, OR, or XOR gate will *not* lie exactly on the arc that the pins lie on when the gate is drawn with real ink.

A Josephson junction might be patterned as a filled circle and a JFET gate might be patterned as a filled doughnut. An unfilled semicircle might be part of the schematic-symbol for an AND gate and an unfilled circle might be part of the symbol for a voltage source.

A data model may extend the doughnut slice to an elliptical shape and include splines for

making free-form drawings.

A *label* is defined by a string constant that gives the characters in the label itself, a string constant that gives the character font, and an integer constant that gives the font's height. A control point defines where the label is to be placed subject to a justification such as "left", "center", or "right":

```
SHAPE
  LAYER blue
  JUSTIFICATION left
  LOCATION 0 0
  HEIGHT 10
  LABEL framework
  FONT roman

  LOWER LEFT 0 0
  COMMENT The x-coordinate of the upper right corner is an upper bound,
  COMMENT because the width of the letters vary with the font and
  COMMENT perhaps even the characters in the label.
  UPPER RIGHT 90 10
```

More refined justifications may be desirable including justifying relative to other shapes. The height can also be represented by the distance between the location control point and an offset control point. The justification can also be represented by the direction vector defined by the offset and location control points. A document with multiple fonts could be stored in a cell view on an ink layer.

2.3.8. Geometric Functions

Geometric functions represent the relation between the net list information and the geometric information that implements or pictures it. Any of the shape types—such as circle and rectangle—can serve any of the four geometric functions. Examples of geometric functions are illustrated in Figure 2.6.

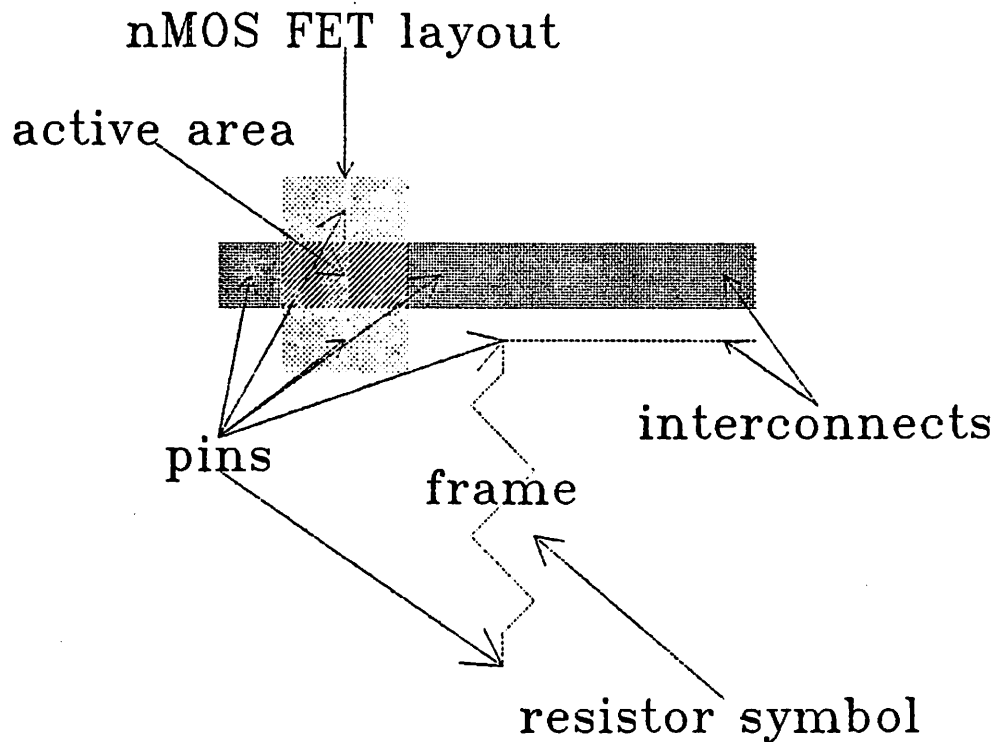


Figure 2.6: Examples of geometric functions.

A shape associated with a net is an *interconnect*. Wires, contacts, and edge-connectors are interconnects.

A shape associated with a terminal that represents the piece of material to be connected to in order to determine the terminal's voltage is a *pin*. In Figure 2.6, two resistor pins and four FET pins are illustrated. A vector functioning as an interconnect is connected to the top resistor pin and a line functioning as an interconnect is connected to the right FET pin.

A shape that is not an interconnect or a pin, but represents an instruction for manufacturing is an *active area*. In Figure 2.6, the rectangular channel of the FET functions as an active area.

A shape that is not an interconnect or a pin, nor does it represent an instruction for manufacturing is a *frame*. In Figure 2.6, the jagged vector functions as a frame. It is intended that shapes functioning as interconnect, pin, or active area be placed on layers that represent instructions for manufacturing. Again, the meaning of layers is application-dependent.

A *frame* is a shape that functions as part of an abstraction. For example, the schematic-symbol for an AND gate is an abstraction. As another example, a *protection frame* [36] summarizes a layout and it is forbidden to place a shape too close to one. The space between protection frames may be used as routing channels. Protection frames are the topic of the last section of Chapter 6.

Geometric functions are also important in an automatic routing method such as the gate-array design approach. As the first step in a design, the user defines the net list in a schematic-diagram view. As the second step, the router is invoked to compute interconnects for all nets. For each terminal in the schematic-diagram view, there is a corresponding terminal in the gate array view. In the schematic-diagram view, each terminal usually has one pin, but in the gate array view, each terminal may have more than one pin to give the router more freedom. As the third step, the designer can route any overflows. As the fourth step, equivalent circuits for the routed nets are computed by an extractor and fed back into the schematic-diagram view.

The nMOS [8] layout view of the resistor resembles a barbell as illustrated in Figure 2.7.

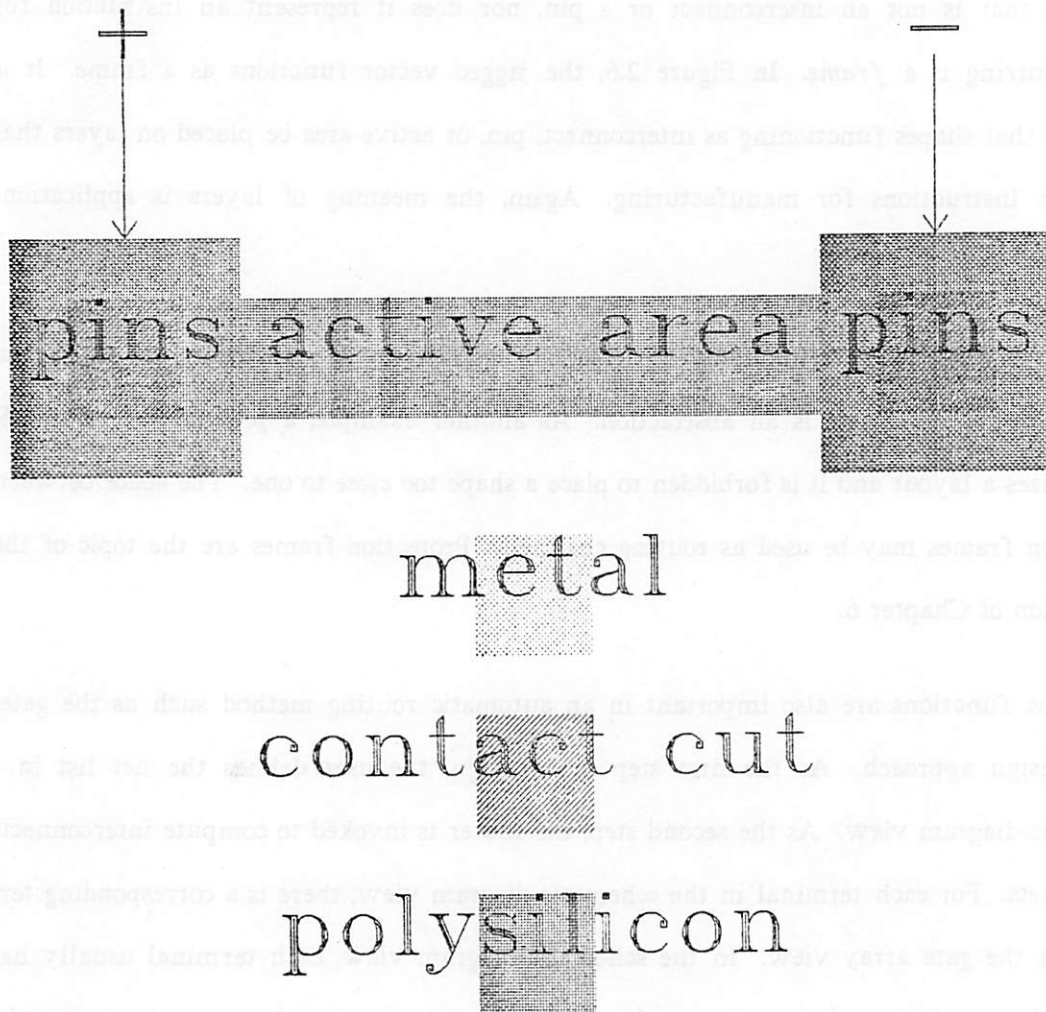


Figure 2.7: A 1 kilohm polysilicon resistor.

The shapes in this view follow:

- The "bar" represents a polysilicon rectangle functioning as active area. Because the resistance of polysilicon is assumed to be 50 ohms/square, the rectangle's width is 2 and length is 1K ohm divided by 50 ohms/square divided by the width or 10. A line shape with two reference points would also be valid.
- Each "weight" represents a polysilicon to metal contact of minimum size made up of three rectangles functioning as pins. There is one contact for each of the two terminals of the resistor. Thus, there are three pins for each of the two terminals

of the resistor.

2.3.9. Placed Instances

In the presentation of net list information, masters were said to be *used*. Actually, when a geometric view, such as layout or schematic-diagram, of a master is used, it is geometrically transformed to *place* it in the coordinate system of the dependent. The transformation may involve rotation about the origin, flipping about the axes, scaling, and translation. Manhattan transformations speed transformation of masters— see Chapter 3—and can be represented by a point to encode the translation and three bits to encode the eight possible combinations of flips and rotations.

A designer often creates a formal terminal and connects it to an actual terminal merely to rename the actual terminal. In layout, the pins of each actual terminal to be renamed are copied into the dependent where they are made pins of the formal terminal whose name is the new name. A naive display of the dependent would show actual pins overlapping formal pins which is fine, but if the name of the terminal a pin is associated with is displayed inside the pin, terminal names on overlapping pins will also overlap and be unreadable. A possible solution is to have the data model decree that if pins on the same layer overlap, then only the name of the terminal associated with the pin that is closest to the dependent in its hierarchy will be displayed. If the pins are all at the same level of the dependent's hierarchy, then a more sophisticated strategy must be chosen for displaying terminal names which take into account the area around overlapping pins.

2.3.10. Geometric Regularity

In IC layout, often views are laid out as 2-dimensional arrays. In memories, each array element is an instance of a master that processes one bit of information.

The KIC 2 graphics editor [37] is built on top of a CIF [8] database package that uses the CIF user extension command to encode a 2-dimensional array of instances of a single master.

This command has the form:

OA s m n dx dy;

which means to create an m by n array whose elements are instances of the symbol s with a distance of dx between columns and a distance of dy between rows. This 2D, homogeneous array object—homogeneous as only one symbol is replicated—has been used with some success. Many graphics editors have used this approach successfully. The main functions of such a simple array object are to use less space than would be used if each element of the array is stored as a separate instance, capture designer intent, make it unnecessary to replace instances if the master's dimensions change, and make it easier for layout analysis tools to avoid rechecking the same inter-element interactions repeatedly.

This simple array object has several problems. The first problem is that often, adjacent elements are the mirror images of each other in order to enable supply sharing. This can be represented as a simple array object that references a "dummy" view composed of 4 instances arranged as a 2 by 2 array. This level of indirection is clumsy and artificial. The second problem is that ROM arrays are not purely homogeneous though simple RAMs or register files are. The third problem is that all masters do not necessarily have the same dimensions. This is true of the layout fragments that can be arranged as an array to form a PLA. The fourth problem is that bus terminals on each element are not suffixed with a bit subscript as in:

operandA < i >

Note that in layout, a bus is represented by all terminals whose names match after subscripts have been removed from their names. Another problem is that even if the master has no layout rule violations and an equivalent circuit has been extracted for it, the equivalent circuit for the entire array must be stored. It would be much more space-

efficient to store how an element that is not on the border of the array connects to its neighbors so that the equivalent circuit for the entire array can be derived, but is not stored explicitly. The final problem is that it is impossible to alter any of the elements without modifying all of them.

The solution is a new breed of array object that enables each element of the array to be a code that indexes a table. It is assumed that elements in the same row have the same height and elements in the same column have the same width. However, each row can have a different height and each column can have a different width. Each table entry stores the following information:

- Master.
- Rules for neighbors on the left, below, on the right, and above.

Each neighbor rule has the form:

- Amount of overlap.
- A list of pairs of pins. In each pair, one pin is the master's and one pin is the neighbor's.

2.3.11. Geometric Operations

A tool that processes layouts performs repeatedly geometric operations such as 2D searches. Sometimes a layout contains thousands of shapes. Thus, the tool's data structures affect its performance. If the framework's database is going to perform common geometric operations on the behalf of tools, then the framework's database must employ data structures that perform efficiently. The types of common geometric operations and many of the common layout tools that use them follow.

2D Geometric Operation Taxonomy	
Terms for Operation Type	Tools
rectangular range query area enumeration windowing	layout rule checker display tool extractor graphics editor extractor
nearest neighbor search	graphics editor
cover search ray casting	compaction tool maze router river router
touching intersection	diagram parser extractor
recursive touching recursive intersection	diagram parser extractor
scanning	layout rule checker extractor compaction tool routing channel generator
incremental shape merging	graphics editor
mask operations	mask modification tool
boxing pattern factoring	pattern-generator code-generator display tool

Each of these geometric operations is illustrated in Figure 2.8.

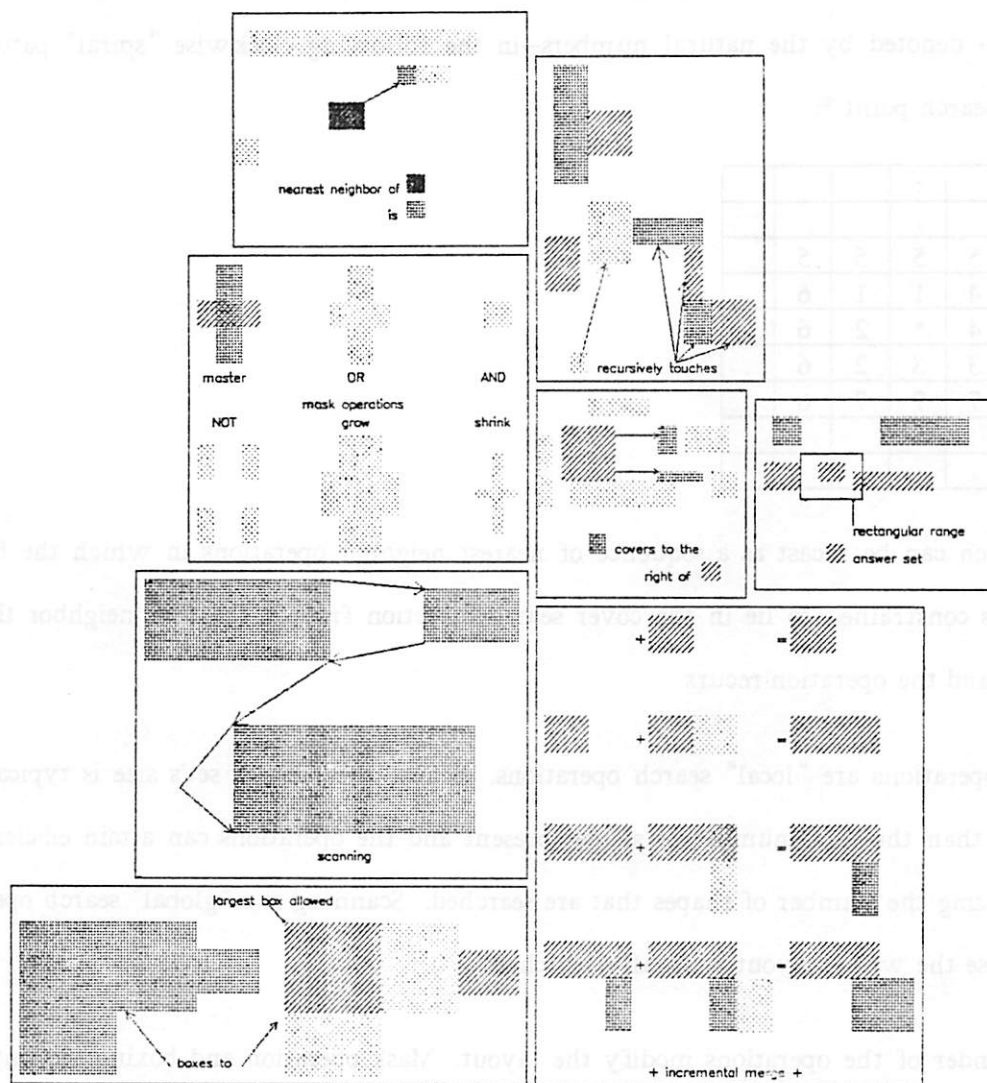


Figure 2.8: Geometric operations.

One operation can often be recast as another. Touching can be recast as a range query in which the range rectangle is the bounding box of the shape that is being touched. Recursive-touching invokes recursive-touching recursively to calculate the transitive closure of the touching relation. Nearest neighbor can be recast as a sequence of range queries with each successive range rectangle being larger than, and enclosing, the previous one. It may be

possible to derive an even more efficient nearest neighbor operation by selecting the range rectangles-- denoted by the natural numbers--in the following clockwise "spiral" pattern about the search point *:

			:			
			:			
	8	5	5	5	5	
	8	4	1	1	6	
...	8	4	*	2	6	...
	8	3	3	2	6	
	7	7	7	7	6	
			:			
			:			

Cover search can be recast as a sequence of nearest neighbor operations in which the first neighbor is constrained to lie in the cover search direction from *, the first neighbor then becomes * and the operation recurs.

The first operations are "local" search operations, because the answer set's size is typically much less than the total number of shapes present and the operations can attain efficiency by minimizing the number of shapes that are searched. Scanning is a "global" search operation, because the whole layout is usually searched.

The remainder of the operations modify the layout. Mask operation and boxing are implemented by a variant of scanning. Incremental shape merging can be implemented by a variant of scanning or recursive touching.

The most popular geometric data structures are square bins [38], 4-d trees [39], and quad trees [40]. Less popular is the bit map of [41] mentioned here for completeness. Because of the way Squid is structured as presented in Chapter 3, it would be straightforward to use any of these data structures.

Square bins have to be implemented as a sparse matrix or the bins waste too much space on empty bins. At the floor plan level of an integrated circuit layout, there are a few large

instances that take up much of the space and there is no sense binning them. There are several ways to handle shapes that intersect more than one bin.

These problems lead to bin data structures like quad trees in which the bins adapt as shapes are inserted and deleted. Quad tree operations require more arithmetic than the other data structures.

4-d trees are an application of Bentley's k-d trees that uses a non-standard search key space. The reference [39] indicates that 4-d trees work very well. The trees trade time for space in internal nodes. In a highly dynamic application, it is unclear how costly in time tree balancing is. The trees can answer cover search and range query. Until recently, they have been the state-of-the-art.

Recently, two new data structures have been proposed: vertical and horizontal bins [42] and corner stitching [43]. I have implemented a variant of the former that is the subject of Chapter 4. The latter shows great promise, because it can be used to implement virtually all of the operations efficiently. All of the other data structures represent only the shapes themselves and the empty space between shapes is implied. An important contribution of corner stitching is that its explicit representation of empty space enables efficient local searching.

2.4. Database

In Chapter 1, database approaches were presented and the data structure package approach was seen to be the most promising. Thus, the objects and operations presented in this chapter are best implemented as a data structure package.

It does not matter how the information is kept in *secondary* storage. If a DBMS was constructed that solved the problems alluded to in Chapter 1 and Section 2.1, it could work

well. Also, a flat file system, or a hierarchical file system can work well. All three make it possible to cluster the information in a single view and support the access control and locking operations. When a view's *contents* are referenced or accessed for the first time, they must be compiled into a *primary* storage data structure that makes it possible for the operations to be *efficient*.

A virtual memory primary store is desirable because in a single *session* or process lifetime, a client inspects many views often, but not all at once. A virtual memory is not critical, because the data-structure package can easily demand-page view data structures between primary and secondary stores. However, neglecting operating system space, the primary store should be at least several times the size of the largest view and the largest view should be paged only if it has not been accessed for a long time. Thus, the largest view and the masters it uses one level down in its instance hierarchy should all fit in the primary store. The largest views are typically the *root* or *top-level* views of instance hierarchies in the case of IC layout.

The implementation of dependency sets is not straightforward. It is well known that it is hard, but possible, to insure that update to the dependency set *and* update of the dependent is atomic. A better solution is to update the dependency set when the dependent is saved to secondary storage. Thus, the dependency set would not have to be updated if no new instances were created or deleted *before* saving the view.

The choice of the programming language in which to implement the objects and operations described here is a deep topic. The languages of today differ greatly in programmer convenience and in the quality of execution of programs written in them. The latter is partly determined by the language itself, the computer, the language processor, and the programmer. From a semantic point of view, the choice does not matter as long as the language has long identifiers for abstraction; subroutines, methods, messages, or procedures for implement-

ing operations; macros, in-line code, or open code for making operations very efficient; dynamic creation or allocation of data space for implementing objects; and subroutine pointers for implementing demons [20], triggers, or attached procedures that make it possible for a client to *extend* the operations *without* modifying the source code for them. Most modern high-level programming languages have these features.

Processes that share heap and semaphores are very desirable, but not critical. If the language or its packages have processes that do *not* share heap, processes must communicate via secondary storage which leads to replicated information and slow communication.

2.5. Hardware Independence

A CAD framework *might* have to be hardware independent, because the hardware must be matched with the tools and tools vary greatly in their computer and user interface requirements. If the graphics editor is being used for schematic diagram capture, then a monochrome or four-plane frame buffer is sufficient while for leaf cell layout, a frame buffer with a larger number of planes may be necessary. A microprocessor with no floating point instructions may be enough for a graphics editor while the SPICE circuit simulator [11] can benefit from a processor with matrix instructions [44].

There are several ways to construct a computer-independent framework: use a portable language that invokes system calls through a run-time package, use a portable OS, or use an OS that is available on a range of computers. Examples of the first way include FORTRAN, SMALLTALK, MAINSAIL, and PASCAL. An example of the second way is the UNIX OS. The first way can emulate the second way. Both ways are complicated by the fact that each vendor's product deviates from the standard so that the framework builders must first study all of the products for a lowest common denominator and resist the temptation to use a feature that is not in the lowest common denominator.

As noted in Chapter 1, a framework that uses a model or virtual graphics terminal package such as MFB can be terminal independent. It is well-understood [31] how to write a virtual graphics terminal package.

2.6. User Interface

There must be an inspection service—a set of commands for inspecting all the information in the database. Various display services must be provided that can be invoked by this inspection service as well as the services that edit the database. For example, a tree drawing service would be useful for displaying name spaces, instance hierarchies, and dependency hierarchies. A service that can draw the shapes of Squid circuit views will be used by all graphics editors. A service to draw waveforms would be useful to tools such as simulators.

The editing services must enable the contents of circuit views, and cells and views of them to be edited. A text editor can serve as a basic editing service for stranger views.

The scheduling services must enable tools and other services to be invoked, controlled, and their resource utilization and progress monitored. There is a wide range of capability in scheduling. The most basic capability enables the client to invoke one tool at a time and optionally abort it. The most sophisticated capability enables the client to fully control a set of tools executing simultaneously as a set of processes. Processes can be started, interrupted, resumed, re-started, aborted, and set up to pause under certain conditions.

2.7. Summary

In this chapter, the requirements for an ideal circuit CAD framework have been described. In the following chapters, an experimental framework is described which implements many of the features presented in this chapter.

CHAPTER 3

THE SQUID PACKAGE

3.1. History

As stated in Chapter 1, the framework is built on top of 4.2 BSD UNIX. Thus, the Squid package was constrained to be written in a 4.2 BSD UNIX programming language.

When the implementation of the Squid package was started two and a half years ago in September of 1981, the languages and packages available were Berkeley's Lisp dialect Franz, Pascal, FORTRAN 77, C, and the EQUEL query language for Berkeley's INGRES [45, 46] relational DBMS.

Franz was considered carefully largely because of my belief in Lisp. Most of the programs in use by my research group were written in C, Pascal, and FORTRAN which do not interface easily to Franz. Franz's mark and sweep garbage collection is poor for interactive tools. At this time, Franz starts up with 1.1 megabytes of virtual memory—quite a bit for a work-station. Access-oriented and object-oriented [23] programming packages like GLISP [22] and Flavors [20] were not available on top of Franz. On the other hand, the other languages available did not have a source level debugger, strings, lists, an evaluator procedure like EVAL, Franz-based graphics and graph-making packages under development by Berkeley's Vaxima project, etc.

EQUEL is an excellent relational calculus with the power of an ALGOL family language as it is embedded in C. I designed a relational schema for the CIF data model which led to a benchmark [17] comparing INGRES and my CIF database package [37] CD. Problems with INGRES were found—the most important was very poor performance relative to CD. The

causes include:

1. INGRES' caching is poor.
2. Its access methods are limited and hard to control by the client.
3. It does not cluster data spread across relations. Often, the client is willing to declare the cluster in the schema. Example clusters are view of cell and CIF symbol.
4. The data model is missing a multiset feature. The client simulates the multiset feature via integer-valued "id" domains whose values serve as identifiers of sets. Each set element may be spread across many relations. To collect a set, the same integer domains in different relations are joined on the "id" of the set.
5. Queries cannot be built at run-time and interpreted via EQUQL.

After evaluations of all of these languages and packages, I decided to build a special-purpose, access-oriented [23] package in C. C is used widely and the 4.2 BSD UNIX compiler generates fast code reliably. The ramifications were that the only ways that the user could extend the data model was to use stranger views or property lists; because UNIX does not have a C interpreter, unplanned queries could not be supported; graphics, graph-making, string, list, heap management, dynamic linking, circuit view file parser, and circuit view file generator packages had to be built; and a circuit view file format had to be designed. Parsing and generation could have been avoided by building a tool that would read and write heap images. The form of a heap image is defined in C data structure declaration files termed *header files*. This tool would be table-driven by a data type table built from header files.

3.2. File System

UNIX's secondary store—the UNIX file system—is a rooted tree whose internal nodes are directories and whose leaves are files. Thus, a directory holds information about files and sub-directories. Each node has a name that need not be unique, but each node at the same level of the same subtree must be named uniquely. The name / (slash) denotes the root

directory of the tree. The full name of a node named **nodeName** at level **n+1** of the tree is:

/nodeName1/.../nodeNamen/nodeName

where the **nodeName_i** are the elements of the simple path to the node starting from the root. Only the first slash denotes the root—any remaining slashes separate node names. Such a name is termed a *path name*. Each login process possesses a directory termed its *working directory*.

In the Squid system, each cell is represented by a directory node whose name is the same as the name of the cell. Each view of a cell is represented by a file in the directory representing the cell. The file's name is usually the same as the view type's name, but does not have to be. Circuit and stranger views are different, because the contents of a file representing a circuit view can be parsed and generated by the Squid package, but the contents of a file representing a stranger view is unconstrained. *View-of-cell* is the only inter-UNIX-node relationship in circuit and stranger views. In addition, *instance-of* is an inter-UNIX-node relationship in circuit views. Other inter-UNIX-node relationships can be represented by property lists. For clarity, from here on a file representing a circuit view will be termed simply a *view file*, a file representing a stranger view will be termed a *stranger view file*, and a directory representing a cell will be termed a *cell directory*.

3.3. Path Mechanism

Each UNIX process has a store with it that is used as a property list termed an *environment*. Each element of the list is termed an *environment variable*. When a process invoking the Squid package starts, the value of the process' environment variable **SQROOT** is the *current cell* if the variable is bound. Otherwise, the process' working directory is the *current cell*. Each cell directory represents a single name space—because the file system is hierarchical, the name space is also hierarchical. Thus, the current cell is equivalent to the current name

space and vice versa. Each cell contains a special stranger view, named the **path** view, that contains the cell's path. Effectively, this view contains a list of views that can be used as masters inside other views of the cell. The name searching algorithm is implemented as described in Chapter 2. Squid operations that change the current cell are presented below.

If the current name space's path is changed, the `<master, instance >` associations of the current cell are not re-evaluated automatically. A `NameSearchMasters` call to force re-evaluation is not in Squid, but should be. If the path stranger view was instead a circuit view with the elements of the path represented in the circuit view's property list, syntax errors in paths would be impossible. A change to a path view could be detected by a Squid demon that could invoke `NameSearchMasters` automatically.

One policy in use showcases this mechanism. In the directory `~/cad/lib/hawk` there are two sub-directories named `technology` and `process`. The cells in `process` contain information about processing lines such as transistor threshold voltage and `metall` spacing. The cells in `technology` represent circuit-level information, such as valid masks and transistor ratios, for each circuit technology like `2PhaseStaticNMOSWithDepletionLoads`.

A client has project directories within the subtree rooted at his home directory where he keeps circuits designed in technology `technology`. To study the impact of a processing line line on his circuits, the client makes a project directory's path:

```
(●
 ~ cad/lib/hawk/technology/technology
 ~ cad/lib/hawk/process/line)
```

3.4. Protection

For each UNIX file system node, its owner can control read, write, and sense-existence-of access for three sets of users: himself, a user group, and everybody but him. Because Squid

package operations on cells and their views are implemented by file-system system-calls, UNIX protection implements Squid package protection.

An *advisory* lock is a lock that can be ignored by an uncooperative user. A node is termed *open* to, in, or by a process, if the process has invoked `open` or `creat` to assert its intent to read or write the node's contents. 4.2 BSD UNIX has an advisory lock feature that can implement multiple-reader or one writer locking. Because only an open node can be locked and there is a UNIX-compile-time upper bound on the number of nodes a process can have open at once, there is the same bound on the number of locks a process can have. If the Squid package used the 4.2 BSD UNIX lock feature and the maximum number of open nodes per process was 30, then if the client tried to access its 31st view, a lock would have to be broken. `creat` can be invoked to atomically create a read-only file if it does not exist and thus can be used to implement locking. Two files are associated with a view file. The first file is the read-only "lock" file. The second file is the "policy" file and contains:

```
<host name, process id, mode >
```

triples. If a view file is being read, then each process that has compiled it contributes a triple, with `mode` equal to `r`, to the view file's associated policy file. In the case of view file editing, each policy file contains one triple and `mode` equals `w`. When a process attempts to lock a view file, it must check the consistency of the view file's policy file, because some of the processes referenced in the policy file may no longer exist as a result of framework or computer crashes. If none of the referenced processes exist, the lock is "broken". At this time, Squid does not implement locking at all.

Because UNIX nodes are not locked, an uncooperative user could cause inconsistencies even if Squid does lock. Suppose an uncooperative user invokes a text editor and edits the contents of a circuit view file. Later, a cooperative client invoking Hawk to edit the view will be told by Squid via Hawk that the view file is corrupt, empty, or a stranger unless the

uncooperative user knew the circuit view file format or was lucky.

UNIX's file system commands do not have enough power to manipulate views, because a view's state is stored in the file system and in UNIX's primary store—the address space of each process that compiled the view. Thus, each file system command has a corresponding Squid package operation that insures that the command is applied to the address space copy, if any, as well as the file system copy. For example, UNIX's `cp`, which copies nodes, corresponds to an operation that copies the current view—part of a process' Squid package state—to the view `dst`. The declaration of this operation is:

```
SQStatus
SQ(sqCp,sqView,dst)
SQView dst;
/*sqCp and sqView are the names of constants.*/
```

or for those unfamiliar with the C programming language:

```
SQStatus /*The type of the value returned by the function.*/
SQ( /*SQ is the name of the function.*/
  sqCp, /*1st formal parameter—the operation to be done.*/
  sqView, /*2nd parameter--the type of the object to change.*/
  dst) /*3rd parameter--the DeSTination view to be copied to.*/
SQView dst; /*dst's type is SQView.*/
/*sqCp and sqView are the names of constants.*/
```

3.5. Dependency Hierarchy

A brute-force solution is to have a registry file, say `~cad/lib/hawk/registry`, on each computer that contains a list of names of cell directories that are roots of circuit hierarchies. To compute a master's dependency set, a procedure searches every circuit view file, reachable from every cell directory that is listed in the registry file of every computer, for an instance of the master. For each master searched, only its instances need to be searched so performance is determined by file system performance. Also, if an instance of a master `master` is created in a view `dependent` while a search is underway for dependents of `master`, `dependent` will not be found unless it already contains a saved instance of `master`. By

constraining the number of computers, keeping per project registry files, exploiting the fact that certain view types could not possibly be instanced in other view types, and storing the instance list at the beginning of view files, the cost of the search can be limited.

The best solution is to have a dependency set file associated with each view file. Change to a dependency set file is arbitrated by locking and the changes are performed incrementally. The lifetime of a dependency set file lock is very short relative to the lifetime of a view file lock. The former is the time to create, update, or delete an instance.

When designing a complex circuit with a complex design method, it is very easy to forget to apply the design method completely. A simple tool that traverses the instance hierarchy representing a circuit and insures that all tools that must be applied in order to guarantee correctness have been used is essential. This is the deferred approach to dependency propagation described in Chapter 2. The stand-alone UNIX program *make* can play the role of this tool. In general, *make* works quite well for this purpose, but several disadvantages follow. First, since *make* does not execute in the same address space as Hawk, the interactive graphics user interface and previously compiled Squid circuit views cannot be exploited. Second, the designer must construct the stranger views that drive *make* by hand which is error-prone. To correct these problems, the framework should be extended to provide a tool of this type.

3.6. Revision Control

Each view file represents the current version of the view. Squid also stores one version back in a backup view file whose name is the concatenation of the view file's name and the string "BackUp".

3.7. The Package from the Public Point of View

3.7.1. Terminology

For realism, the Squid package is presented in terms of the C programming language. The following table should make it possible for readers unfamiliar with C to understand the package:

C Term	Algol Family Synonym
struct	record
struct member	record field
#define	constant
enum	enumerated type
void	procedure
typedef	data type
typedef	type
argument	parameter
malloc	new
free	dispose
static	own
extern	public
union	variant record with no tag

For the reader unfamiliar with C, some further notations must be explained. A comment begins with `/*` and ends with `*/`. Variables named `p` and `q` whose data type is `Type` are declared by:

```
Type p, q;
```

Record assignment is allowed as in:

```
p = q;
```

A variable `pointerToType` whose data type is a pointer to a data type `Type` is declared by:

```
Type *pointerToType;
```

The following statement causes `pointerToType` to point to the storage occupied by `p`:

```
pointerToType = &p; /* Ampersand is the unary "address-of" operator. */
```

If **Type** is a scalar data type, then the value of the relational expression:

```
p == *(&p) /* Asterisk is the unary "indirection" operator. */
```

is always true. Even if **Type** is not a scalar data type, ***pointerToType** and **p** are interchangeable.

If a record data type **Type** has fields **fieldi** of data type **Typei**:

```
typedef struct Type Type;
struct Type {
    Type1 field1;
    Type2 field2;
    ...
};
```

then:

```
p.fieldi
```

denotes the **fieldi** field of **p** and so does:

```
pointerToType->fieldi /* -> is the binary "dereferencing" operator. */
```

The value of the relational expression:

```
p.fieldi == pointerToType->fieldi
```

is always true.

3.7.2. Communication

For each object type in the data model, there exists a single C data type. Object variables are declared by the client and passed as arguments to the Squid routines. When a client accesses or *pushes* a circuit view, the contents of the view file is compiled or *staged* into a data structure in the client's virtual address space. The virtual address space is partitioned into Squid's private segment, the client's private segment, and the free segment. When Squid is

called, input actual parameters are copied by Squid into its private segment and Squid copies information from its private segment into output actual parameters. Thus, communication between Squid and the client is implemented by copying between the allocated segments of the virtual address space. Squid's "copying out" slows retrievals. When a circuit view is *unstaged*, Squid will return to the free segment the part of Squid's private segment occupied by the view. Thus, a client can use Squid as a "view segment cache". It is possible therefore for Squid to execute on a UNIX OS that does not have virtual memory or has a small virtual address space.

Character strings are an exception to inter-segment communication. Client strings are copied into Squid's segment. However, strings retrieved by Squid for the client *are not copied into string pools in the client's segment. Thus, the client must not change a retrieved string.* Copying retrieved strings into client string pools would have kept communication pure, but experiments indicated that it would have slowed retrievals dramatically.

3.7.3. SQ

The package provides one central function named **SQ** that takes the same first three formal parameters on all calls. Having one central function simplifies the implementation of demons presented later in this section. The parameters form the triple:

```
< name of operation,
  name of data type of object that operation is applied to,
  object >
```

The remainder of the formal parameters vary in number and meaning depending on the values of the first three parameters. **SQ** is applied to objects in the current view and has data type **SQStatus**. If a function of data type **SQStatus** does not return the value **sqOK**, then:

```
String SQDiagnostic()
```


can be called to return complete English sentences that explain why the function did not succeed. If the second parameter is invalid, then `sqUndefinedObjectType` is returned. If the first parameter is invalid, then `sqUndefinedOperationType` is returned.

The call:

```
typedef enum {sqGeo,sqTerm,sqNet,sqView,sqInst,sqParm,sqViewStk}
SQObjectType; /* ENUMerated type. */
/*
The terms shape and geometry will be used interchangeably though
the dictionary meaning of geometry is different from shape's meaning.
*/
```

```
SQ(sqCreate, objectType, object, ...)
SQObjectType objectType;
```

allocates an object of data type `objectType` that is initialized via the values of a subset of the members of `object` and may return a unique identifier or key for it as the value of `object.objectsId`. The key is only valid if the current view is the same as the current view was when the key was returned. The keys for each object are:

Object	Key
view	view file's full name
net	net id
shape	shape id
formal terminal	terminal's name
formal parameter	parameter's name
instance	instance id
actual terminal	< terminal's name, instance id >
actual parameter	< parameter's name, instance id >

The user must not trust a key that has been returned if `sqCreate` fails, because, for efficiency, Squid does not validate the integrity of keys each time Squid is called. The call:

```
SQ(sqUpdate, objectType, object, ...)
SQObjectType objectType;
```

changes information about `object`. The call:

```
SQ(sqGet, objectType, object, ...)
SQObjectType objectType;
```

gets information about an object whose key is in `object`. The call:

```
SQ(sqDelete, objectType, object, ...)
SQObjectType objectType;
```

deletes an object whose key is in `object`. The following C fragment gets all of the objects whose members' values match those of `object`:

```
SQStatus sqStatus;
SQID generatorId;
SQObjectType objectType;
... object;
...

SQ(sqBeginGen, objectType, object, ..., &generatorId);
for(;;) {
    sqStatus = SQ(sqGen, objectType, generatorId, &object, ...);
    if(sqStatus == sqEndGen) break;
    switch(sqStatus) ...
}
```

Each pass through the `for` loop returns or generates an object of data type `objectType`.

The advantages of one central routine follow. For each client process, the Squid package has a list of demons.

```
SQStatus SQAttachDemon(demon)
int (*demon)();
```

inserts the function pointed to by `demon` in Squid's demon list.

```
SQStatus SQDetachDemon(demon)
int (*demon)();
```

deletes the function pointed to by `demon` from Squid's demon list. When `SQ` is invoked with anything but `sqGet`, `sqBeginGen`, and `sqGen` as its first actual parameter, each demon in Squid's demon list is invoked with the same actual parameter list that was passed to `SQ`. If the operation deletes an object, then the demons are invoked *before* the operation has completed. Otherwise, the demons are invoked *after* the operation has completed. The distinction between before-demons and after-demons is made in other access-oriented packages [20].

The circuit view file and change log—for undo and crash recovery—formats can be the same and are easy to design if each line is just a textual representation of the actual parameter list of SQ. A format should have the following characteristics: be textual so that debugging and patching is easy, be possible to compile in a single pass, be compact to economize on storage, and be line-oriented so that a revision control tool such as RCS can be used. Squid's format meets these goals. As each line is parsed, the parser action just calls SQ. Note that the format problem is irrelevant to the client user when a data structure package such as Squid is used, but Squid's solution is a windfall of one central routine.

The disadvantages of one central routine include the decrease of performance due to dispatching in SQ and in each demon.

3.7.4. Switching Contexts

The template for a client is:

```

/* Include stream package declarations for stranger views. */
#include <stdio.h>

/* Include Squid package declarations. */
#include <cad/sq.h>

void Client()
{
    SQID viewStackId;

    SQBegin();
    viewStackId = SQCreateViewStk();
    SQPushViewStk(viewStackId);
    /* Body of client. */
    SQPopViewStk();
    SQEnd();
}

```

A view stack is a stack of views. Squid has a stack of view stacks whose top is termed the *current view stack*. The top view of the current view stack is termed the *current view*. The cell the current view is in is the *current cell* and *current name space*. The client may

create as many view stacks as he wishes and switch between them in a first-in-first-out fashion. Suppose many client packages are executing in the same process, a client package caller is processing the current view, and caller calls a client package callee. If callee pushes its view stack when it starts to execute and pops its view stack when it finishes, caller's current view will be the same as when it called callee. The Hawk graphics editor uses one view stack per Hawk window on the screen so that Hawk can avoid invoking Squid's name search feature each time the current window changes.

The view stack whose identifier is `viewStackId` is empty initially, but the current cell is the value of the environment variable `SQROOT`, if it is bound. Otherwise, the current cell is the client process' working directory. To push a stranger view to change or browse through on the current view stack, the following C fragment can be used:

```
#include ...

void Client()
{
  SQView view;
  FILE *stream; /* A pointer to a stream allocated by the stream package. */
  ...
  view.cell = "resistor"; /* Cell named resistor. */
  view.view = "documentation"; /* View type named documentation. */
  view.mode = "w"; /* Writing or changing. */
  SQ(sqPush, sqView, sqStranger, &view, &stream); /* Push. */
  ...
  fprintf(stream, ...); /* "FilePrintFormatted" or write stranger view file. */
  ...
  fscanf(stream, ...); /* "FileScanFormatted" or read stranger view file. */
  ...
  SQ(sqPop, sqView, sqUnstage); /* Pop. */
  ...
}
```

The name search algorithm is executed always when the first actual of `SQ` is `sqPush`. Since Squid does not lock, `SQView.mode` is used only to check the protection on the view file. If `r` is substituted for `w` in `SQView.mode`, the view can be browsed through or Read from exclusively.

To push a circuit view to change or browse through on the current view stack, the

following C fragment can be used:

```
#include ...

void Client()
{
  SQView view;
  FILE *stream;
  ...
  view.cell = "resistor"; /* Cell named resistor. */
  view.view = "sim"; /* View type named physical. */
  view.mode = "w"; /* Writing or changing. */
  SQ(sqPush, sqView, sqCircuit, &view, &stream); /* Push. */
  ...
  SQ(...);
  ...
  SQ(sqPop, sqView, sqStage); /* Pop. */
  ...
}
```

If `sqUnstage` is substituted for `sqStage`, the storage associated with the view will be freed.

3.7.5. The Resistor Cell of Chapter 2

The following C fragment creates the simulation view of the resistor of Chapter 2 assuming it is the current view:

```
SQTerm formalTerminal;
SQParm formalParameter;
...
formalTerminal.netID = NULL; /* Floating net's id is NULL. */
/* SQTerm.instID == NULL means formal terminal. */
formalTerminal.instID = NULL;

formalTerminal.name = "+";
SQ(sqCreate, sqTerm, formalTerminal); /* Create +. */

formalTerminal.name = "-";
SQ(sqCreate, sqTerm, formalTerminal); /* Create -. */

formalParameter.instID = NULL;
formalParameter.name = "R";
formalParameter.valueType = sqReal; /* Real-valued. */
formalParameter.value.real = 1000; /* 1000 ohms. */
SQ(sqCreate, sqParm, formalParameter);

SQ(sqSave, sqView, sqText); /* Save the current view. */
...
```

Note the operation to save the current view.

The following C fragment creates the physical or layout view of the resistor of Chapter 2 assuming it is the current view:

```

SQGeo pin, active;
SQIntegerPoint twoPoints[2];
...
/* The physical view has the same terminals as the simulation view. */
...

active.layer = pin.layer = "NP"; /* nMOS polysilicon mask. */
active.filledP = pin.filledP = sqTrue; /* A filled surface. */
active.manhattanP = pin.manhattanP = sqTrue;
active.function = sqActiveArea;
pin.function = sqTermArea; /* Terminal area is equivalent to pin. */

/* Left weight of barbell. */
pin.geoType = sqRect; /* A rectangle. */
pin.def.rect.l = 0;
pin.def.rect.b = 0; /* Left Bottom corner is (0,0). */
pin.def.rect.r = 4;
pin.def.rect.t = 4; /* Right Top corner is (4,4). */
pin.implements.term.name = "+"; /* Pin for + terminal. */
SQ(sqCreate, sqGeo, &pin);
/*
SQ produces the following side-effects.
pin.geoID is the unique id of the weight.
pin.bb is the bounding box of the weight.
*/

/* Bar of barbell. */
active.geoType = sqLine; /* A line. */
active.def.line.nPath = 2; /* Two reference points. */
active.def.line.width = 2; /* Center line's width is two. */
active.def.line.path = twoPoints; /* Pool of points. */
twoPoints[0].x = pin.bb.r;
twoPoints[0].y = pin.bb.b+2; /* First reference point relative to pin. */
twoPoints[1].x = twoPoints[0].x+10;
twoPoints[1].y = twoPoints[0].y; /* Second reference point relative to first. */
SQ(sqCreate, sqGeo, &active);
/*
active.geoID is the unique id of the bar.
active.bb is the bounding box of the bar.
*/

/* Create the rest of the shapes. */
...

SQ(sqSave, sqView, sqText); /* Save the current view. */
...

```

It would not be hard to change the above fragment into a *module generator* [47] for minimum-width, nMOS, polysilicon resistors. The steps are:

- Create a subroutine named **MinWidthPolySiR**.
- Have the subroutine push a view whose cell name has the form:
`CopyString(cellsName, "MinWidthPolySiR");`
`ConcatenateString(cellsName, FloatToString(R));`
 If the view exists, the subroutine returns.
- Have the subroutine prompt the client for the resistance and store it in the float-valued, local variable named **R**.
- Include the above fragment in the subroutine. Substitute:
`R/resistancePerSquare/minWidth[SQLayerNameToNumber("NP")]`
 for **10** in the fragment. Fetch the values of **resistancePerSquare** and **minWidth** from a process library.

A graphics editor can call the resistor module generator when the user points at a menu selection or types the name of the subroutine. How this task can be performed by Hawk is described in Chapter 5. Another module generator could then call the resistor module generator if desired.

The following C fragment creates the voltage divider example of Chapter 2, assuming it is the current view:

```

SQNet net;
SQTerm formalTerminal, actualTerminal;
SQInst instance1, instance2;
...

instance1.masterCell = "resistor";
instance1.masterView = "sim"
/* Transformation can be given as a matrix or in CIF style. */
instance1.cif = "t 0 0"; /* No transformation. */
instance2 = instance1; /* C allows record assignment. */

instance1.name = "instance1";
SQ(sqCreate, sqInst, &instance1);
/*
instance1.instID is the unique id of instance1.
instance1.bb is instance1's bounding box.
*/

instance2.name = "instance2";
SQ(sqCreate, sqInst, &instance2);

```

```

/*
instance2.instID is the unique id of instance2.
instance2.bb is instance2's bounding box.
*/

formalTerminal.instID = NULL;

net.name = "net1";
SQ(sqCreate, sqNet, &net);
/* net.netID is net1's unique id. */
formalTerminal.netID = actualTerminal.netID = net.netID;

/* Create dividedVoltage terminal and connect it to net1. */
formalTerminal.name = "dividedVoltage";
SQ(sqCreate, sqTerm, formalTerminal);

/* Connect the rest of net1. */
actualTerminal.instID = instance1;
actualTerminal.name = "-";
SQ(sqUpdate, sqTerm, actualTerminal);
actualTerminal.instID = instance2;
actualTerminal.name = "+";
SQ(sqUpdate, sqTerm, actualTerminal);

/* Create and connect net2 and net3. */
...
...

```

3.7.6. Range Queries

The "special shape generator" performs range queries and has two calls:

```

SQStatus SQSpecialBeginGen(area, layerFMask, hierarchyLevel, generatorId)
SQBB area; /* Rectangular area. */
int layerFMask[SQMAXLAYERS*4+1][2]; /* Match criteria. */
int hierarchyLevel; /* Match criterion. */
SQID generatorId;

SQStatus SQSpecialGen(generatorId, /* Generator id. */
    geo, /* A shape. */
    integerPath, nIntegerPath, /* Pool of points. */
    NULL, 0, /* Present for historical reasons. */
    instIDs, nInstIDs) /* Hierarchy node of generated shape. */
SQID generatorId; /* An SQID. */
SQGeo *geo; /* A pointer to a shape record. */
SQIntegerPoint *integerPath;
    /* Variable-length vector of data type SQIntegerPoint. */
SQID *instIDs; /* Variable-length vector of data type SQID. */

```



```
int nIntegerPath; /* An integer. */
int *nInstIDs; /* Pointer to an integer. */
```

Calling `SQSpecialBeginGen` initializes the generator to generate all shapes in the current view that *match* the following criteria:

- Shape is not deeper than level `hierarchyLevel` of the instance hierarchy rooted at the current view. `hierarchyLevel = 1` matches shapes in the current view only, `hierarchyLevel = 2` matches shapes in the current view and in masters of all instances in the current view, etc.
- Shape intersects the rectangular area `area`.
- Shape's layer and geometric function numbers match a pair in `layerFMask`. For example, if `layerFMask[0][0] = 1` and `layerFMask[0][1] = (int)sqFrame`, shapes on layer `SQLayerNumberToName(1)` functioning as frames would be matched. Each:


```
< layerFMask[i][0], layerFMask[i][1] >
```

 represents a pair:


```
< layerNumber, geometric function number >
```

 The pair:


```
< -1, ... >
```

 must be the last element of the array.

The search of the instance hierarchy is *depth-first*. Each time `SQSpecialGen` is called, `*nInstIDs` should be the length of the vector `instIDs`. Each time `SQSpecialGen` returns:

```
instIDs[0], ..., instIDs[*nInstIDs-1]
```

represents the simple path down the instance hierarchy to the shape.

3.7.7. A Demon

An example of a demon that watches for changes to the nMOS, polysilicon layerFMask follows:

```
#include <stdio.h>
#include <cad/sq.h>
/* Include the variable-length formal parameter list package. */
#include <varargs.h>

static SQObjectType objectType;
static SQOperationType operationType;
static va_list ap;
```

```

static SQGeo np, *pNp;

int NPDemon(va_alist)
va_dcl
{
    va_start(ap);
    operationType = va_arg(ap, SQOperationType); /* Fetch first actual. */
    objectType = va_arg(ap, SQObjectType); /* Fetch second actual. */

    if(objectType != sqGeo) return; /* Uninteresting operation. */

    switch(operationType) {
    case sqCreate:
        pNp = va_arg(ap, *SQGeo); /* Fetch third actual. */
        break;
    case sqUpdate:
    case sqDelete:
        np = va_arg(ap, SQGeo); /* Fetch third actual. */
        break;
    };

    va_end(ap);
}

```

3.8. The Package from the Implementation Point of View

In this section, *private* refers to details about the implementation of Squid that the Squid client is not aware of.

3.8.1. The Data Structure

Each public data type declared in the header file named `sq.h` maps to one or more private data types. The central data structure is a hash table:

```
HashTable theViews;
```

of view records of data type `View` indexed by the full path name of the view file. 4.2 BSD UNIX has special file system nodes, termed *symbolic links*, that serve as synonyms for other nodes. If a view file node's name and a symbolic link node aliasing the view file node are passed to Squid, then the view represented by the view file node will be compiled twice,

because the two names will hash differently. Moreover, if the first view compiled is changed but not saved, when the second view is compiled, the primary store representation of the two views will be inconsistent. The solutions are to forbid symbolic links or to have a 4.2 BSD UNIX system call yield the node being aliased by a symbolic link.

Squid stores a string table common to all objects in a process in order to avoid storing the same string many times:

```
HashTable theCommonStrings;
```

Each private data type has a property list and Squid has common code to deal with property lists. Property lists are not hashed—they are self-organizing, linear lists:

```
#define LIST_HEAD(dataType, head)\
    dataType *head;

#define LIST_ELEMENT(dataType, next)\
    dataType *next;

struct Prop/*erty*/ {
    LIST_ELEMENT(next);
    String name;
    SQValueType valueType;
    union {
        float real;
        int integer;
        SQBool bool;
        RefAny refAny;
    }
    value;
};
```

There is a vector of length `SQMAXLAYERS` of layer names indexed by layer number:

```
String layerNumberToName[SQMAXLAYERS];
```

At this time, there is not a hash table of the form:

```
HashTable layerNameToNumber;
```

As a result, hashing layer names to layer numbers so `SQLayerNameToNumber` does not

execute as fast as it could.

A multi-list is a set of elements and an object that all of the elements are associated with. Each view record has single-linked multi-lists of net records of data type *Net*, parameter records of data type *Parm*, and instance records of data type *Inst*:

```
#define MULTI_LIST_HEAD(dataType, head)\
    dataType *head;

#define MULTI_LIST_ELEMENT(elementDataType, next, parentDataType parent)\
    elementDataType *next;\
    parentDataType *parent;

struct View {
    MULTI_LIST_HEAD(Inst, instancesInThisView)
    MULTI_LIST_HEAD(Parm, formalParameters)
    MULTI_LIST_HEAD(Net, nets)
    LIST_HEAD(Term, formalTerminals)
    LIST_HEAD(Inst, instancesOfThisView)
};
```

Given a view, its dependency set can be found quickly via *instancesOfThisView*. If all dependents have not been compiled, then this dependency set will be incomplete. Each net record has a multi-list of terminals:

```
struct Net {
    MULTI_LIST_HEAD(Term, terminals)
    String name;
};

struct Term/*inal*/ {
    LIST_ELEMENT(Term, next)
    MULTI_LIST_ELEMENT(Term, nextInNet, Net, net)
    String name;
};
```

The multi-lists yield fast answers to queries. Given a terminal, the net it is in can be found rapidly. Given a net, all of its terminals can be found rapidly. Given an instance, its dependent can be found quickly. Actual and formal terminals are represented by the same data type:

```
struct Inst/*ance*/ {
    MULTI_LIST_HEAD(Term, actualTerminals)
```

```

MULTI_LIST_HEAD(Parm, actualParameters)
View *master;
};

```

Parameters are treated the same way. Given an instance, its master can be found quickly via **master**.

Each view record has an **SQMAXLAYERS**- by-4 matrix of pointers to geometric index records of data type **OSLIndex**:

```

struct View {
    OSLIndex *layerFPairs[SQMAXLAYERS][4];
};

```

The details of geometric indexing are presented in Chapter 4, but the geometric index is used to answer quickly a subset of the geometric queries presented in Chapter 2. The first dimension of the matrix is a layer number. The second dimension of the matrix is the geometric function number—zero denotes frame, one denotes active area, etc. Thus, a view record yields fast access to geometries associated by:

<area, layerNumber, geometric function number >

A drawback of the data structure is that the maximum number of layers is a compile-time constant so that if a view does not have shapes associated with all possible:

<layerNumber, geometric function number >

pairs, space is wasted and if a view has shapes on more than **SQMAXLAYERS** layers, the client must re-compile Squid. This can be remedied by turning the vector of pointers to geometric index records into a hash table keyed by:

<layerNumber, geometric function number >

A geometric index indexes bounding box records of data type **OSLRect**— a bounding box record points to a shape record of data type **Geo** that the bounding box encapsulates:

```

struct OSLRect {
    SQIntegerPoint lowerLeftCorner,upperRightCorner;
    Geo *shape;
};

```

```

    };

    struct Geo/*metry*/ {
        SQGeoType shapeType;
        union {
            Label *label;
            Polygon *polygon;
        }
        nonManhattan;
    };

```

To change the sort of geometric index used, change the data type of `View.layerFPairs`. Because a shape record points to a label record if the shape is a label, etc., Squid can use the speed and simplicity of a rectangle-oriented geometric index yet still represent non-Manhattan geometries. If the shape is a rectangle, `OSLRect.shape == NULL`.

Changing the data type of instances into `OSLIndex` would speed up range queries for instances in views containing many instances. A public generator for instance range query would have to be implemented and exported.

Each non-view record in the heap is given a non-negative, integer identifier that is unique within the view that contains the record. Each view record houses a monotonically increasing, 32-bit sequence number. When a record is created, the sequence number of the view that contains the record is incremented and it becomes the record's identifier. Identifiers impose an order on the objects in a view that is invariant throughout the view's lifetime and is also useful to clients at times.

3.8.2. Circuit View File Format

When the physical view of the resistor was saved, the view file's contents were approximately as follows. (Though comments are not present in Squid circuit view files, C-style comments and blank lines have been inserted for clarity.)

```

SQUID
/*
By reading the first line,
circuit and stranger views can be differentiated quickly.
*/

/*
PUT the INTegeR value 7 on the
property named squidNextObjectID.
*/
PUT VIEW "resistor" "physical" "w" "squidNextObjectID" INT 7

/*
MaKe a TERMinal named + whose NET ID is floating and INSTance ID
is 0 meaning it is a formal terminal.
*/
MK TERM "+" NET_ID 0 INST_ID 0
MK TERM "-" NET_ID 0 INST_ID 0

/*
MaKe a RECTangle whose unique id is 1,
that is a pin for the TERMinal named +,
that is on the LAYER CP,
that is FILLed,
whose Lower Bottom corner is (0,0),
whose Right Top corner is (4,4).
*/
MK RECT 1 TERM "+" LAYER "CP" FILL LB 0 0 RT 4 4

/*
MaKe a LINE whose unique id is 2,
that is ACTIVE area,
that is on the LAYER CP,
that is FILLed,
whose WIDTH is 2,
whose PATH is (4,2) to (14,2).
*/
MK LINE 2 ACTIVE LAYER "CP" FILL WIDTH 2 PATH 4 2 14 2

/* The five remaining shapes follow. */
--

```

Before Squid became a semi-production package, it used a format very similar to the text format in the example, but instead of a "line" representation of SQ's parameter list, the binary representation of the parameter list was used. The time to write a circuit view file is so fast and only one at a time is typically written that either format yields acceptable performance. The space to store a circuit view file is about the same for either format

unless a string table is used in the binary format.

To propagate dependencies up a deep instance hierarchy, at least the instances of all the masters in the hierarchy must be read at once. Thus, the time to read a circuit view file must be minimized. The time to read a circuit view file is less when a binary format is used. The following table exhibits measurements of the time to read integers in both formats. The "XPort" chip has about 3600 drawn rectangles in it. If it takes 4 integers to represent a rectangle's 4 edges or corners, then XPort can be represented in 14,400 integers.

Number of Integers	User Time (sec)	
	Binary	Text
1000	.1	.4
5000	.4	2
20,000	1.5	8

Thus, a binary format yields about 10,000 integers/sec and a text format yields about 2000 integers/sec. The difference can be attributed to number conversion.

The time to read a text format increases if the text format is parsed. Parsing can permit the client to permute "fields" in a line, and to abbreviate and alias reserved words. Again, if the client is using a data structure package, the format is irrelevant, but to ease the implementation, Squid parses using the UNIX parser and scanner generators [48]. If the client is going to use the format as a hardware description or layout language, then the format matters. The text format the client types may be used as the database format and the language syntax, or compiled into a binary database format. The drawback of compilation is that having two copies costs space.

From the point of view of the implementor, the advantages of a text format are that it is easy to debug, easy to change, and could be used with RCS while a binary format cannot. A binary format is not easy to change at all. A relocatable binary image of the view data structure can be read even faster than Squid's development binary format.

3.8.3. Transformation

To retrieve the shapes represented by an instance, the special shape generator must transform shapes fetched from the master into the coordinate system of the dependent. The Manhattan Transform package is used by Squid and Hawk to transform points in the plane. Because only rotation by a multiple of 90 degrees and flipping about the two axes are possible in the data model, MT uses integer arithmetic and can invert any transformation matrix very quickly. If the original transform is:

a	d	0
b	e	0
c	f	1

then the inverse is:

a	b	0
d	e	0
-ca-fd	-cb-fe	1

Transformation matrix inversion is used in *boxing* tests and for implementing editing-in-context. Boxing tests are described in [31].

The following C fragment was executed to test the speed of transforming points. The fragment transforms a random point *pt* 10,000 times—one transform per iteration of the for loop.

```
int i;
MT *mt;
SQIntegerPoint pt;

/*Allocate storage for and initialize a transform stack.*/
mt = MTBegin();
MTIdentity(mt); /*Assign the identity matrix to the current transform.*/
MTTranslate(mt,100,100); /*Translate the current transform by (100,100).*/
for(i = 10000;i > 0;--i) /*For 10,000 times.*/
    MTPoint(mt,&pt.x,&pt.y); /*Transform (pt.x,pt.y).*/
```

The fragment required:

0.5 user-seconds or 50 microseconds/point

3.9. Conclusions

The Squid database has been the subject of Chapter 3. Though some of the material has been drawn from the application— a framework for circuit CAD—most of the material is independent of the application. The majority of material presents implementations of state-of-the-art methods of computer science such as the demons of access-oriented programming, object-oriented programming, propagation of dependencies, locking, protection, naming, persistent objects, objects with non-trivial history, and context-switching. The "framework for software CAD"— 4.2 BSD UNIX's C programming environment— did not permit these implementations to be as easy as they could have been. Research and development in the areas of programming environments and computer architecture must seek to integrate these methods so that the client need not implement these methods and the client can spend more time at the application level.

CHAPTER 4

ORTHOGONAL SCAN-LINES

4.1. Background

A survey of past work is in Chapter 2.

4.2. Motivation

OSL falls into the class of bin data structures, but OSL's bins are rectangular rather than square so that the problems of square bins can be avoided. There are several reasons why rectangular bins make sense. First, even for a large area to be partitioned, the number of rectangular bins is linear in the sum of the height and width of the area. Thus, space requirements are not excessive. Second, shapes are small and square as in contacts cuts, shapes are rectangular and have a shortest side that is close to minimum line width, or shapes are relatively large but there are not very many per cell as in wells and pad over-glass. Thus, except for the latter class of shapes, as long as the width of each rectangular bin is not less than the minimum line width, a shape will rarely lie in more than 2 bins. Exhaustive search is sufficiently fast for shapes of the latter class. Third, the number of horizontal edges with the same y-coordinate and the number of vertical edges with the same x-coordinate in each cell has been measured to be no more than 10 on a variety of cells. Thus, if the bin width was chosen to be 2λ , then the worst case number of shapes that will be examined in searching for what intersects a single point can be computed. The horizontal bin that contains the point can have 10 edges at each of the 2 y-coordinates it covers. Similarly, the vertical bin can have 20 edges. Therefore, 40 edges could be searched.

4.3. Principles of Operation

Orthogonal Scan-Lines (OSL) is a data structure or index for editing and range querying 2D shapes enclosed by their rectangular bounding boxes. If the 2D shapes are rectangles, then the bounding boxes are the rectangles themselves. Assume this is the case for the rest of this chapter. Let:

upperLeftArea	upperRightArea
middleLeftArea	middleRightArea
lowerLeftArea	lowerRightArea

be an example of the bounding box of all of the shapes on a single layer— there are six **binDistance** lambda by **binDistance** lambda, non-overlapping, rectangular areas in the bounding box denoted by **upperLeftArea**, **upperRightArea**, etc. Let **bb.left**, **bb.bottom**, **bb.right**, and **bb.top** define the bounding box's four edges.

An OSL index partitions the bounding box of the layer into non-overlapping, horizontal, rectangular bins termed *logs* each **binDistance** lambda wide as in the three logs:

... upperLog ...
... middleLog ...
... lowerLog ...

and non-overlapping, vertical, rectangular bins termed *poles* each **binDistance** lambda tall as in the two poles:

:	:
:	:
leftPole	rightPole
:	:
:	:

The log **upperLog** intersects areas **upperLeftArea** and **upperRightArea**. The log **middleLog** intersects areas **middleLeftArea** and **middleRightArea**. The log **lowerLog** intersects areas **lowerLeftArea** and **lowerRightArea**. The pole **leftPole** intersects areas **upperLeftArea**, **middleLeftArea** and **lowerLeftArea**. The pole **rightPole** intersects areas **upperRightArea**, **middleRightArea** and **lowerRightArea**.

`RightArea`, `middleRightArea` and `lowerRightArea`. The number of logs is $(bb.top - bb.bottom) / binDistance$ and the number of poles is $(bb.right - bb.left) / binDistance$.

An example OSL data structure for a small layout fragment is shown in Figure 4.1.

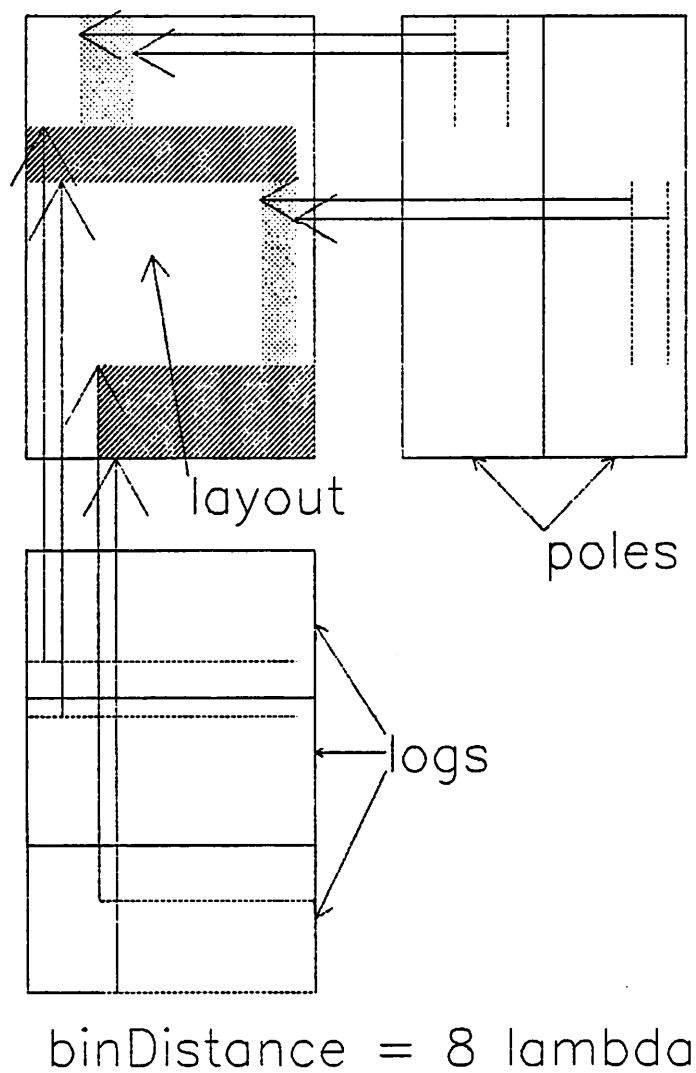


Figure 4.1: An OSL data structure.

If a rectangle is wider than it is tall, it must be associated with all logs that it intersects. However, if a rectangle can lie in a variable number of logs, storage allocation is

complicated. Rather, its top and bottom *edges* are associated with the logs in which the edges lie. Because of the distribution of shapes encountered in circuit CAD, its edges will not be very far apart. Each top and bottom edge contains a pointer to the rectangle it bounds so each rectangle is pointed to by two edges. The vertical case is symmetrical.

To answer a rectangular range query, all bins that intersect the query rectangle, **queryRectangle**, are searched. If **queryRectangle** equals **middleLeftArea** as in:

upperLeftArea	upperRightArea
queryRectangle	middleRightArea
lowerLeftArea	lowerRightArea

then **middleLog** and **leftPole** must be searched. Actually, it is the *rectangles* pointed to by the *edges* pointed to by the intersecting *bins* that are searched.

If **queryRectangle** is small and lies entirely inside a rectangle whose edges are in bins that do not intersect **queryRectangle**, then the rectangle will be missed in the search of the bins that **queryRectangle** does intersect. To solve this problem, **queryRectangle** is always grown by half of the maximum of the widths of the rectangles. Again, this maximum must be close to the minimum line width.

A way of visualizing the pruning behavior is to let the query rectangle partition the bounding box into eight octants surrounding it:

octant1	octant2	octant3
octant4	queryRectangle	octant5
octant6	octant7	octant8

Assuming that the octants' boundaries lie on bin boundaries, octants 2, 4, 5, and 7 are needlessly searched during a search of **queryRectangle**. **binDistance** controls the performance of the data structure.

If **queryRectangle** equals **middleLeftArea**, then **upperLeftArea** (octant2), **lowerLeftArea** (octant7), **middleRightArea** (octant5), and **octant4** are wastefully searched—

Octants 1, 6, 3, and 8 are pruned. Octants 1, 4, and 6 are infinitesimal in this example. By sorting the edges in a bin, some of this wasted search can be eliminated.

A *horizontal scan-line* is a set of edges with identical y-coordinates. *Vertical scan-line* is defined symmetrically. In order to eliminate the aforementioned needless search, sort the edges by y and then x to form a sorted list of scan-lines. Associate the top scan-line in each log with the log the scan-line intersects. Now, assuming a raster scan order search, when searching an edge whose left-coordinate is greater than the right-coordinate of `queryRectangle`:

$$\text{edge.left} > \text{queryRectangle.right}$$

any remaining edges in the scan-line can be skipped. Similarly, when searching a scan-line whose y-coordinate is greater than the bottom-coordinate of `queryRectangle`:

$$\text{scanLine.y} > \text{queryRectangle.bottom}$$

any remaining scan-lines can be skipped. Both of these refinements could be applied to `middleLog`. The vertical case follows as above. This refinement gains a factor of 2 in pruning on the average.

4.4. Algorithm Analysis

The pruning behavior of the OSL data structure is illustrated in Figure 4.2.

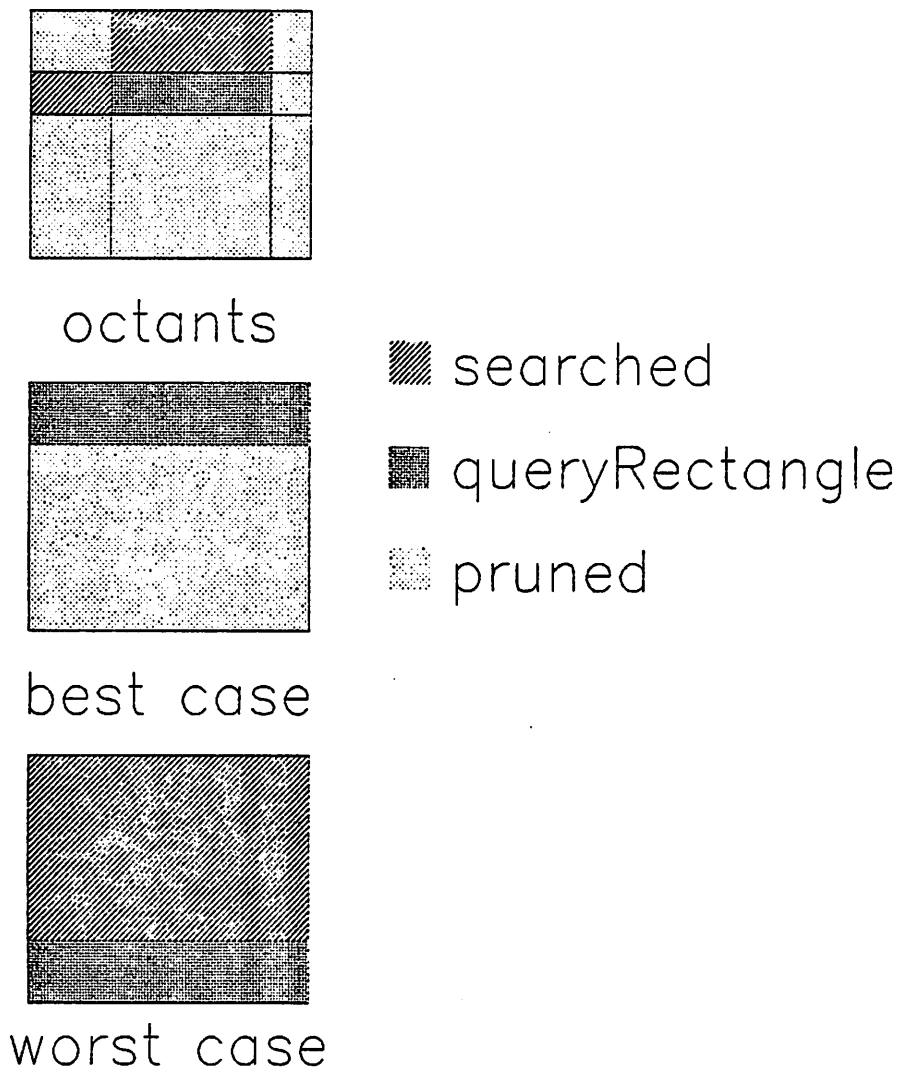


Figure 4.2: Pruning behavior of OSL data structure.

With these refinements, the data structure will search needlessly all shapes in Octants 2 and 4 only. As `queryRectangle`'s shape and location change, the areas of these two octants change. If `queryRectangle`'s shape is constant, these two areas increase as `queryRectangle` moves toward $(-\infty, -\infty)$ and tend to zero as `queryRectangle` moves toward (∞, ∞) . A smaller `binDistance` wastes space on more bins, but improves time, because fewer shapes

are needlessly searched.

An example of the worst case is a `binDistance` lambda by `bb.width` lambda query rectangle whose lower edge corresponds to the lower edge of the bounding box. A search will search the one log containing it and *all* the rectangles in *all* poles. If the rectangles are evenly split among the logs and poles, the pruning will be only a half.

An example of the best case is the same rectangle in the worst case example moved so that its upper edge corresponds to the upper edge of the bounding box. The best case is close to optimal.

4.5. Measurements

```
struct SlowRectangle {
    SlowRectangle *nextSlowRectangle;
    Point lowerLeft,upperRight;
    RefAny rectangle; };
```

An OSL data structure occupies 124 bytes of space for the "header", a pointer per bin, and 52 bytes per rectangle. A linear list data structure would occupy `sizeof(SlowRectangle)` or 24 bytes per rectangle. Thus, an OSL data structure costs more than *twice* the space of a linear list.

Two very different views were used to measure the speed of OSL. The RISC I floor plan is the layout view of the top level cell in the RISC I, nMOS, 45,000 device microprocessor. The floor plan contains over 7000 contacts and wire segments. The layout view of the CMOS SOAR leaf cell is a high-density, p-well, bulk CMOS layout with 260 rectangles in it. The panning and zooming commands of Hawk were used to answer several random-range queries on these cells. At least half of them were on the lower right area of these cells to enable the poorer behavior of the data structure to have an effect.

Cumulative Statistics for Random Range Queries				
Circuit	# Queries	# Found	# Spurious	# Possible
CMOS SOAR Leaf Cell	1013	6311	1240	17164
RISC I Floor Plan	330	30148	10020	266724

Found is what an optimal data structure must search. # Spurious is the number of rec-

tangles searched that did *not* answer the query. # Possible is the number of rectangles that would be searched in an exhaustive search. These cumulative statistics can be summarized in two figures of merit:

Figures of Merit for Random Range Queries			
Circuit	Times Optimal	Factor Better Than Exhaustive	
CMOS SOAR Leaf Cell	1.2	2.27	
RISC I Floor Plan	1.33	6.64	

A measure of how much worse OSL is than optimal is the number actually searched over the number actually found. The closer this is to unity, the better. A measure of how much better OSL is than exhaustive search is the number that would have been searched in an exhaustive search over the number actually searched. The larger this value is, the better.

The raw speed of the data structure is hard to measure. Range queries on the RISC I floor plan cell indicate a search time of 100 microseconds per rectangle counting spuriously searched rectangles. The same workload on a linear list search indicates a search time of 20 microseconds per rectangle counting spuriously searched rectangles. Thus, OSL must search a factor of *five* times fewer rectangles than exhaustive search to *equal* the performance of exhaustive search. The figures of merit in the above table indicate better prunings than this. In the measurements presented in Chapter 5, it will be clear that for zoom and pan, OSL does not justify the space cost, because the data structure search time per rectangle is so much less than the display time per rectangle. In a layout rule checker or extractor, the use of OSL would improve time performance. See the last section of Chapter 6.

CHAPTER 5

HAWK

Hawk displays Squid circuit views in multiple windows and enables the user to invoke tools. Figure 5.1 shows the screen of a terminal under control by Hawk via the MFB package.

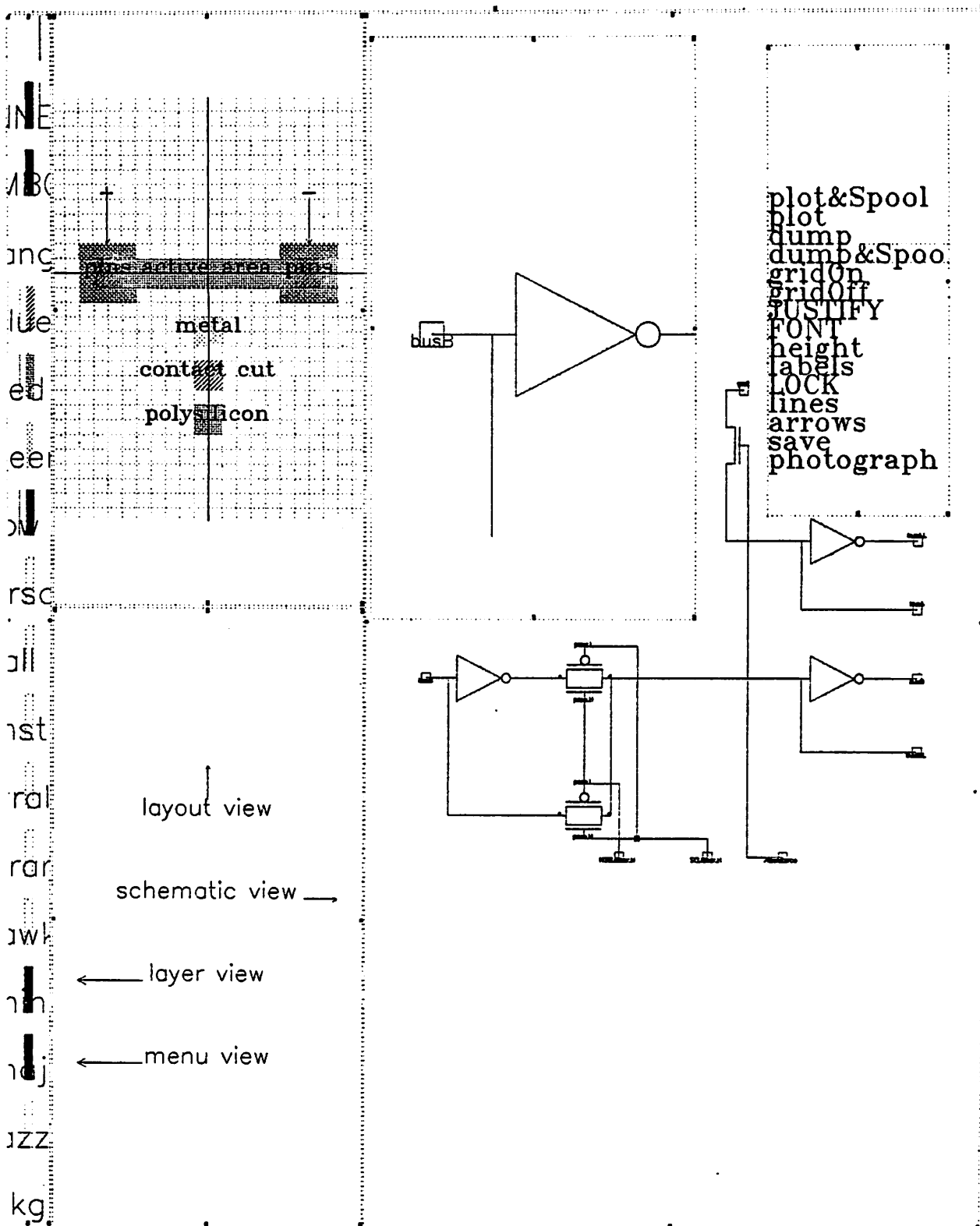


Figure 5.1: Screen controlled by Hawk.

The tiny window at the top of the screen is termed the *typescript* window. User typing is echoed here and status information, help, and diagnostics are displayed here as well. The largest window contains a schematic view and the leftmost window contains a layer menu. Other windows contain a command menu, a layout view, and the legend for Figure 5.1. Because of a bug in the `dumpScreen` command, each window is not completely full.

A *derived shape* is a shape that is computed from information in the database, but is not stored explicitly in the database. Thus, space is conserved and it is easier to keep the display consistent with the database. This term will be used throughout this chapter.

5.1. Initialization

To initialize itself, Hawk must have four sets of information: the graphics terminals it controls via MFB, the layer display-control property lists, command tables, and specific windows to display.

5.1.1. Layers

Hawk uses several special layers—they are special in that it is assumed that they will not be used by clients in Squid circuit views. Otherwise, these layers are like any other. For each window, a derived rectangle that covers the window is drawn on the layer named `backgnd`. The rectangle serves as the "sheet of paper" that the other "ink" layers are drawn on top of. In retrospect, rather than having a layer named `backgnd`, having the layer that serves as the sheet of paper bound on a per window basis would have been better, because menus, schematics, and layouts may be easier to perceive on different colors of "paper" as they have different colors of "ink" in them. Many designers like black paper for schematics, but prefer lightly colored paper for layouts—often a light gray.

Most shapes derived by Hawk are drawn on the layer named `hawk`. Shapes—termed *jazz*—

that highlight other shapes are drawn on the layer named *jazz*. There is a *major grid* line for every five *minor grid* lines. Major and minor grid lines are drawn on the layers named *maj* and *min* respectively.

Each of the four Squid geometric functions on each layer can be given its own *display-control property list* that controls just how shapes are drawn by Hawk. For example, polysilicon pins and frames can be distinguished by adjusting the values on their display-control property lists so that they are drawn in different shades of red. A display-control property list's properties are: color, filled in or not, outlined or not, line style—such as dotted, priority—as explained below, and fill pattern if filled. Thus, a shape associated with a <geometric function, layer > pair may be filled and outlined, just outlined, or just filled.

As in KIC 2 [37], <geometric function, layer > pairs are displayed in priority order. The derived rectangle on the layer named *backnd* is always displayed first. The priorities of the grid line layers can be controlled by the client. Usually, the lowest priorities are associated with <geometric function, layer > pairs that are filled in completely, middle priorities are associated with pairs that are filled with stipple patterns, and higher priorities are associated with pairs that are just outlined. Often, the grid line layers have the highest priority and the grid is drawn using a sparse line style. Thus, the completely filled in shapes are visible *through* the stippled and outlined shapes, and the grid. The layer named *jazz* is displayed last so that highlighted shapes are clearly shown.

Hawk does not use the multiple memory plane capability of MFB at all. If memory planes become inexpensive, it would be beneficial to reserve planes for highlighting, for rubberbanding newly-created shapes, for cursors tracked by Hawk, and for color mixing layers.

Special Squid circuit views termed *layer views* contain the display-control property lists. A layer view's property list contains a Boolean-valued property named *hawkLayerViewP* whose value is *sqTrue*. For each layer view, Hawk expects at most one non-label shape for

each <geometric function, layer > pair. The shape's property list is the pair's display-control property list. When a layer view is displayed in a window, it serves as a layer menu. The leftmost window in Figure 5.1 is such a window. Hawk enables the client to add and subtract layers from a selected set of layers. Tools can use this selected set of layers as operands of commands.

Each MFB has its own layer menu, because the client will desire pairs to be shown on a full color frame buffer differently than on a black and white raster plotter. When Hawk is executed, the value of the `-display` switch is the name of the terminal used for display as well as user input and the value of the `-plotter` switch is the name of the terminal used for plotting. For both terminals, Hawk searches for a layer view whose cell name is the name of the terminal and whose view type name is `symbolic`.

When Hawk is executed, the value of the `-interface` switch is the name of the UNIX device driver that the terminal is controlled by. Omission of this switch causes Hawk to use the teletype port the user logged in at. This is the norm when low-cost terminals with integral RS-232 controllers are used. An example of such a terminal is the AED 512.

A user who does not wish to draw layer views may use a tool named `layerview` that reads a text file and outputs a layer view such as the one displayed in the leftmost window in Figure 5.1. A fragment of such a text file is included here:

```
COMMENT NP denotes Nmos Polysilicon.
LAYER NP
COMMENT — —
LINE_STYLE 347
NO_OUTLINE
FILL
COMMENT Each octal number is a row of an 8 by 8 bit matrix.
STIPPLE 33 cc 33 cc 33 cc 33 cc

COMMENT Saturated red.
COLOR 1000 0 0
FRAME
COMMENT Shapes on <frame, NP > will be shown filled with a stipple pattern,
COMMENT not be outlined, vectors will be dashed, and colored red.
```

COLOR 800 0 0

PIN

COMMENT Shapes on < pin, NP > will be shown just as < frame, NP > shapes,

COMMENT except the former will be colored a darker red.

5.1.2. Commands

Each command is implemented by a subroutine stored in an object file. If an object file containing commands was not linked to the Hawk load module prior to Hawk execution time, then during Hawk execution the object file is *dynamically* linked and loaded if and only if a command in the object file is invoked. A tool is implemented by one or more commands. Defining new commands to Hawk is the mechanism a client uses to add a tool into the framework. There are three types of commands: *long commands*, *key commands*, and *menu selections*. Long and key commands are typed on the keyboard of the display terminal while menu selections are picked or pointed to via the graphical input device of the display terminal. Thus, initialization must build three tables:

< menu selection's name, subroutine's name, object file's name >

< key, subroutine's name, object file's name >

< long command's name, subroutine's name, object file's name >

A menu is just a Squid circuit view termed a *menu view* whose property list contains a Boolean-valued property named `hawkMenuViewP` whose value is `sqTrue`. An example of a menu view is displayed in the window in the lower left corner of Figure 5.1. Each menu selection in a menu view is represented by a shape whose property list contains two string-valued properties. The value of the property named `hawkRoutine` is the subroutine's name that implements the menu selection. The value of the property named `hawk.o` is the name of the object file that contains the subroutine. Thus, a client can draw his own menus.

The tool `menuview` reads a text file and writes a menu view that houses a classical, textual menu. An example of such a menu is displayed in the window in the lower left corner of

Figure 5.1. Each line of the text file can have one of these two forms:

```
LABEL menuSelectionsName subroutinesName objectFileName HELP theHelp
SUB_MENU subMenusName menuCellsName menuViewsName HELP theHelp
```

`menuview` compiles each line into a label on the layer named `hawk` in the output menu view. `theHelp` is the value of the label's string-valued property named `hawkHelp`. When the layer named `help` is selected and the label is picked or pointed to, the value of the property named `hawkHelp` is displayed as help for the command associated with the label.

Each line that begins with `LABEL` becomes a label that will invoke a command when picked or pointed to. The label itself becomes `menuSelectionsName`, the value of the label's `hawkRoutine` property becomes `subroutinesName`, and the value of the label's `hawk.o` property becomes `objectFileName`.

Each line that begins with `SUB_MENU` becomes a label that, when picked or pointed to, will overlay the menu that contains the label with another menu usually termed a *sub-menu*. The label itself becomes `subMenusName`. `menuCellsName` becomes the value of the label's string-valued property named `hawkSubmenuCell` and `menuViewsName` becomes the value of the label's string-valued property named `hawkSubmenuView`. When the label is picked or pointed to, the window displaying the layer view containing the label is updated so that it displays the layer view whose name is:

```
< menuCellsName, menuViewsName >
```

Example `menuview` input is:

```
SUB_MENU WINDOWS windos symbolic HELP Window manager ...
SUB_MENU HAWK ../hawkCommand symbolic HELP ...
SUB_MENU LAYOUT layout symbolic HELP ...
SUB_MENU SCHEMATIC schematic symbolic HELP ...
SUB_MENU SYMBOL symbol symbolic HELP ...
SUB_MENU NLP nlp symbolic HELP ...
SUB_MENU SLIDES slides symbolic HELP ...
SUB_MENU P-LISTS plists symbolic HELP ...
LABEL save save save HELP ...
```

```

LABEL write w w HELP ...
LABEL editByTyping pushByTyping push HELP ...
LABEL editByPointing pushByPointing push HELP ...
LABEL lyra lyra layout/lyra HELP ...
LABEL violation violation layout/lyra HELP ...
LABEL critical critical layout/lyra HELP ...
LABEL re-edit pushBang push HELP ...
LABEL subedit pushInContext push HELP ...

```

A special file named `.cadrc` in the user's home directory has been used to store tool-specific information in the Berkeley CAD environment for several years. Each file named `.cadrc` is divided into sections. Each section may house information used by many tools. The syntax of each section is unconstrained except that each line of the section has the form:

```
reservedWord ...
```

The key table is stored in the `HAWK` section. The form of a line binding a key in the `HAWK` section is:

```
ALIAS key subroutinesName
```

An example is:

```

begin HAWK
  ALIAS r rectActive
  ALIAS e delete
  ALIAS + addPtSel
  ALIAS - subPtSel
  ALIAS t addRectSel
  ALIAS _subRectSel
  ALIAS ^ x dese1
  ALIAS m move
  ALIAS g copyWindo
  ALIAS S save
end

```

A long command is a string whose first character is a colon and whose tail is the name of the subroutine that implements the long command. Thus, an explicit long command table is unnecessary. The problem with not having a long command table is that a subroutine name may be awkward to remember and to type.

The problem with the way long and key commands are implemented is that the object file's name is not stored though the subroutine's name is. This is historical and should be fixed.

A subroutine whose name begins with **HA** implements a *Hawk command*. Otherwise, the subroutine implements a *client command*. If a client command is executing, it can invoke a Hawk command indirectly by calling the Hawk library subroutine **HAListen**, but it cannot invoke another client command. If a Hawk command is executing, it cannot invoke any another command. Thus, Hawk commands *nest* with client commands, but not vice versa. **HAPAN**, **HAzoomin**, and **HAzoomout**, are Hawk commands that can be invoked to adjust windows while in the middle of a client command without disturbing the client command's state. For example, during manual routing, it is often convenient to pan to find jog points.

5.1.3. More Customization

As the final phase of initialization, Hawk dynamically links a subroutine named **HABegin** that can be written by the user to customize Hawk. **HABegin** is stored in a stranger view named **HABegin.o** in the cell named **menus/userCommand**. It is expected that **HABegin** will parse command line arguments that have not been set to **NULL** by Hawk.

After **HABegin** has executed, Hawk displays all of the windows that have been created and invokes the Hawk scheduler described in the next section.

5.1.4. Desktops

A *desktop* is a configuration of windows independent of what is displayed in the windows. At this time, the default **HABegin** implements desktops. This subroutine's first argument is a desktop name and its remaining arguments are a sequence of:

<cellName, viewName, accessMode >

triples that name circuit views to display in the windows of the desktop. The desktops and their functions so far are:

Desktop	Function
slides	Slide-making.
horizontalhop	Editing fat, short layouts.
verticalhop	Editing tall, skinny layouts.
one	One big window.
three	Two little windows stacked and a big window next to them.
zap	Four windows used for layout.

The desktop in Figure 5.1 is **three**.

In retrospect, desktops should be implemented by Hawk in the following way. First, a client draws a desktop *template* as a view type named **template** of a cell representing the desktop. Each rectangle in the view corresponds to a window on the screen. Each such rectangle has three labels inside it:

```
CELL %i
VIEW %i
ACCESS_MODE %i
```

that name the circuit view that is to be displayed in the window corresponding to the rectangle. **%i** is the value of the *i*th argument passed to the desktop *instance* creation command. The command works by letting the user pick or point at a desktop template out of a menu and typing a list of arguments— **%1, %2, ..., %n**. Only one desktop instance can be displayed at a time. A desktop instance can be stored as a view just like its desktop template as long as the **%i** are stored on the property list of the view.

Commands to switch desktop instances should be implemented. One command switches to the desktop instance that displays any view of a particular cell or a particular view. Another command enables the client to pick or point to a desktop instance out of a menu.

5.2. Scheduler

The Hawk scheduler is an infinite loop that invokes a subroutine named **HAListen**. **HAListen** invokes **MFBPoint** which returns when the user has typed a keyboard key or pressed a button on the graphical input device. This section presents the flow of **HAListen**.

5.2.1. Keyboard Command

If the client typed a key, then if the key is a colon, the client is prompted to type a subroutine's name. Otherwise, the subroutine name associated with the key is fetched from the key table. The dynamic linking package is invoked to fetch the pointer to the subroutine that implements the the keyboard command. Finally, the subroutine is invoked if it has been linked. Upon return from the subroutine, **HAListen** returns.

5.2.2. Pressed Button Number One

Assume the user pressed button number one. If the cursor does not lie in a window, **HAListen** returns.

Otherwise, the coordinate of the cursor is computed by looking up the type of the object displayed in the window and calling an object-specific subroutine that returns a string representing the coordinate. So far, it has been assumed that Hawk can display only Squid circuit views in windows, but really Hawk can display other objects as well. A graph-drawing object's subroutine might return a <time, voltage > pair while a Squid circuit view's subroutine would return a coordinate in lambda.

If the window is displaying a menu view, Hawk invokes Squid to fetch the shape intersecting the cursor and retrieves the shape's property list. If the shape represents a submenu, then the submenu is displayed and **HAListen** recurs. This implements the "overlay sub-

menu" command.

If the shape represents a command, Hawk highlights the shape and invokes the subroutine that implements the command. When the command returns, Hawk un-highlights the shape. If the shape is contained in a menu view that is displayed in a pop-up window and the shape's property list does not contain a property named `hawkKeepMenuP` whose value is `sqTrue`, then the pop-up window is erased. Otherwise, the Hawk scheduler will not erase the pop-up window, because it assumes that the menu selection will not change the display and it is likely that the user will pick or point to yet another menu selection when the menu selection returns. This saves the user a few button presses.

If the window is displaying a layer view, Hawk invokes Squid to fetch the shape under the cursor, selects the layer the shape is on if the layer is not already selected and highlights the shape, un-selects the layer the shape is on if the layer is already selected and un-highlights the shape, and `HAListen` recurs. The subroutine `HALayerSelectedP` may be called by a client command to fetch the layers that are selected by the client.

If the window is not displaying a layer or menu view, `HAListen` returns. There are several Hawk routines that a client command can call to fetch the window pointed at, where the cursor was, etcetera.

5.2.3. Pressed Any Other Button

If the client pressed any other button, the window, if any, bound to the button is popped up if it is not already visible. Otherwise, the window is erased. Hawk places the viewport of an invisible pop-up window outside the display terminal's coordinate system. Hawk will not display such windows, but the windows reside in the same data structure as visible windows.

5.3. Demons

Many of the shapes that are computed by Hawk from other shapes (labels for terminal names are computed from pins, for example) are stored explicitly as shapes in contrast to being derived as a by-product of redisplay. This makes redisplay simpler at the expense of space. Let the term *derived shape* be used for such computed shapes even though they are explicitly stored.

When a shape is updated, any of its derived shapes must be updated as well to insure consistency between display and database. Hawk's Squid-demon performs this task. When a view is compiled, as each shape in the view is created, the Hawk demon is invoked by Squid to possibly create derived shapes. When a shape is updated, the Hawk demon is invoked by Squid to update any of its derived shapes. The association between a derived shape and the shape it is derived from is stored on the property lists of the shapes.

The derived shapes implemented by the Hawk demon are labels picturing instance names, terminal names, and instance parameter lists. The code to maintain the association properties and the derived shapes is cumbersome. Only terminal names are still used with success, except that in schematic capture, clients do not wish the names of standard terminals to be displayed. For example, displaying the name `input3` on a NAND gate just creates clutter and transmits no information, because the information is carried in the shape of the symbol for the gate.

If derived shapes are not stored explicitly, redisplay is impacted greatly. Each redisplayed area must be grown to insure that all derived shapes that may impinge on the redisplayed area are collected. Each impinging derived shape must be clipped to the ungrown redisplay area. Conservatively, the grown area must be the entire window and shapes that are derived from shapes that lie outside the window must never be displayed. In retrospect, this space-saving and code-saving alternative is preferred.

5.4. Selection

Like most graphics editors, Hawk enables the client to manipulate a set of selected objects that are highlighted to indicate that the objects are selected.

At this time, Hawk highlighting is implemented by shapes on the layer named `jazz`. Consistency between highlighted shapes and highlighting shapes is insured by the Hawk demon. A shape is highlighted by drawing an outlined rectangle on the layer named `jazz` around the shape's bounding box. Unfortunately, a bounding box is too coarse for shapes such as arcs, lines, and polygons. The "`jazz`" shapes consume considerable space when everything in a view is selected. The selected set is implemented by making a copy of everything selected in a special view. For each set element, the association between the copy and its master is stored in the property lists of master and copy. This implementation is appealing, because selection can be implemented as a set of client commands completely independent of the Hawk kernel. However, the special view consumes considerable space when everything in a view is selected and only objects from a single view can be selected at once.

At this time, either a whole shape is selected or it is not selected, but it is convenient to select control points and edges of shapes so that stretch operations can be cast as a move operation. Early in the implementation of Hawk, control point selection was possible. However, the `jazz` implementation of Hawk had to be bypassed because, in the `jazz` implementation, only one `jazz` shape can be associated with each shape while one `jazz` shape per selected control point or edge was desired. Thus, control point highlighting was implemented in the selection commands. The value of a shape's integer-valued property named `selReferencePointMask` encoded which of the shape's control points were selected. This implementation bounded the number of control points per shape to be the number of bits in an integer. In an unrestricted implementation, this property must be integer-vector-valued. For lines and polygons, when consecutive control points are selected, the redisplay code

should highlight the thus defined segments and edges respectively instead of the points themselves.

In retrospect, each view should have its own list of selected objects as in [25]. Highlighting would be implemented by deriving from these lists shapes on the layer named *jazz*. When all of the shapes on a layer are selected, each one should not be highlighted. Rather, a highlighting rectangle should be drawn around the bounding box of all the shapes on the layer. This saves an enormous amount of redisplay time and storage for the selected set.

Shapes, instances, and terminals can all be selected in Hawk. The latter arises when editing nets such as with the Beaver schematic capture menu presented in Chapter 6.

5.5. User Interface

In summary, the Hawk user interface possesses graphical menus, invocation of commands via keyboard and graphical input, pop-up windows, hierarchical menus, and multiple windows displaying multiple objects.

Hawk pops up windows which may display anything a window may display. This enables a pop-up "part" menu to be implemented as a view that contains instances of parts and that is displayed in a window bound to a button.

Each window has a few rectangles termed *tabs* or *light buttons* on its border that invoke Hawk commands when the tabs are picked or pointed to. The tabs are clearly visible along the borders of the windows of Figure 5.1. At present, the tab in the middle of the top border of a window is used for panning by half the window's height in the positive direction of the y-axis. The tab in the middle of the right border is used for panning by half the window's width in the positive direction of the x-axis. The tab in the middle of the bottom border and the tab in the middle of the left border behave similarly for the negative direc-

tion. The tab in the upper right corner is used for popping out of an in-context edit, popping from a submenu, or emptying a window. Finally, the tab in the lower right corner pops up status information about the window.

5.6. Measurements

The same two views that were used to measure the performance of OSL in Chapter 4 were used to measure the performance of Hawk. Here, the time to zoom in on a single random area was measured. For the leaf cell, the area was the full bounding box and the statistics printed by Hawk are:

334 shapes and 4 user-millisecond/shape

For the floor plan cell, the statistics are:

302 shapes and 3 user-millisecond/shape

The contributions to the 3 or 4 user-milliseconds/shape follow:

Contributions to Show Time Per Shape	
Source	Time
Window Lookup	NA
Set <nd Style	NA
2D Data Structure Search	200 microsec
Instance Transformation	0
Clipping	NA
Window to Viewport Transformation	45 microsec/vertex
Shape Showing via MFB	NA

Each contribution must be counted for each window, but there is only one window in the workloads. Instance transformation is skipped by Hawk for each shape in the source of the instance hierarchy being shown. Each shape in the workloads is in the source and a rectangle. Clipping is not expensive for rectangles. A rectangle can be clipped using compare instructions only. For a rectangle, window to viewport transformation takes 90 microsec. Besides the contributions in the table that could not be measured, there are others: subrou-

tine call overhead—about 30 microsec on a VAX-11/780, detail suppression tests, and derived shape tests.

If Hawk can produce a call to MFB's MFBSHOWRECTANGLE every 4 milliseconds, then Hawk can make a 250-rectangle change in one second. So unless Hawk is tuned by in-line coding, etc., expensive frame buffers will not turn Hawk into a real-time graphics editor.

5.7. Multiple Windows

The data structure that supports display of multiple objects in multiple windows is quite simple. All windows displaying the same object—usually Squid circuit view—are linked into a circular list. Also, all windows are linked into a linear list. Each object is linked to one of the windows, if any, it is displayed in. The rest of the windows the object is displayed in are available through the circular list containing the one window.

This paragraph holds for windows displaying Squid circuit views. The *display view* of a window is the view being displayed in the window. The *edit view* of a window is the view being edited in the window. The *edit view* is a master of an instance in the instance hierarchy whose source is the *display view*. This is used to implement in-context editing and propagation of changes up the hierarchy as presented shortly. Each window displaying a Squid circuit view has an *edit view* and a *display view*.

Hawk does not display a correlation between windows displaying the same area of the same object. It would be useful to have a master cursor appear in the window the graphical input device is in and slave cursors appear in all other windows that display the same object as the window. This could be done with a work-station.

Hawk does not permit windows to overlap, but it could be extended as follows to allow this. Each window is given a priority by the client. When a window is created, it becomes

the window with highest priority. There are also commands to push a window to the bottom thus making its priority the lowest, pull a window to the top thus making its priority the highest, and highlighting temporarily all fully or partially obscured windows. All windows would be partitioned into non-overlapping subwindows. A window is linked to its non-obscured subwindows. Subwindows behave exactly like windows in Hawk at this time.

A mask operation algorithm can be used for this partitioning:

1. Number the windows so that window number one has lowest priority, etcetera.
2. For each window, make a layer with one rectangle on it corresponding to the window and make the layer's name be the number of the window.
3. Akin to the classical AND and OR operations, calculate a MAX operation.
4. For each layer, make the rectangles on it, if any, into subwindows of the window corresponding to the layer.

5.8. Redisplay

When the database is changed, the display of the database may have to be altered to track this change. There are two types of display changes:

1. Change a viewport. For example, a Hawk window is deleted, created, or its viewport changed. Redisplaying the screen is just a huge viewport change.
2. Change an area. For example, a shape is added to a Squid circuit view or the object in a Hawk window is zoomed in on.

First, consider a viewport change:

```
Hawk.Window hawkWindow;
Squid.BoundingBox changedViewport,area;

foreach hawkWindow
  if(NOT Squid.Intersects?(hawkWindow.viewport,changedViewport))
    continue;
  area = Squid.IntersectionOf(hawkWindow.viewport,changedViewport);
  area = Squid.ViewportToWindow(hawkWindow,area);
```

```
Hawk.Display(hawkWindow,area);
```

Second, consider an area change. All Hawk windows displaying the same Squid circuit view are in the same list. Also, all of the Squid circuit view's shapes that intersect the area can be searched for rapidly via Squid. Thus, only one database search of the view's area is necessary:

```
Squid.BoundingBox area;
Squid.Shape shape,g;
Hawk.Window hawkWindow;

Squid.BeginSearch(area);
while(Squid.Search(&shape) != Squid_End_Search)
  shape = Hawk.ClipShape(shape,area);
  foreach hawkWindow
    /*All windows displaying the same object are in a circular list.*/
    g = Hawk.WindowToViewport(hawkWindow,shape);
    g = Hawk.ClipShape(g,hawkWindow.window);
    switch(g.shapeType)
      case Squid_Rectangle:
        MFB.DisplayRectangle(g.locus.rectangle);
    ...
```

If the MFB can clip and transform window to viewport, the inner loop body can be replaced by MFB calls to redefine the window and viewport, and to display the shape. To tighten this, the MFB should have a bank of:

```
< window, viewport >
```

pairs. The banks would be initialized outside the loops and the redefinition call would only have to send a bank number.

Seldom is the changed area a rectangle—in general it is the union of disjoint polygons. However, Squid can only search for the shapes that intersect a rectangle and thus this union must be approximated by rectangles. In most graphics editors, this union is approximated by the bounding box of the union or by the bounding boxes of the disjoint polygons. In Hawk, the former is used most of the time and sometimes the latter.

The former code fragment corresponds to the Hawk subroutine named `HADisplayViewport` and the latter to the subroutine named `HADisplayView`. The declarations are interesting:

```
HADisplayViewport(viewport)
SQBB viewport;

HADisplayView(windoID,area,aloneP,editViewDidNotChangeP)
int windoID;
SQBB area;
SQBool aloneP,editViewDidNotChangeP;
```

`windoID` is the id of or pointer to the window to be redisplayed. `area` is the rectangular area in world coordinates to be redisplayed. If `aloneP == sqTrue`, then *only* the window given by `windoID` shall be redisplayed. Otherwise, *all* windows displaying the same *display* view as the window given by `windoID` will be redisplayed. `aloneP == sqTrue` is used while panning or zooming a single window. `editViewDidNotChangeP == sqFalse` *implies* `aloneP == sqFalse`, because this means a change has been made to the area and one wishes everything on the screen to reflect this change. The change will be propagated *up* the hierarchy. This can be quite expensive and that is why `editViewDidNotChangeP == sqTrue` means no change to the area was made and thus propagation is *not* to be done.

Thus, the implementation of change propagation makes editing in context easy:

```
HADisplayView(...)
{
  if(NOT editViewDidNotChangeP)
    ... /*Let EDIT view be DISPLAY view.*/

  ... /*Display DISPLAY view.*/

  if(NOT editViewDidNotChangeP) {
    /*Propogate changes UP.*/
    foreach instance OF EDIT view
      area' = Transform(instance->transform,area);
      HADisplayView(instance->instanceIN->window,area',...); }
}
```

5.9. Instance Transformation

The transformation package presented in Chapter 3 is used by Hawk to transform points. When editing in context, the client points at the coordinate system of the view being *displayed* and this point must be transformed into the coordinate system of the master being *edited*. Inverting the instance transformation and multiplying the *display point* by the inverse yields the *edit point*.

5.10. Clipping

For each Squid shape type, there is a corresponding Hawk clipping routine. For example:

```
void HAClipRect(clippingRectangle,src,dst)
SQBB *clippingRectangle;
SQGeo *src,*dst[];
```

clips **src* to form zero or one rectangles in **dst[0]*. When clipping an **SQPolygon** or **SQLine**, many shapes can be produced as a by-product of clipping. When clipping an **SQCircle**, between zero and four shapes can be produced.

To clip to the *outside* of a clipping rectangle, the clipper is invoked on the rectangles:

```
SQBB *clippingRectangle;

HAAboveClippingRect(clippingRectangle);
HALeftClippingRect(clippingRectangle);
HARightClippingRect(clippingRectangle);
HABelowClippingRect(clippingRectangle);
```

One clipping routine should have been implemented— at least from point of view of the client of the clipping package— that has the declaration:

```
void HAClipShape(clippingRectangle,src,dst,insideP)
SQBB *clippingRectangle;
SQGeo *src,*dst[];
SQBool insideP;
```

5.11. Window to Viewport Transformation

In this section, the term *window* means a rectangular area in world coordinates while *viewport* means a rectangular area in screen coordinates as usual. Let `viewportDimension` and `windowDimension` be the width *or* height of a viewport and window respectively.

The window to viewport transformation mathematical equations are:

$$\begin{aligned} \text{pixels} &= \text{Round}(\lambda * \text{viewportDimension} / \text{windowDimension}) \\ \text{viewportDimension} &\gg \text{windowDimension} \end{aligned}$$

The finite, integer computer equations are:

$$\text{pixels} = (2 * \lambda * \text{viewportDimension} + \text{windowDimension}) / (2 * \text{windowDimension})$$

Pre-computing yields:

$$\text{pixels} = (\lambda * A + \text{windowDimension}) / B$$

Hawk's macro `HALFTOP` transforms a world coordinate point to a screen coordinate point. A program was written that invoked this macro 100,000 times. Compiled without the optimizer, this program executed for 5.1 user-sec or 51 microsec/point. With the optimizer, this program executed for 4.5 usersec or 45 microsec/point.

When the window is zoomed a long way out, i.e., `viewportDimension` \ll `windowDimension`, there are many λ per pixel, and only a coarsely sampled image can be displayed. Since this image is not very pleasing and can be very costly to display, suppressing detail is very important as presented in the next sections.

When there are one or more pixels per λ , the number of pixels per λ may not be integral and this leads to asymmetries. Here's an example:

```
viewportWidth = 1000 pixels
windowWidth = 588 lambda
1.7 pixels/lambda
firstRectangle.left = 0 lambda
firstRectangle.right = 2 lambda
secondRectangle.left = firstRectangle.right = 2 lambda
```



```
secondRectangle.right = 4 lambda
firstRectangle.width = secondRectangle.width = 2 lambda
```

Transformation yields:

```
firstRectangle.left = 0 pixels
firstRectangle.right = Round(3.4) = 3 pixels
firstRectangle.width = 4 pixels
secondRectangle.left = firstRectangle.right = 3 pixels
secondRectangle.right = Round(6.8) = 7 pixels
firstRectangle.width = 5 pixels
```

Though in world coordinates the width of both rectangles is the same, their widths in screen coordinates differ by one. This is annoying, but because the shared edge in world coordinates is still common in screen coordinates, connectivity is preserved.

5.12. Ellipses

The control points in the last example may be interpreted slightly differently. If the x coordinate of the center of a circle is 2 lambda and the circle's radius is 2 lambda, then the x coordinate of the center is 3 pixels, the minimum x coordinate is 0 pixels, and the maximum x coordinate is 7 pixels. Thus, the perfect circle in world coordinates is *warped* in screen coordinates. This is annoying and strikingly visible, because a circle's main perceptual property is its symmetry. If the circle is displayed as an ellipse, symmetry will be preserved as well as connectivity. However, a line passing through the center of the circle in world coordinates will not pass through the center of the ellipse. The ellipse solution trades inter-shape symmetry for intra-shape symmetry. Though it is a highly subjective matter, schematic display experiments reveal that this is a good trade-off.

Display of ellipses is an interesting problem. The common solutions sweep theta in the parametric equation of an ellipse to yield points on the ellipse, use a DDA-class algorithm, or use a Bresenham-class algorithm. A package named `EI` was built whose data structure is a table of elliptical arcs sweeping through the first quadrant. Arcs with major and minor

axes ranging from one up to a compile-time constant of ten are tabulated. Ellipses or elliptical arcs that sweep between two angles that are each a multiple of 90 degrees are built up by symmetry operations.

The points on the ellipse must *not* be connected by lines drawn by a line drawing algorithm, because if an equilateral triangle is drawn with one side parallel to the x axis, and one of the remaining two sides is flipped sideways and overlaid over the other, my experience is that they will *not* match pixel for pixel. Plotting the trajectory insures perfect symmetry.

Other elliptical arcs are built up by constructing an ellipse, computing the point on the ellipse that is closest to the point corresponding to the starting angle, and traversing the ellipse, starting from this closest point, accumulating consecutive points until the point on the ellipse that is closest to the point corresponding to the finishing angle is reached. Clipping is very similar. The ellipse is traversed searching for visible-to-invisible and invisible-to-visible transitions that yield visible sub-arcs.

5.13. Bounding Boxes

There are Squid operations to fetch the graphics and "graphics+text" bounding boxes of an instance and of a view. Squid also propagates bounding box changes *up* the instance hierarchy so that the bounding box information is correct unless the propagation has been deferred by calling `SQDeferBBUpdates`.

Yet another example of derived shapes is the problem of displaying bounding boxes of instances rather than all of the shapes the instances contain. Two of the modes for a Hawk window are `bb1` and `bb*`. `bb1` displays the bounding box of each instance *in the view* displayed in the window as well as all shapes in the view. `bb*` displays the bounding box of all instances *in the hierarchy* whose source is the view displayed in the window as well

as all shapes in the view. In **bb***, the bounding box of an instance is displayed in a color that indicates the level of the hierarchy it is on. This makes a clear, spatial decomposition of the hierarchy.

5.14. Suppressing Detail

If a Hawk window is in **expand** mode, all shapes in the hierarchy whose source is the view displayed in the window are displayed. If the RISC I microprocessor floor plan is displayed in a 1000 pixels by 1000 pixels viewport, about 500,000 rectangles would be displayed at a scale of about 400 square microns per pixel. On a frame buffer that can draw 300 rectangles/sec, a redisplay would take over 27 minutes.

If the Hawk client, in error, commands Hawk to run a long redisplay, he may press an interrupt key and Hawk will cease. Hawk does display fewer derived shapes as the client zooms out more. First, the grid is suppressed, then names of terminals, then names of instances. Polygons and arcs are depicted by their bounding boxes when the client zooms out far enough. There is no sense scan-converting an arc that is on the order of one pixel in area.

It may be reasonable to have Hawk automatically suppress the display of shapes whose width and height are less than a certain number of pixels. A more interesting possibility is to partition all such shapes into equivalence classes. Each class is a set of shapes that are separated by no more than a certain number of pixels. Each class is shown as the bounding box of all shapes in the class. When a class is zoomed in on, it breaks apart into its component shapes.

5.15. Obscured Pins

When a circuit is designed, its terminals are given names that suggest their function. For example, a RAM cell may have a `busA` terminal on it. However, when the RAM cell is used in a register as a master, it is convenient to rename `busA` to a contextual name such as `IR <3>` and have the name `IR <3>` obscure the name `busA` when the register is displayed or plotted. As proposed in Chapter 2, it may be argued reasonably that this semantics should be prescribed by the data model. Instead, Hawk insures this in its redisplay algorithm. Before a pin `pin` at level `level` on layer `layer` is displayed, the area of layer `layer` that `pin` covers in the levels of instance hierarchy above `level` is searched for pins that obscure `pin`. `pin` is displayed only if there are no such pins. This can be interpreted as another example of derived shape.

CHAPTER 6

NEW CIRCUIT CAD TOOLS

6.1. Logic Design Tools

6.1.1. Schematic and Block Diagram Capture by Beaver

Traditionally, a schematic diagram or block diagram—henceforth abbreviated schematic— is drawn with a graphics editor by using the same commands that are used in drawing any two-dimensional line drawing. The schematic differs from any two-dimensional line drawing in that it must conform to a particular syntax so that a schematic extractor tool—henceforth abbreviated extractor—can read the schematic and produce the net list, or connectivity, represented by the schematic. A typical schematic syntax is as follows:

1. A shape functioning as a pin is associated with a terminal by adding a `terminalName` property to the shape's property list.
2. A shape functions as a frame if it is not a pin and there are no instances in the view the shape is contained in. Such a view would represent a schematic-symbol.
3. A shape is asserted to function as an interconnect if it connects two or more pins and there are instances in the view the shape is contained in. Such a view would represent a schematic.
4. Two terminals are connected if a pin associated with one terminal intersects a pin associated with the other terminal. Also, two terminals are connected if a pin associated with one terminal intersects an interconnect and a pin associated with the other terminal intersects the same interconnect.
5. If two interconnects intersect, the extractor will assume that the interconnects are not connected unless an object is present at the intersection to explicitly create a connection. Such an object is termed a *solder dot* and either a shape or instance of a solder dot master can function as one.

A useful command for editing a schematic is to select all control points that intersect a

selected instance—normally the control points will be associated with interconnects that are connected to terminals on the selected instance. Thus, when the selected instance is moved, the interconnects whose shape is controlled by the selected control points will move along with the selected instance.

However, the following problems arise when capturing schematics with the commands of the graphics editor for 2D line drawings. Sometimes the moved interconnects will result in a pleasing schematic, but sometimes the schematic will appear ghastly, unwanted connections will be created, and solder dots will have to be moved or deleted manually after the move. After an instance is deleted, often lines functioning as interconnects possess control points that no longer intersect pins, because the pins were deleted along with the instance. Such lines must be edited after the deletion. A terminal is termed *floating* if it is not connected to any other terminal. Commands that highlight all terminals connected to a selected terminal and that highlight floating terminals cannot be implemented without executing the extractor.

The schematic capture editor, Beaver, is a set of commands built within the framework that solves some of those problems. Beaver's schematic syntax follows:

1. A view named **body** that represents the schematic-symbol for a cell is drawn as a set of frames on the special layer **SYMBOL** and a set of pins on the special layer **CONNECT**. The frames function as obstacles for Beaver's maze router. The pins must lie on or outside the bounding box of the frames. Shapes on other layers are ignored, but the other layers may be useful for shapes representing documentation, terminal names, logic bubbles, etc.
2. A view named **schematic** that represents the schematic-drawing for a cell is drawn as a set of pins on the special layer **CONNECT** and a set of instances of **body** views. If the cell also has a **body** view, then every formal terminal in the **body** view must appear in the **schematic** view. A *global terminal* is a terminal whose name ends with an exclamation point. A global terminal is automatically associated with a global net whose name is the same as the terminal's name. Thus, global terminals do not have to be connected by interconnects.

Thus far, no commands specific to Beaver are required for schematic capture, except that the

selection tool permits the user to add and subtract terminals from the selected set as well as shapes and instances. Selecting a pin causes its associated terminal to be selected as well as all of the terminal's associated pins.

There are only four commands that manipulate the schematic view in the current window. At all times, the net list is simply the set of nets in the schematic view—extraction is unnecessary. First, the command **connect** deletes interconnects implementing any nets that the selected terminals may be participating in, merges the nets together into one net and deletes them, adds any selected terminals that are floating to the one net, and routes the one net. The routing algorithm will be described in the next paragraph. Second, the command **disconnect** deletes interconnects implementing any nets in which the selected terminals participate, inserts the selected terminals in a special floating net, and then re-routes the nets that were initially un-routed. Third, the command **dangling** inserts into the selected set any floating terminals. Fourth, the command **jazzNet** inserts into the selected set any terminals that are connected to terminals that were members of the selected set when the command was invoked.

The selection tool's commands that move and delete the selected set had to be modified to invoke the Beaver router to re-route affected nets. This could have been implemented as a Beaver Squid-demon, but was not. The routing algorithm is divided into two phases. First, for each net, a global routing phase calculates a minimum spanning tree for the net using Manhattan distance between the net's pins as the branch weight. Second, each minimum spanning tree branch is routed by a rule-based, pin-to-pin, maze router. Each rule is applied and generates one route that is tested against the existing objects in the schematic view. If a route intersects an obstacle frame implicitly defined by an instance of a body view or the route contains an interconnect that lies on top of an existing interconnect associated with a different net, then the route is rejected. Otherwise, a cost that measures the quality of the route is calculated that sums the length of the route and a penalty factor for the number of

interconnect crossings. If all routes are rejected, a possibly non-Manhattan straight line is drawn as an interconnect on the special layer **SOLDER** between the pins. Otherwise, the route with the minimum cost is chosen. As a post-processing step, interconnects functioning as solder dots are placed on the special layer **SOLDER** at any intersections of interconnects associated with the net. In many cases, a Steiner tree would yield a more pleasing route than a minimum spanning tree.

Because the pairs:

```
< SOLDER, interconnect >
< CONNECT, interconnect >
< CONNECT, pin >
< SYMBOL, frame >
```

are used, four different colors can be used to aid perception of schematics. As expected, global routing that is "more global" than one net at a time is necessary to obtain schematics with the minimum number of interconnect crossings. Highlighting failed routes by placing them on the layer **SOLDER** guides the user in moving instances apart to yield more routing area between instances.

6.2. Layout Design Tools

When designing custom integrated circuits, there are at least four design methods applied in the layout phase of design:

1. Detailed layout of views such as RAM cells, bonding pads, and large transistors.
2. Automatic layout by module generator as in PLAs and gate matrices.
3. Wire-less connection by careful placement as in bit-slice data paths and registers (abutment or overlap).
4. Connection by routing as in floor plans.

Each of these approaches will be referred to during the description of a number of layout tools in the remainder of this chapter.

6.2.1. Drawing Layouts

There are two types of paradigms for layout editing:

1. Shapes are painted. As shapes are created, they are *merged* with existing shapes. The painting analogy arises because after painting with a good paint and brush, the exact strokes of the brush are not visible and neither are the originally created shapes except by chance. Shape deletion is modeled as painting with a background paint. Shapes can only be modified by creating and deleting shapes. Thus, creation, deletion, and modification of shapes are performed by painting [49, 50].
2. Objects are manipulated. When a shape is created, it is added to the database without altering the structure of existing shapes or the added shape. Deletion and shape modification are performed by selecting control points, edges, or whole shapes and moving, replicating, and removing them. Moving a control point or edge is sometimes termed *stretching*.

In the paint paradigm, since the relationship between what specific shapes were drawn and the area they cover is not maintained, a *drawn set* of rectangles on a layer can be represented by a smaller, perhaps minimal, set of rectangles in which any two rectangles that intersect must abut. In the object paradigm, the drawn and minimal sets will only be identical if special care is taken by the drawer or by chance. If the drawn and minimal sets are not the same, there will be several problems. Redispays will be longer than necessary, because there are more rectangles to display. If rectangles are outlined, the outlines may be confusing. Yet, if rectangles are not outlined, selecting all objects that contain a single point becomes a game of chance, because the individual objects are not differentiable.

The selection tool I have implemented uses an object paradigm. Recall from Chapter 5 that selection is implemented outside of the Hawk kernel.

6.2.2. Array

In harmony with the viewpoint on geometric regularity in Chapter 2, a tool called array has been constructed whose input is a command line that describes the array the tool is to build and whose output is a physical view representing the array. To make a 4 bit register

named **NibbleReg** with odd bits mirrored out of a memory cell named **BitReg**, for example a typical command line is:

```
array -row -mirror -width 6 -pitch 5 -# 4 NibbleReg -homo BitReg
```

Figure 6.1 illustrates the construction.

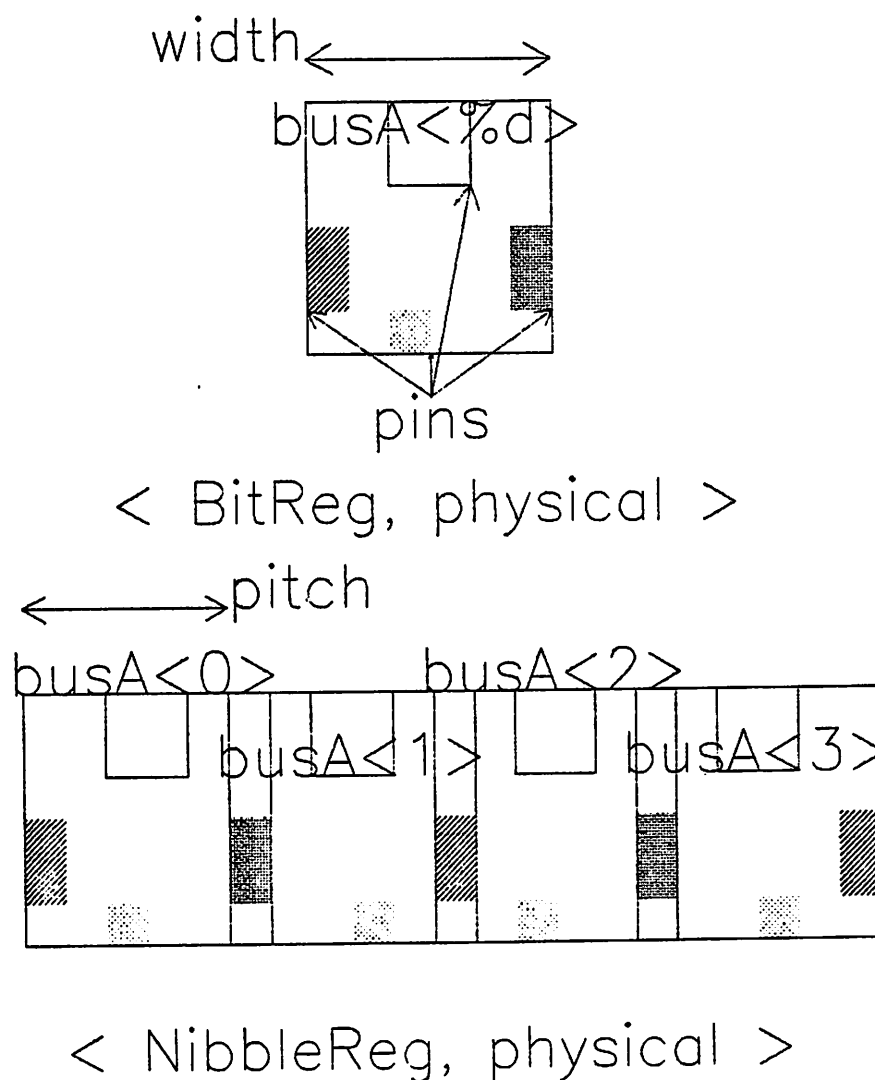


Figure 6.1: NibbleReg is a homogeneous row vector of BitRegs.

-row means that the array is a row vector—each instance is placed side by side. **-mirror**

means that odd bits are flipped sideways. **BitReg** is 6 lambda wide and has a pitch of 5 lambda. The value of the **-#** switch is the number of bits. **-homo** means that the array is homogeneous—all of the elements are identical. Each terminal of **BitReg** whose name has the form:

terminalName%d

as in **busA <%d>** is termed a *subscripted terminal*. For each integer *i* between 0 and 3, *i* is substituted for %d in all subscripted terminals— formal terminals are created in the physical view of **NibbleReg** with the resulting names. The pins associated with subscripted actual terminals on the *i*th instance of **BitReg** are copied into the physical view of **ByteReg** properly transformed. When the physical view of **NibbleReg** is displayed, recall from Chapter 5 that the names of the terminals associated with the pins created by **Array** will obscure the names of the subscripted terminals associated with the pins that **Array** copied. Effectively, %d plays the role of a bit number, but often a word number is desired. To solve this problem, %d must be generalized to %subscriptName. Thus, a physical view might contain formal terminals named **busA%bit** and **enable%word**. The value of **Array**'s new switch **-subscriptName** is a substring that follows % in a formal terminal name.

The tool can build a row or column vector composed of simple instances of element views. Matrices must be built as rows of columns or columns of rows, but this is usually quite natural as exemplified by a register file composed of registers with each register composed of memory cells. The tool aligns adjacent elements so that they intersect properly and mirrors every other element if desired so that the silicon area occupied by power and ground pins is shared.

At this time, **Array** is simply a drawing tool. For a layout analysis tool to exploit the regularity of a physical view which represents an array, **Array** should also produce a view which contains additional implied connectivity information and a summary of the array.

Not all arrays are homogeneous. The following command line constructs the physical view of the byte insert/extract stage of a CMOS microprocessor's data path:

```
array -mirror -height 124 -pitch 117 "-#" 32 bfloor -subscript baby
```

Each of the 32 bits is different, but `-subscript` means they are named `baby0/symbolic`, `baby1/symbolic`, ..., `baby31/symbolic`. The physical view will be stored in `bfloor/physical`. Unless a layout analysis tool such as the next section's Frame can prove that the `babyi` have the same perimeter characteristics, a layout analysis tool will be unable to exploit `-subscript` regularity.

The following command line builds the physical view of the ALU stage of a CMOS microprocessor's data path:

```
array -pitch 117 "-#" 32
  afloor #Name of the cell.
  -random #Random pattern among the elements.
  alufirst aluodd alueven alulast
  alufirst aluodd alueven alulast
  alufirst aluodd alueven alulast
  alufirst aluodd alueven alulast
  alufirst aluodd alueven alulast
  alufirst aluodd alueven alulast
  alufirst aluodd alueven alulast
  alufirst aluodd alu30 alu31
```

Because the bits are already mirrored, `-height` is unnecessary. If the original array is:

$$E_1 E_2 \cdots E_n$$

or equivalently:

$$P_1 P_2 \cdots P_{n-1}$$

where P_i is:

$$E_i E_{i+1}$$

then the *minimal array* for the original array is the array that contains only the unique P_i .

For the ALU, the minimal array is:

```
alufirst aluodd alueven alulast
alufirst aluodd alu30 alu31
```

If the minimal array is processed by the layout analysis tools, then these tools must generalize their output slightly to assert that they have processed the entire original array.

If the array's pattern is:

```
even odd even odd even odd ...
```

the command line's form is:

```
array -mirror -pitch # -height # theArray -evenodd even odd
```

If **odd** is already mirrored, **-mirror** and **-height** are unnecessary. Clearly, layout analysis tools can exploit the regularity here.

6.2.3. Protection Frames

For a long time, schematic-symbols have been used widely to suppress detail in schematic-diagrams and block diagrams. When an instance of a schematic-symbol is created, an instance of the schematic-symbol's associated schematic-diagram is created implicitly as well. By assigning the proper values to the properties and actual parameters of the schematic-symbol, as well as connecting to the schematic-symbol's actual terminals, the behavior of the circuit represented by the schematic-symbol's associated schematic-diagram is determined completely. Thus, the schematic-symbol is an abstracted view of its associated schematic-diagram view.

In the remainder of this section, the use of abstract views for layouts is described. The goals are to speed layout analysis, speed layout drawing, decrease space requirements, and decrease screen clutter. The small amount of past research in this area is reviewed and then new results are presented. All examples will be drawn from an nMOS process [8] that is used for low-density custom circuits. The process has the following characteristics:

- Depletion and enhancement, self-aligned, polysilicon-gate transistors.
- A minimum channel area of 2λ by 2λ .
- A minimum line width of 2λ .
- External thick-oxide contacts between polysilicon and metal, and diffusion and metal.
- External gate-oxide contacts between polysilicon and diffusion used mostly for connecting gates and sources when building depletion load devices. Such contacts are sometimes termed *buried*.
- One level of metal with a minimum line width of 3λ and a minimum separation of 3λ .

Figure 6.2 is a legend depicting, for each mask, the mask's name, the mask's function, and stipple pattern that shapes on the mask are drawn in.

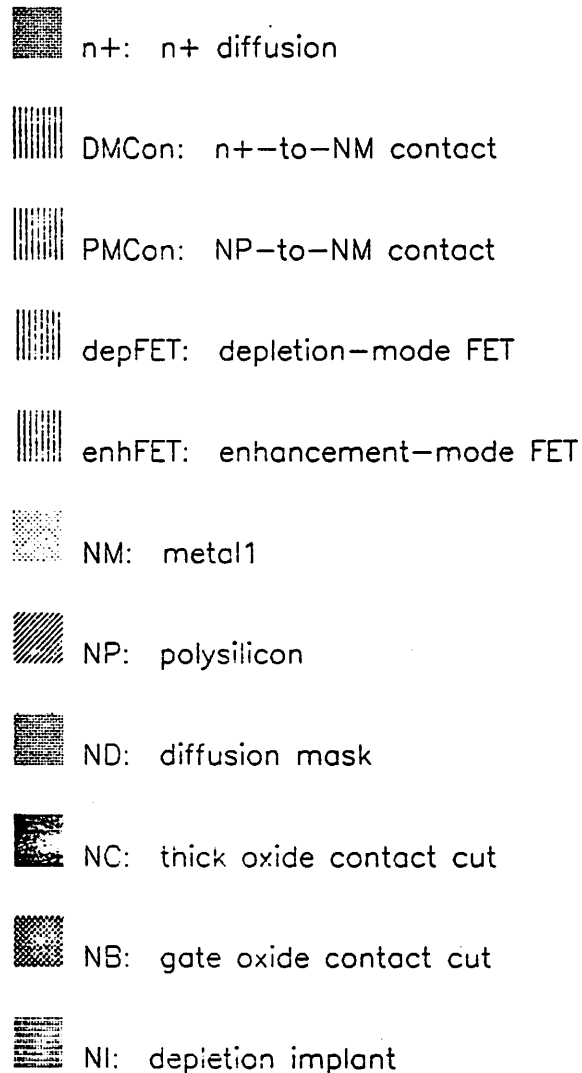


Figure 6.2: Legend for nMOS process.

At the low end of the abstraction spectrum is the *mask view* of a layout. All mask modification has been performed so that each layer in the mask view corresponds to a *blow-back* [8], *plate*, or *mask*.

A view that is somewhat more abstract than the mask view, is the *layout view* which con-

sists of layers that are easily transformed to masks by a mask operation tool. The layout view of an nMOS, depletion-load inverter is illustrated in Figure 6.3.

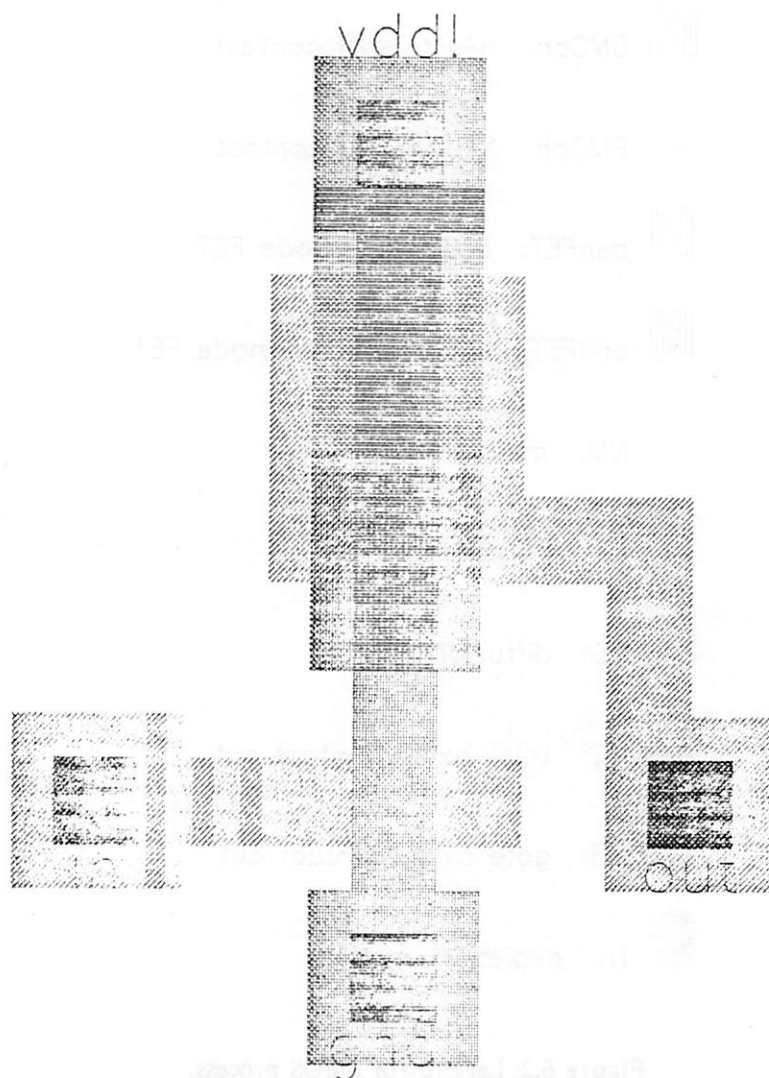


Figure 6.3: Layout view of a depletion-load inverter.

This inverter will be abstracted in several ways in the remainder of this section. Each labeled rectangle is a pin—the label the pin contains is the name of the terminal the pin is associated with. Neglecting mask operations that correct for the effect of wafer processing

on nMOSFET channel dimensions and neglecting pins, the layout view would be the same as the mask view. The large majority of layout processing and display tools process the layout view.

When a chip is designed using a gate array or standard cell design method, the chip's floor plan is regular. Regularity makes possible a high level of abstraction which in turn simplifies layout processing.

A gate array chip is designed by customizing a standard floor plan by connecting gates and vias by routing on conductor layers. Because the placement of the floor plan's gates and vias is fixed, the floor plan can be abstracted as a set of routing channels that interconnects are routed in. Except for the conductor masks, the mask view is computed once and does not change from chip to chip. After routing a chip, the mask view of the routing masks for the chip can be computed trivially and is correct by construction.

A standard cell chip is harder to abstract than a gate array chip, because the placement of standard cells is not fixed. However, the floor plan and standard cells are regular. A standard cell can be abstracted by its clock and supply pins which abut matching pins on adjacent standard cells and its signal pins which lie on the bottom and top edges of it. A signal pin can be connected to by an interconnect, but not by another signal pin, because standard cells are placed in separated rows. A floor plan can be abstracted by routing channels between rows of standard cells. Because the standard cells are carefully designed to abut correctly and are precisely characterized once and re-used in each chip, as in the gate array case, the mask view for a particular chip is easily computed and correct by construction.

Currently, custom chips are designed with tools that process the layout view and thus without the aid of abstraction. Several programs [51, 52] have been built that exploit regularity to simplify processing, but they process the layout view. Several researchers [36, 53-57] have implemented abstractions to simplify processing that are presented next.

At the high end of the abstraction spectrum is the *bounding box view* as exemplified by the abstraction of the inverter from Figure 6.3 in Figure 6.4.

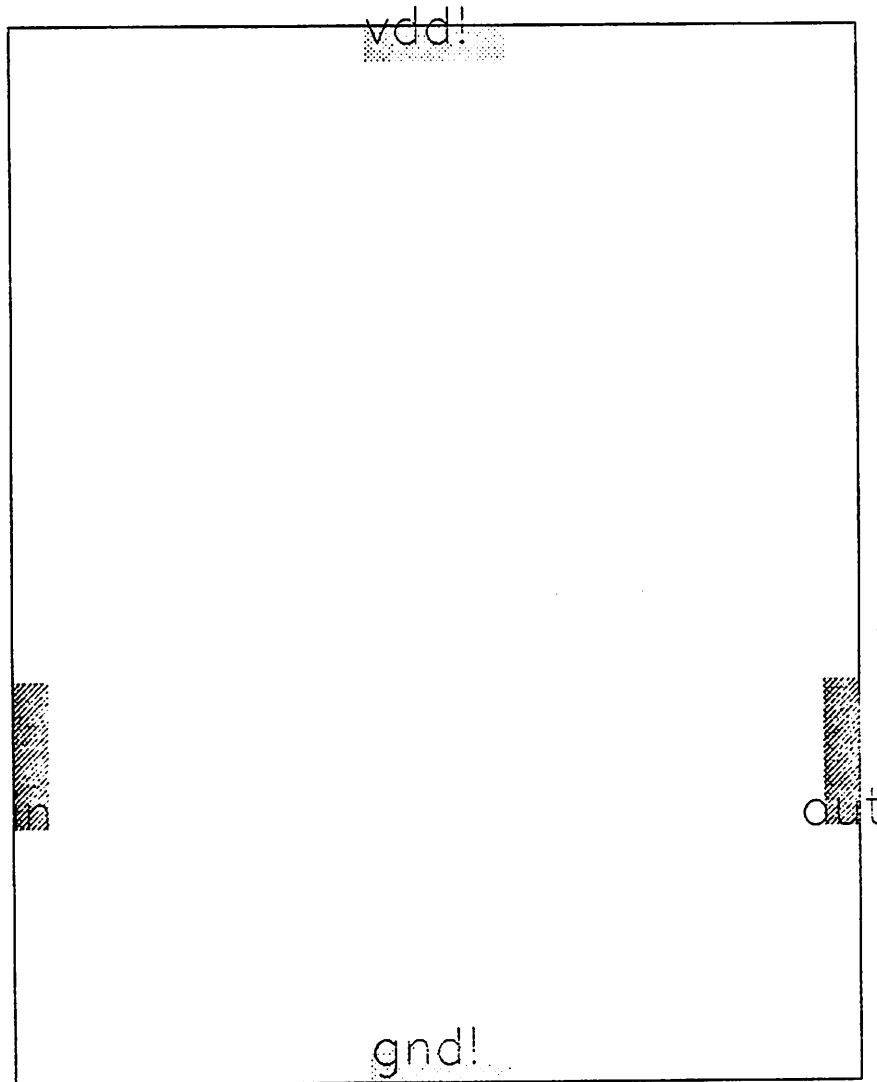


Figure 6.4: Bounding box view of a depletion-load inverter.

The bounding box view represents the two pieces of information that every layout abstraction must: pins to be connected to and obstacles to stay away from termed *protection frames*. The bounding box approximates the layout view by covering at least all of the

shapes of the layout view. The internal and border details of the layout view are lost. The algorithm for computing the bounding box view from a layout view in which the designer has drawn pins follows:

1. Merge all shapes.
2. Grow the shapes resulting from the merge by the largest distance **MAXRULE** that two shapes *must* be separated by.
3. Compute the bounding box of the shapes resulting from the growth.
4. Make this bounding box a rectangle on the layer named **BB**. (In CMOS, **MAXRULE** is usually equal to the greatest well-to-well spacing. In nMOS, **MAXRULE** is usually equal to the greatest metal-to-metal spacing. In bipolar, **MAXRULE** is usually equal to well-to-base spacing or to the greatest metal-to-metal spacing. Because of the growth, interconnects and other bounding boxes can *touch* a bounding box, but *cannot overlap* the bounding box if layout rules are to be obeyed. The bounding box abstraction is quite conservative, because two bounding boxes placed side by side will be further apart than they need to be by at least the amount of growth. This abstraction does have the appeal that interconnects do not have to be any particular distance away from bounding boxes.)
5. *Stretch* the pins by river-routing the pins to the edges of the bounding box. (For each pin, disregard its route except for that which is colinear with the bounding box edge and let that be the pin in the bounding view.)
6. For each pin, calculate the largest current that can flow through the pin. (Thus, the width of any interconnects that are routed to the pin can be adjusted so that the interconnects do not break up due to a worst-case current flow.)

This algorithm can be automated easily by invoking a river routing tool and a mask operation tool. Current calculation is harder and thus may be left to the designer. This algorithm can be applied to a view that contains only instances of bounding box views, interconnects, and pins so that this abstraction is hierarchical. A slight variant of this abstraction is used in the BBL placement and routing tool [58]. In this variation, the protection frame is a Manhattan polygon and worst-case current flow is indicated by pin width. Instances of bounding box views cannot be routed over. In general, an abstraction with a single protection frame on a special layer cannot be routed over except by pre-planned jumpers. This is a disadvantage in some design methods however it also has several advantages. First, there is no problem in extracting node-to-node, cross, mutual, or coupling capacitance. For multi-

layer metal processes, this is usually not a problem anyway. Second, each layout view can be prepared for pattern generation independently at least in the domain of optical pattern generators. If interconnects are grown during preparation of the mask view, care must be taken to shrink them back so they do not overlap grown pins.

The bounding box protection frame is so coarse that an unacceptable amount of chip area may be traded for the processing efficiency of the bounding box view. A view that represents the border of the layout view in more detail in order to enable greater area utilization is the *doughnut view* in Figure 6.5.

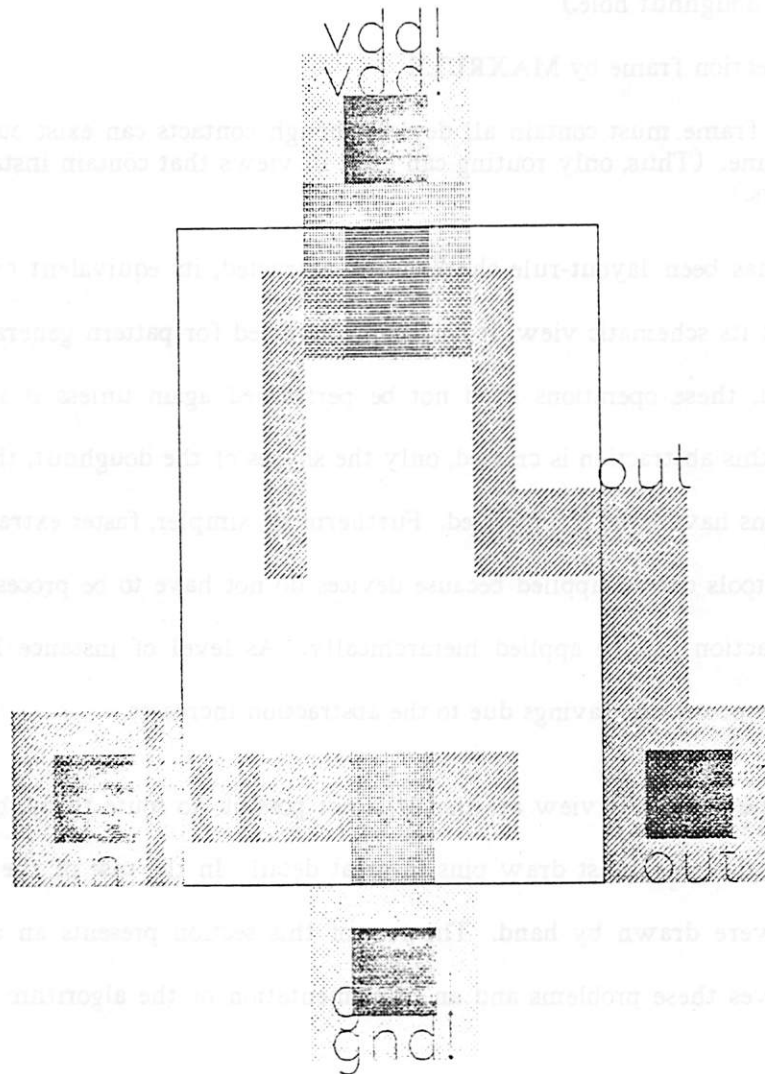


Figure 6.5: Doughnut view of a depletion-load inverter.

The algorithm to compute this abstraction is:

1. Copy the layout view to the doughnut view **doughnut**. The rest of the operations are performed on **doughnut**.
2. Draw a protection frame polygon on the layer named **BB** that encloses all of the shapes that are not pins to be shared when an instance of another doughnut view overlaps an instance of **doughnut**.
3. Shrink the protection frame by **MAXRULE**.

4. Clip to the outside of the shrunken protection frame all of the shapes contained in **doughnut**. (This forms a doughnut-shaped abstraction, because the clipping operation forms a doughnut hole.)
5. Grow the protection frame by **MAXRULE**.
6. The protection frame must contain all devices though contacts can exist outside the protection frame. (Thus, only routing can exist in views that contain instances of doughnut views.)

Once a layout view has been layout-rule-checked and extracted, its equivalent circuit has been compared against its schematic view, it has been converted for pattern generation, and it has been abstracted, these operations need not be performed again unless it is altered. When an instance of this abstraction is created, only the shapes of the doughnut, the protection frame, and the pins have to be re-processed. Furthermore, simpler, faster extraction and layout rule checking tools can be applied because devices do not have to be processed. The doughnut view abstraction can be applied hierarchically. As level of instance hierarchy increases, typically the processing savings due to the abstraction increases.

Two problems with the doughnut view are that it is not possible to route through the protection frame and the designer must draw pins in great detail. In the case of the inverter, 13 rectangular pins were drawn by hand. The rest of this section presents an algorithm called **Frame** that solves these problems and an implementation of the algorithm which is also called **Frame**.

First, it is assumed that the designer has drawn pins in each layout view. However, he does not need to draw the pins in their entirety— in Figure 6.3, the **vdd!** pins at the top can be drawn as a single vector on the metal layer instead of having to draw rectangles on the polysilicon, metal, and thick oxide contact cut layers.

The protection frame algorithm implemented in the **Frame** program is used to construct all protection frames described here and is as follows: First, layers are derived from the layout view that represent devices, contacts, and interconnects. The section named **FRAME** in the

file named `.cadrc` contains the rules for deriving these layers. The following excerpt from the `FRAME` section used to build the following examples is:

```

begin FRAME
# "Rules file" for Mead&Conway-MOSIS nMOS process with buried contacts
#and no butting contacts.
#NOT layer1 layer2 means layer1 = NOT layer2.
  NOT !P NP
  NOT !D ND
  NOT !C NC
  NOT !I NI
  NOT !B NB
  NOT !M NM
#AND layer1 layer2 layer3 means layer1 = layer2 AND layer3.
  AND n+ !P ND
  AND PD NP ND
  AND PD!B PD !B
#Must differentiate between the two FET modes so that enh FETs are
#not implanted by accident. depFET accounts for ND that isn't n+.
  AND enhFET PD!B !I
  AND depFET PD!B NI
  AND MC NC NM
#Must differentiate between the two thick oxide cuts or else when routing
#to a "contact pin" on NP (n+), we have to check if n+ (NP) is present.
#All NC is accounted for by PMCon and DMCon.
  AND PMCon MC NP
  AND DMCon MC ND
#CONNECTS layer1 layer2 means when a shape on layer1 intersects a shape
#on layer2, the two shapes are connected.
  CONNECTS NM NM
  CONNECTS NP NP
  CONNECTS n+ n+
  CONNECTS NM PMCon
  CONNECTS PMCon NP
  CONNECTS NM DMCon
  CONNECTS DMCon n+
  CONNECTS NP NB
  CONNECTS NB n+
#FRAME layer1 %d means to compute shapes functioning as protection frames
#on layer1 by growing shapes on layer1 that are not pins by %d,
#ORing the grown shapes, and shrinking the ORed shapes by %d.
#This operation closes up gaps between shapes on layer1 that
#cannot be used for routing on layer1.
#The choice of %d that yields the most freedom in routing is:
# %d = .5 * minLineWidthOnLayer1 + minSeparationBetweenLayer1Shapes - .5
#The choice of %d that forbids routing over a symbolic view on layer1 is:
# %d = INF
  FRAME enhFET INF
  FRAME depFET INF
  FRAME DMCon INF
  FRAME PMCon INF
  FRAME n+ INF

```

```

FRAME NB INF
FRAME NI INF
FRAME NP INF
FRAME NM INF
end

```

Consider derivation of the layer named **enhFET**. An enhancement-mode FET channel is formed everywhere polysilicon intersects diffusion and there are no gate oxide contact or depletion-mode implant shapes present. This can be expressed by the notation:

```
enhFET = NP AND ND AND (NOT NB) AND (NOT NI)
```

and as shown above:

```

NOT ! NI
NOT !B NB
AND PD NP ND
AND PD!B PD !B
AND enhFET PD!B !

```

Second, each pin that the designer drew is *extended* or *extracted* using the recursive touching geometric operation described in Chapter 2 and each line of the form:

```
CONNECTS layer1 layer2
```

in the section named **FRAME**. Considering again the **vdd!** pin on the metal layer in Figure 6.3, the lines:

```

CONNECTS NM DMCon
CONNECTS DMCon n+

```

enable the recursive touching geometric operation to extend the single metal pin through the thick oxide cut rectangle through the diffusion rectangles to the drain of the depletion-mode FET. Distinguishing between diffusion rectangles on the **enhFET**, **depFET**, and **n+** derived layers keeps the recursive touching geometric operation from extending the single metal pin through the depletion-mode FET's channel. Thus, this phase of the algorithm is precisely what a node extraction algorithm would perform.

The maximum width of the drawn and extracted pins are denoted by **maxPinWidth**. In

practice, the pins with the maximum width are always supply pins except in the case of wells and pins of huge transistors. Such pins must not contribute to `maxPinWidth`. Non-supply pins are easily neglected, because their associated terminals do not have names that end with an exclamation point. Well pins are easily neglected, because the `FRAME` section includes lines of the form:

```
SUPPLIED_BY CW GND!
```

`CW` denotes a CMOS p-well that must be grounded. Huge transistor pins are not so easily neglected. They have not been encountered by the program `Frame` as of yet so though this problem must be solved, its solution has been put off. Perhaps the best solution is to add a line in the `FRAME` section whose form is:

```
SUPPLY layer1
```

`layer1` is a layer that supplies are routed on—most often the highest level of metal. Only pins on layers that route supplies contribute to `maxPinWidth`.

Fortunately, the `SUPPLIED_BY` construct serves another purpose. After this phase of the algorithm completes, any shapes on supplied-by layers that are not pins associated with the supplying terminal are highlighted by shapes on a diagnostic layer. This is very useful in CMOS processes. In CMOS processes, it would also be useful to check that the wells are connected to their supplies every `wellSupplyingContactSeparation` square lambda. After Phase 3, this could be performed in the follow way:

1. If there is not at least one well-supplying contact in a well, the `SUPPLIED_BY` check will produce a diagnostic.
2. No edge that is part of the perimeter of a well is separated from a well-supplying contact in the well by more than `wellSupplyingContactSeparation`.
3. Each well-supplying contact in a well must have another well-supplying contact in the same well within `wellSupplyingContactSeparation` lambda of it.

Let `maxHW` denote the maximum of the Height and Width of the view being abstracted.

Third, for each layer, the shapes are grown by `maxHW`, merged, and shrunk by `maxHW-maxPinWidth` in order to construct a protection frame. This frame will *suppress the detail* inside the view being abstracted yet enable the pins near the view's perimeter to remain outside of the frame so they can be connected to. This phase can be skipped entirely as will be clear shortly.

Fourth, for each layer, the shapes that are *not pins* are grown by `growAmount`, merged, and shrunk by `growAmount` in order to construct a protection frame. `growAmount` is given in the `FRAME` section by lines of the form:

```
FRAME layer growAmount
```

Thus, the line:

```
FRAME NM INF
```

means to grow and shrink by `maxHW`. When `growAmount` is equal to `maxHW`, the resulting protection frame is a bounding, Manhattan polygon.

Fifth, for each layer, all pins and pieces of pins that are contained entirely inside a protection frame are removed from the abstract view, because they cannot be used in routing.

In summary, the algorithm is:

1. Layers are derived from the layout view that represent devices, contacts, and interconnects.
2. Each pin is *extracted*.
3. For each layer, the shapes are grown by `maxHW`, merged, and shrunk by `maxHW-maxPinWidth` in order to construct a protection frame.
4. For each layer, the shapes that are *not pins* are grown by `growAmount`, merged, and shrunk by `growAmount` in order to construct a protection frame.
5. For each layer, all pins and pieces of pins that are contained entirely inside a protection frame are removed from the abstract view.

This algorithm is coded in 528 lines of C.

The results of applying this algorithm to the layout view of the inverter in Figure 6.3 are Figure 6.6 and Figure 6.7. No metal or polysilicon protection frames were computed, because the inverter occupies such a small amount of area. To produce these frames from the original layout data required less than 5 CPU seconds on a VAX-11/780.

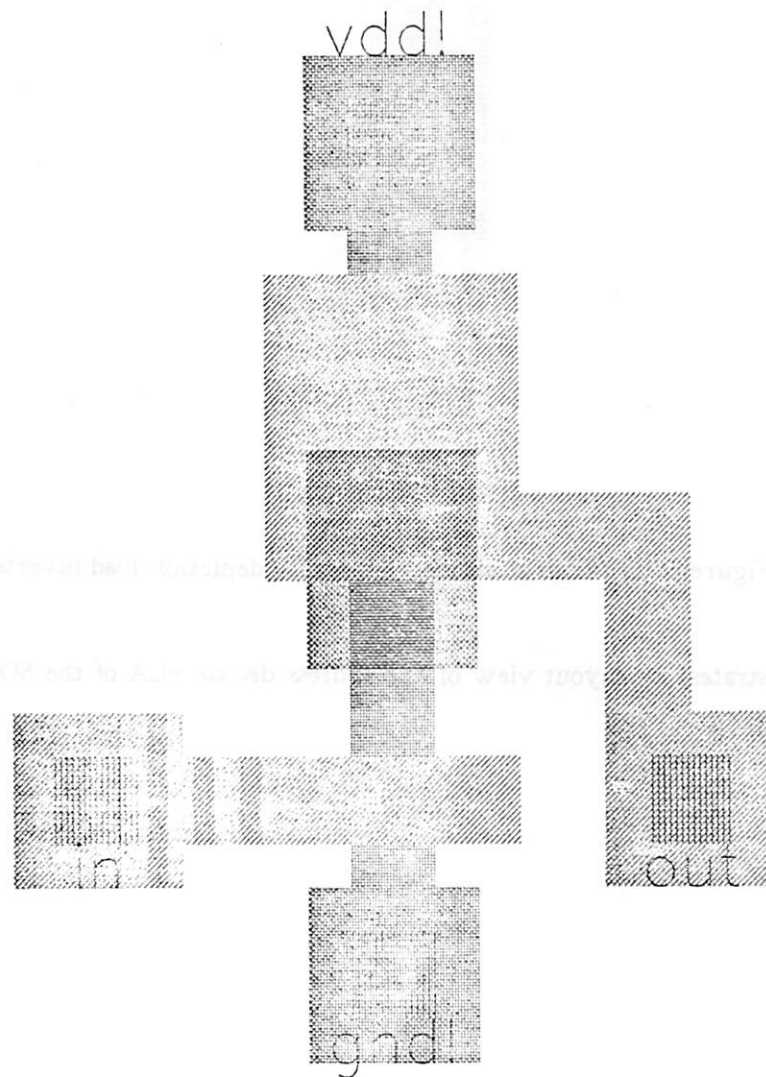


Figure 6.6: Pins of abstract view of a depletion-load inverter.

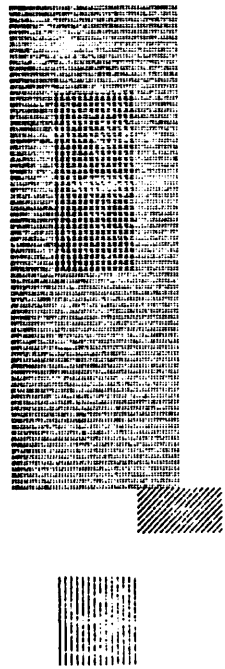


Figure 6.7: Frames of abstract view of a depletion-load inverter.

Figure 6.8 illustrates the layout view of the address decode PLA of the SOAR microprocessor.

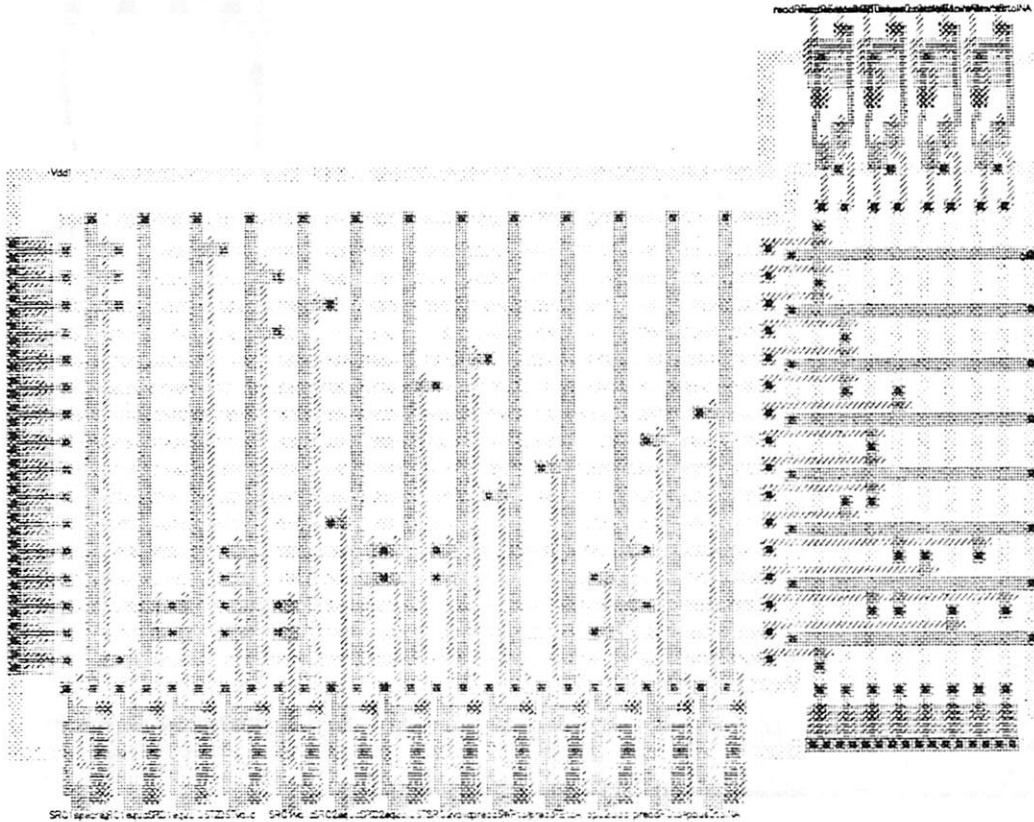


Figure 6.8: Layout view of address decode PLA.

Figure 6.9 and Figure 6.10 illustrate the frame abstraction of the layout view.

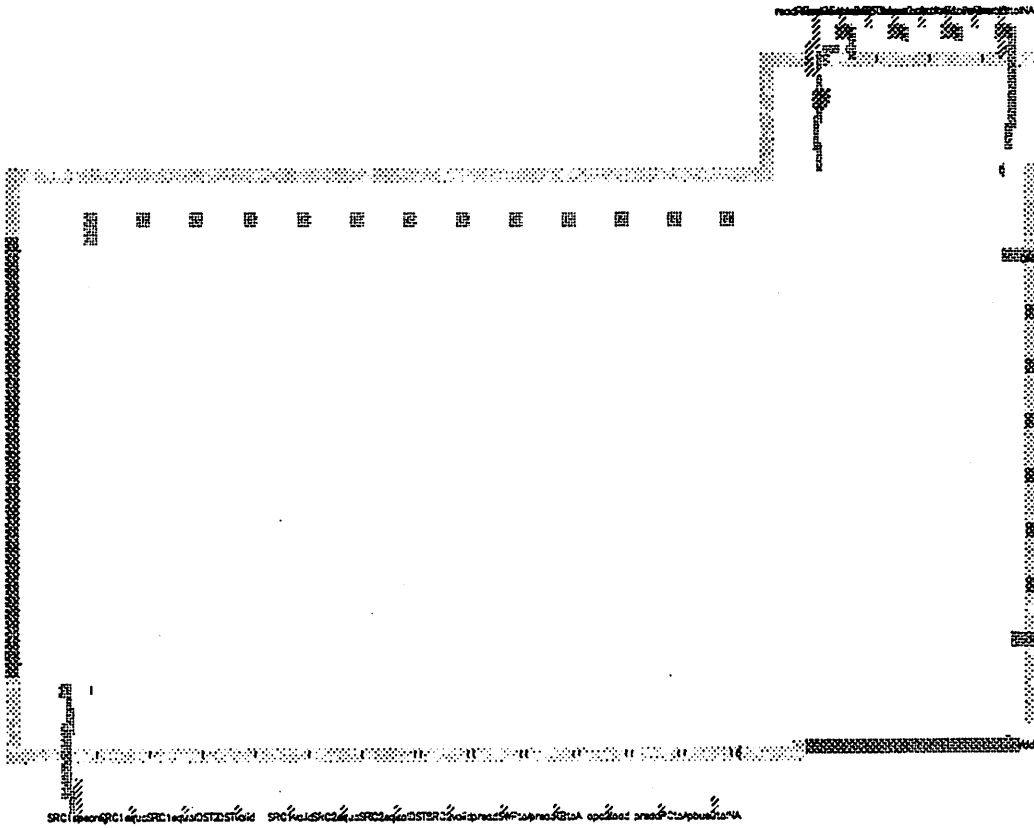


Figure 6.9: Pins of abstract view of address decode PLA.

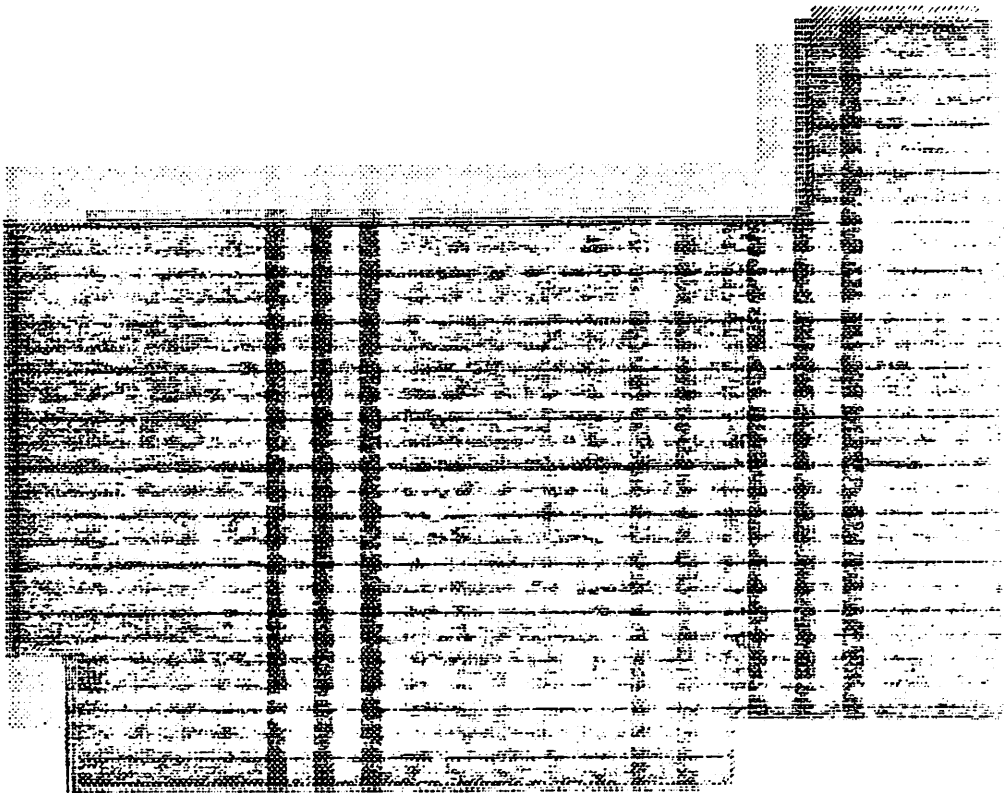


Figure 6.10: Frames of abstract view of address decode PLA.

Clearly, most of the area is occupied by protection frames.

Figure 6.11 and Figure 6.12 illustrate a view containing instances of the abstract view of Figure 6.3.

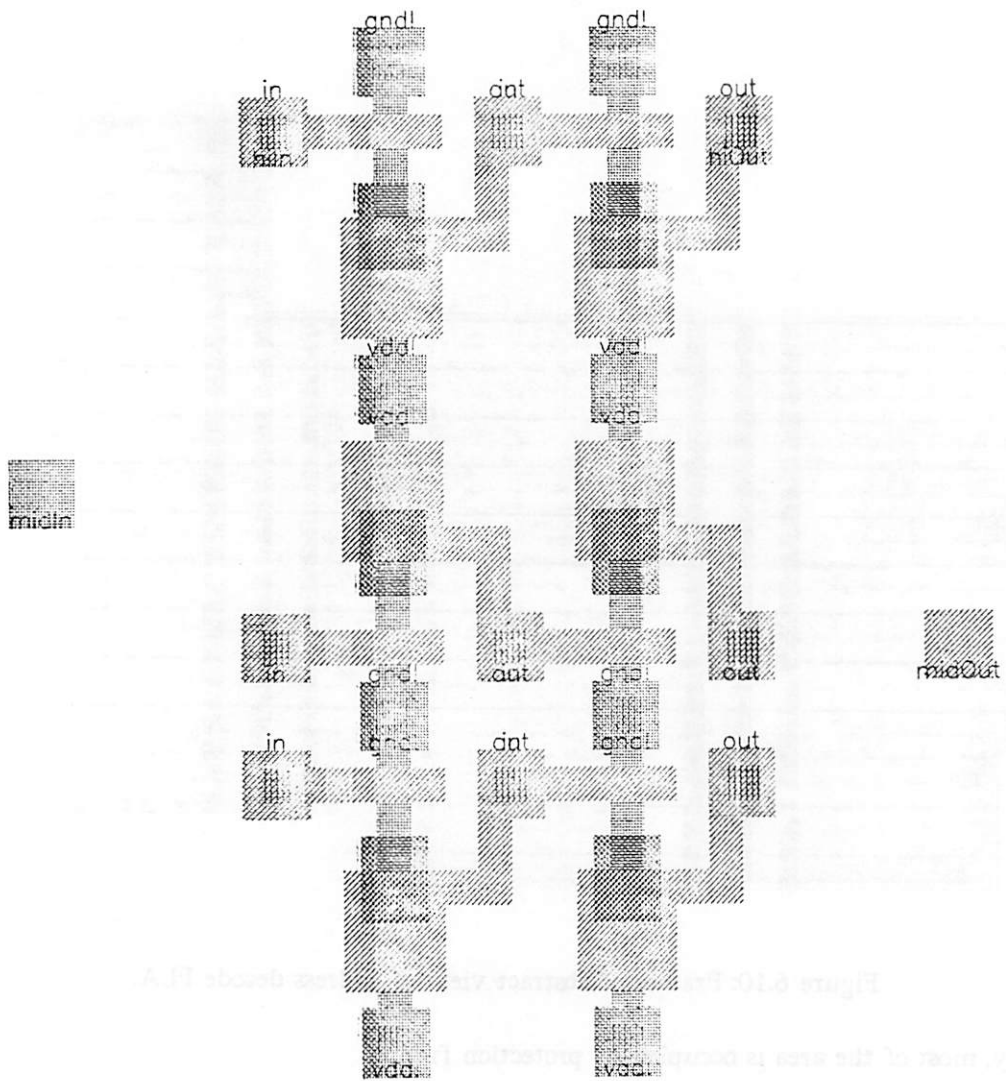


Figure 6.11: Pins of array.

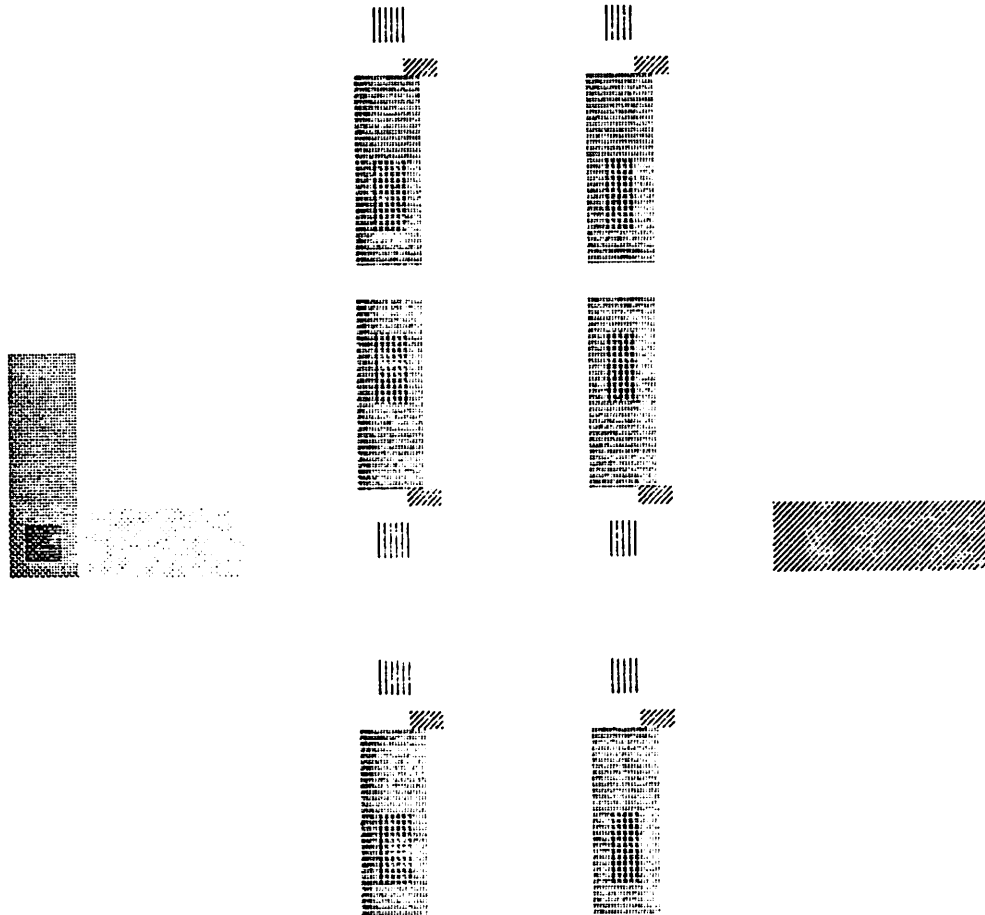


Figure 6.12: Frames and active area of array.

This view is basically a 3 by 2 array of inverters connected into a 3 by 1 array of non-inverting buffers. All possible examples of pin sharing are present— out to in, vdd!, and gnd!. In the top row of the array, the in and out pins have been renamed topIn and topOut respectively by placing metal pins on top of the pins on the top instances. In the case of topIn, topIn is at a higher level in the instance hierarchy than is in. During Phase 2, when the algorithm encounters two connected pins implementing different terminals, the algorithm will always coerce the lower level pin to the higher level pin. Global pins are considered to be at the imaginary Level 0—the highest level of the instance hierarchy. In

the middle row of the array, the pins `midIn` and `midOut` have been connected to `in` and `out` via interconnects and contacts. Phase 2 will recognize this. In the bottom row of the array, no pins have been connected to `in` and `out`. Phase 2 will *qualify* pins `in` and `out` by the names of the instances they are associated with. This qualification avoids potential terminal name clashes that can arise when multiple instances of the same view are present. Figure 6.13 and Figure 6.14 are the array's abstraction.

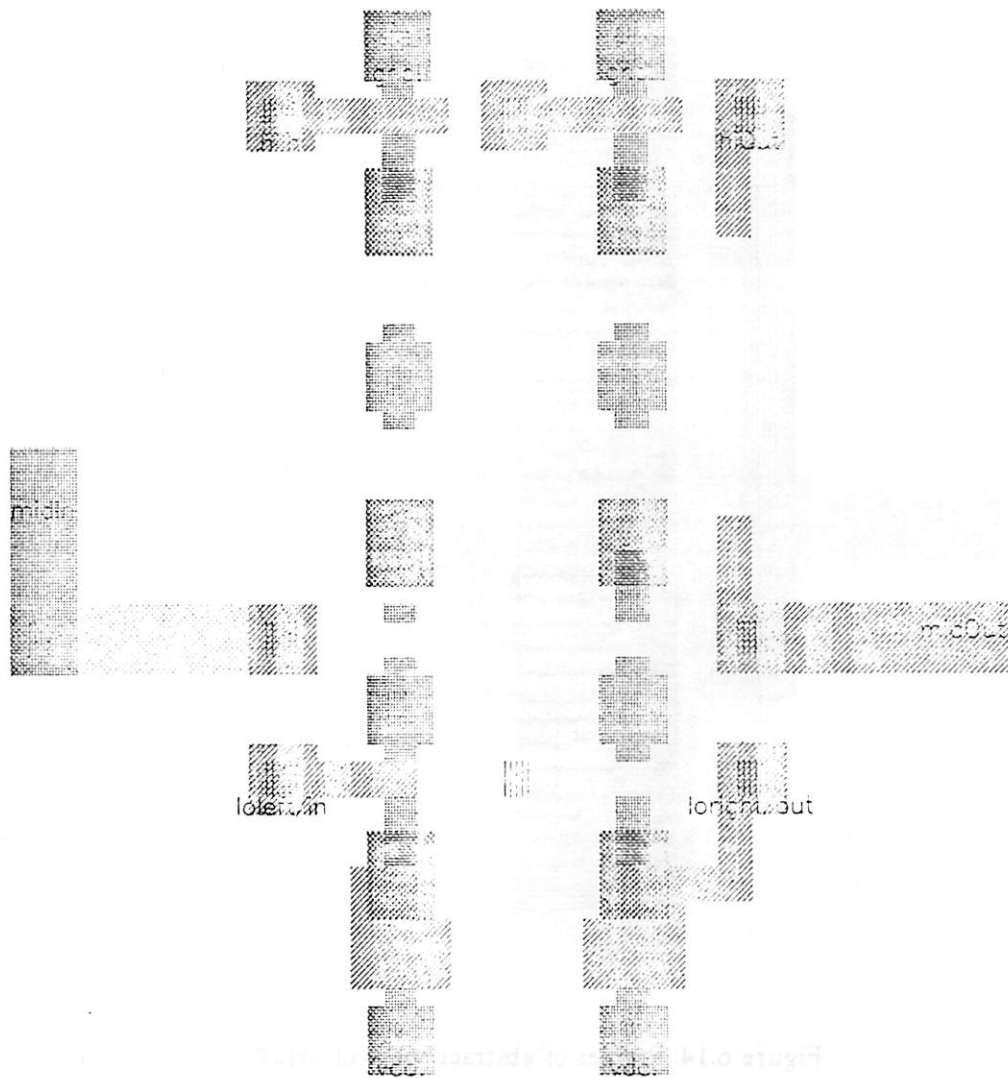


Figure 6.13: Pins of abstract view of array.

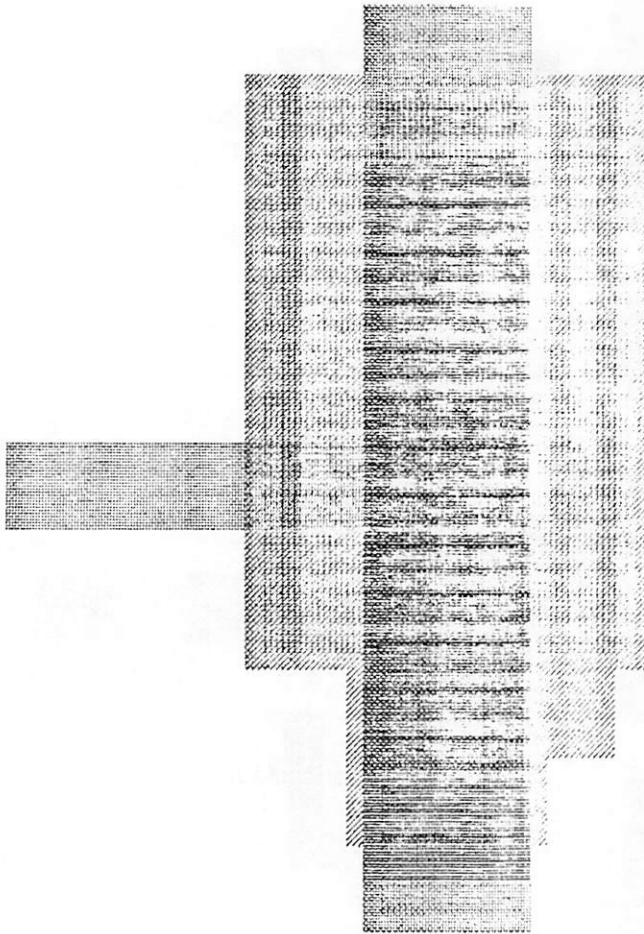


Figure 6.14: Frames of abstract view of array.

Metal and polysilicon protection frames cover the core of the array. All input and output pins are associated with different terminals, because Phase 2 has performed correctly.

If Phase 3 is skipped and the grow amount for conductor layers is:

$$\frac{\text{minLineWidthOnLayer}}{2} + \text{minSeparationBetweenLayerShapes} - \frac{1}{2}$$

then gaps that cannot be used for routing will be closed, but the protection frames will be *sets of disjoint* Manhattan polygons in general. Thus, these abstractions can be routed

through *without pre-planned* jumpers. When Phase 3 is skipped and the grow amounts are equal to small positive integers, term this algorithm the *gate array variant*. Otherwise, term this algorithm the *macro-cell variant*.

This abstraction is computed by the tool Frame available from Hawk and is used in the process-independent compaction tool Python [56]. By convention, the view to be abstracted is named **physical** and the abstracted view is named **symbolic**. The following measurements quantify the performance of Frame and how much less space is required to store symbolic views than physical views:

Circuit	Execution Time	# of Shapes	
		Physical View	Symbolic View
1-bit complementer	206	370	149
32-bit complementer	NA	$370 \times 32 = 11,840$	1095
PLA	126	3979	239
inverter	5	28	25
inverter array	18	$28 \times 6 = 168$	100

The mask operation package *fa* and the Squid package are used to implement Frame. Presently, *fa* does not have the ability to perform the recursive touching geometric operation though it does perform the scanning geometric operation. Thus, Squid performs the recursive touching geometric operation.

Consider the PLA in the above table. The approximately 4000 shapes that are framed is a misleading number, because the PLA is computed by a module generator that does not merge the shapes. When the shapes are merged by the Caesar graphics editor, the number of shapes decreases to 1100. Thus, Frame reduces geometric complexity by a factor of 5 rather than an order of magnitude. The pin extraction phase of the algorithm lasts for 6 seconds when an OSL data structure is used to index shapes and lasts for 18 seconds when a linear list is used to index shapes. With an OSL data structure, 30,000 shapes were searched instead of the 300,000 that exhaustive search traversed. Thus, the 10-fold search pruning of the OSL data structure compensates for the slower raw speed of the OSL data structure.

However, overall the algorithm is 20 seconds slower when an OSL data structure is used. The reason is that insertion and deletion time, and the time to search *all* shapes are more costly when a complex data structure is used.

The complemeter is a CMOS microprocessor circuit. The layout view of the 1-bit complemeter contains 13 terminals. After abstraction, there are 50 frames and 99 pins or about 5 pins per terminal. This is not surprising, because some of the terminals are jumpers and are available on more than one layer. 32 instances of the 1-bit complemeter yield 242 terminals, about 3200 pins, and 1600 frames. After abstraction, there are only 1000 pins and 95 frames. The number of pins is reduced, because pins shared between adjacent instances are absorbed into frames.

An extraction tool and a layout rule checking tool must be written to take full advantage of this abstraction. These tools would process a view that contains only instances of abstract views and shapes that do not form devices. The former tool is really just a modification of Frame, because Frame can extract nodes. Any layout rule checker could serve as the latter tool if it was modified in the following way. Ordinarily, a layout rule checker processes pairs of the form:

< layer, shape >

The checker would be modified to process tuples of the form:

< layer, geometric function, shape, instance id >

Tuples with the same instance id are never checked against each other, because there can be no violations due to their interaction.

In the first case, there are two protection frames tuples whose instance ids are not equal. In the second case, there is a protection frame tuple and a "non-frame" tuple whose instance ids are not equal. In these two cases, if the two tuples are on the same layer, they cannot intersect at all and must be separated by at least the minimum separation distance between

two shapes on the layer.

There is one exception that occurs when pins are being shared. The case is a variant of the second case. First, the non-frame tuple must be a pin. Second, the gap between the two tuples is occupied by a pin tuple whose instance id is identical to the frame tuple's instance id. This gap can be infinitesimal when the two tuples abut.

Two "non-frame" tuples whose instance ids are not equal must be checked against each other. They can always intersect if they are on the same layer.

CHAPTER 7

CONCLUSIONS

Hierarchical and regular design methods for circuits have been used successfully to attack the problem of increasing circuit design times. A circuit computer-aided design (CAD) framework speeds the implementation of instances of these classes of design methods, because the client can re-use software for common operations and can focus on his particular design method. Also, a framework that is built using a portable software design method makes it possible to use a variety of computers and terminals without major impact on the underlying software.

The requirements for a circuit CAD framework were described. The implementation of a real circuit CAD framework—the combination of the portable operating system 4.2 BSD UNIX, the device-independent graphics package MFB, the database package Squid, and the user interface Hawk—was presented and was evaluated. Finally, a number of new circuit CAD tools were implemented within the framework.

Continuation of this work is necessary in order to take full advantage of the framework.

With the addition of:

- partitioning, placement, and routing tools
- refined array makers
- separate extractors and layout rule checkers for leaf views and views that conform to the protection frame model
- the net list comparator WOMBAT
- a refined version of the Python compaction tool that is compatible with the protection frame model
- the simulator SPLICE2

- module generators
- an alphanumeric terminal emulator for Hawk windows
- the net listing tool NLP

the framework and associated tools will be able to support a variety of modern design methods.

Further work requires the use of the framework on a number of designs, using a variety of design methods, to tune the framework and gain further experience with its strengths and weaknesses.

REFERENCES

1. Lattin, W., VLSI Design Methodology: The Problems of the 80's for Microprocessor Designs, *First Caltech Conference on VLSI*, , January 1979, 247-252.
2. Mayo, J., Design Automation: Lessons of the Past, Challenges for the Future, *IEEE Computer Graphics*, Sept. 1983.
3. Newton, A. R., The VLSI Design Challenge of the 80's, *Proc. 17th Design Automation Conference*, , June 1980, 343-344.
4. Noyce, R., Hardware Prospects and Limitations, in *The Computer Age: A Twenty-Year View*, J. Moses and M. Dertouzos (ed.), MIT Press, Cambridge, May 1980, 321.
5. Lattin, B., VLSI Design Methodology The Problem of the 80's for Microprocessor Design, in *Proc. 16th Design Automation Conference*, June 1979.
6. Newton, A., Computer-Aided Design of VLSI Circuits, *Proceedings of the IEEE* 69, 10 (October 1981), .
7. Haydamack, W. and D. Griffin, VLSI Design Strategies and Tools, in *HP Journal*, vol. 32, June 1981.
8. Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.
9. Crawford, J., *A Unified Hardware Description Language for CAD Programs*, ERL, U. of California at Berkeley, Aug. 1979.
10. Newton, A. R., The Simulation of Large-Scale Integrated Circuits, *ERL Memo No. ERL-M78/52*, , July 1978.
11. Cohen, E., Program Reference for SPICE2, *Electronics Res. Lab.*, ERL Memo ERL-M592, June 1976.

12. Sequin, C. and A. Newton, *A Structured Interchange Format for the Description of Integrated Circuits*, CAD Group, 321 Cory Hall, U. of California at Berkeley, May 1980.
13. Newton, R., *Symbolic Layout Language*, CAD Group, 321 Cory Hall, U. of California at Berkeley, 1981.
14. Chu, K., J. Fisburn, P. Honeyman and Y. Lien, Vdd—A VLSI Design Database System, *Engineering Design Application IEEE Catalog No. 83CH 1886-1*, (May 1983), .
15. Rao, K., D. McLeod and K. Narayanaswamy, An Approach to Information Management for CAD/VLSI Applications, *Engineering Design Application IEEE Catalog No. 83CH 1886-1*, (May 1983), .
16. Wiederhold, G., A. Beetem and G. Short, A Database Approach to Communication in VLSI Design, in *IEEE Transactions on CAD of ICs and Systems*, vol. CAD-1, April 1982.
17. Guttman, A. and M. Stonebraker, Using a Relational Database Management System for Computer Aided Design Data, *IEEE Database Engineering* 5, 2 (June 1982), .
18. Stonebraker, M., B. Rubenstein and A. Guttman, Application of Abstract Data Types and Abstract Indices to CAD Data Bases, Memorandum No. UCB/ERL M83/3, Electronics Research Laboratory, University of California, Berkeley, CA, January 1983.
19. Roberts, R. and I. Goldstein, *The FRL Manual*, MIT AI Laboratory, Cambridge, 1977.
20. Moon, D., *LISP Machine Manual*, MIT AI Lab, 1982.
21. Mitchell, J., W. Maybury and R. Sweet, *Mesa Language Manual*, Xerox Palo Alto Research Center, April 1979.
22. Novak, G., GLISP, in *The AI Magazine*, vol. IV, U. of California at Berkeley ERL, Fall 1983.

23. Stefik, M., D. Bobrow, S. Mittal and L. Conway, Knowledge Programming in Loops, in *The AI Magazine*, vol. IV, U. of California at Berkeley ERL, Fall 1983.
24. The American Heritage Dictionary of the English Language, , 1976.
25. Billingsley, G., KIC, MS Report, Department of Electrical Engineering and Computer Sciences of University of California at Berkeley, Fall 1983.
26. Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
27. Jones, A., Protection in Programmed Systems, in *PhD Thesis*, Carnegie-Mellon U., Pittsburgh, June 1973.
28. Dec., T. S., *VAX 11/780: Software Handbook*, Digital Equipment Corp., Maynard, MA, 1978.
29. Tichy, W. F., Design, Implementation, and Evaluation of a Revision Control System, in *Proc. of the 6th International Conference on Software Engineering*, Sept. 1982.
30. Keller, K. and M. Stonebraker, Embedding Hypothetical Databases and Experts in a Database System, in *Proc. SIGMOD Conference*, 1980.
31. Newman, W. and R. Sproull, Principles of Interactive Computer Graphics, , 1982.
32. Teitelman, W., *Interlisp Reference Manual*, XEROX Palo Alto Research Center, Palo Alto, CA, 1978.
33. McWilliams, T. and L. Widdoes, SCALD: Structured Computer-Aided Logic Design, Technical Report No. 152 , Digital System Lab, EECS Dept., Stanford U. , Stanford, CA , March 1978 .
34. Committee, G. S. P., Status Report, in *Computer Graphics*, vol. 13, August 1979.
35. Calma Inc., C., The STREAM Layout Language, in *Internal Memo*, 1984.
36. Keller, K. and A. Newton, A Symbolic Design System for Integrated Circuits, in *Proc. 19th Design Automation Conference*, June 1982.

37. Keller, K. and A. Newton, KIC 2: A Low-Cost, Interactive Editor for Integrated Circuit Design, in *Proc. 24th COMPCON*, Feb. 1982.
38. Bentley, J., D. Haken and R. Hon, Fast Geometric Algorithms for VLSI Tasks, in *IEEE COMPCON*, Spring 1980.
39. Lauther, U., A Data Structure for Gridless Routing, in *Proc. 17th Design Automation Conference*, June 1980.
40. Kedem, G., The Quad-CIF Tree: A Data Structure for Hierarchical On-Line Algorithms, in *Proc. 19th Design Automation Conference*, June 1982.
41. Wilmore, J., The Design of an Efficient Data Base to Support an Interactive LSI Layout System, in *Proc. 17th Design Automation Conference*, June 1980.
42. Ousterhout, J. and D. Ungar, Measurements of a VLSI Design, in *Proc. 19th Design Automation Conference*, June 1982.
43. Ousterhout, J., Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools, *IEEE Transactions on CAD/ICAS CAD-3*, 1 (January 1984), .
44. Vladimirescu, A., Simulating VLSI Circuits, *PhD Thesis, EECS - UCB*, Berkeley, CA, Jan 1983.
45. Stonebraker, M., The Design and Implementation of INGRES, *ACM Trans. Database Systems*, Sept. 1976.
46. Stonebraker, M., Retrospection on a Data Base System, *ACM Trans. Database Systems*, Sept. 1980.
47. Newton, A. R., D. O. Pederson, A. L. Sangiovanni-Vincentelli and C. H. Séquin, Design Aids for VLSI: The Berkeley Perspective, *IEEE Transactions on Circuits and Systems*, , July 1981.
48. Johnson, S., YACC: Yet Another Compiler Compiler, in *UNIX User's Manual*, U. of California at Berkeley Computer Science Dept. CSRG, .

49. Ousterhout, J., Caesar: An Interactive Editor for VLSI Layouts, *VLSI Design II*, 4 (Fourth Quarter 1981), .
50. Ousterhout, J., Caesar: An Interactive Editor for VLSI Layout, in *Proc. 24th COMPCON*, Feb. 1982.
51. Whitney, T., Description of the Hierarchical Design Rule Filter, SSP File # 4027, Caltech CS Dept., October 1980.
52. Newell, M. E. and D. T. Fitzpatrick, Exploiting Structure in Integrated Circuit Design Analysis, *Proc. Conf. on Adv. Research in VLSI*, MIT, Boston, Mass., Jan. 1982.
53. Scheffer, L. and M. Tucker, A Constrained Design Methodology for VLSI, in *VLSI Design*, vol. 3, May/June 1982.
54. McCalla, W. J. and D. Hoffman, Symbolic Representation and Incremental DRC for Interactive Layout, *Proc. IEEE Int. Symp. on Circ. and Syst.*, Chicago, Illinois, April 1981.
55. Niessen, C., Hierarchical Design Methodologies and Tools for VLSI Chips, in *Proc. IEEE*, vol. 71, Jan. 1983.
56. Bales, M., Layout Rule Spacing of Symbolic IC Artwork, MS Report, Electronics Research Lab of U. of CA at Berkeley, May 1982.
57. Lock, E., Techniques for the Construction of Parameterized Functional Modules, MS Report, Electronics Research Lab of U. of CA at Berkeley, Dec. 1981.
58. Chen, N., C. Hsu and E. Kuh, The Berkeley Building-Block Layout System for VLSI Design, ERL Memo, ERL, Univ. of CA at Berkeley, February 14, 1983.
59. Arnold, M. and J. Ousterhout, Lyra: A New Approach to Geometric Layout Rule Checking, in *Proc. 19th Design Automation Conference*, June 1982.

APPENDIX A

SOURCE CODE

The envelope attached to this page contains microfiche upon which is printed the source code for OSL, Frame, Squid, and Hawk.

APPENDIX B

PACKAGE CONTRIBUTORS

The following packages are integrated to some degree into the framework.

Author	Package's Function	Package's Name
John Ousterhout	MOS circuit timing estimation.	Crystal
Mark Bales	Compaction.	Python
Mark Hofmann	Non-Manhattan circuit extraction.	Ibex
Mike Arnold	Layout rule checking.	Lyra
Peter Simanyi	GRaPH making.	GRF
Brian Lee Ken Keller	Emulating alphanumeric terminals on graphics terminals.	WISH
Dan Fitzpatrick	.cadrc file parsing.	Cadrc
Ken Fishkin	Selecting colors.	colortran
Peter Moore	Memory management.	nmalloc
Peter Moore	Mask operations.	fa
Jim Kleckner	Dynamic linking-loading.	Dyn
Ken Keller Mark Bales Giles Billingsley Brian Lee	Device-independent graphics.	MFB
Ken Keller Brian Lee	Device-independent raster hardcopy graphics.	MHC
Ken Keller	Manhattan transformation.	MT
Ken Keller	Clipping shapes to rectangles.	HA
Ken Keller	Elliptical arc display.	EI
Ken Keller	File System routines.	FS
John Ousterhout	Parsing csh-style file names.	Pa
John Ousterhout	Hashing.	Hash
Jim Kleckner Ken Keller	Code timing.	Stop
Jim Kleckner Ken Keller	Storing waveforms.	WAP
S. Johnson	Generating parsers.	yacc
M. Lesk	Generating scanners.	lex
NA	Stream I/O.	stdio
NA	sqrt, sin, etc.	math
NA	Operating system.	UNIX

APPENDIX C

HAWK TUTORIAL

Tutorial for the Hawk Framework for Circuit CAD

Ken Keller
U. C. Berkeley

10.1. Introduction

10.1.1. Assumptions

It is helpful if you have read the chapters entitled:

Requirements of a Circuit CAD Framework
The Squid Package
Hawk

of Ken Keller's PhD thesis.

It is assumed that you can use UNIX, csh, and at least one of the local graphics editors: KIC, Caesar, or gremlin.

10.1.2. Hawk

To the circuit CAD programmer, Hawk is a programming environment which has many useful subroutines for manipulating and displaying circuit CAD information. To the circuit designer, Hawk is a graphics editor and shell.

The Hawk *window manager* manages multiple windows on a graphics terminal's screen.

Assorted types of objects can be displayed in windows. Thus far, the types are:

Type	Represents
WISH	Alphanumeric terminal's screen.
Squid	Multiple views of circuits.
GRF	Waveforms.

The WISH alphanumeric terminal's screen type has not been released yet. Brian Lee will be installing this type in Hawk shortly.

The data model of SPICE decks can represent devices and their connectivity. The data model of the CIF interchange format can represent hierarchical layouts. The data model of the *Squid subroutine package* can represent connectivity and layout information and the relation between them. Each file that has been created and edited solely by calling the Squid subroutine package is termed a *Squid circuit view*.

At this time, the GRF waveform type is available only to the developers of Hawk's Net List Processor (NLP) commands. The NLP commands will provide a common user interface to SPICE3, SPLICE2, and Wombat.

The Squid subroutine package also abstracts the UNIX file system a little. Each directory represents a cell or circuit and each file within a cell directory represents a view or representation of the cell. Thus, a view file is named by a triple:

< cell'sName, view'sName, accessMode >

A view can be accessed for browsing or editing. Cells can be named using the csh's tilde notation or relative to other cells via a library or path construct as explained below. The subroutines that deal with cells and views rather than the contents of Squid circuit views can be called by any tool.

Hawk has a built-in *display list interpreter* for displaying Squid circuit views. The Hawk

window manager calls the WISH and GRF packages to display the types they deal with.

Hawk's *scheduler* listens to the keyboard and graphics input device of the graphics terminal for input, and schedules *commands* to be executed as a result of input. A command may be a subroutine or a process, but is almost always a subroutine. A set of related commands is sometimes termed a *tool*.

Hawk is *extensible* for two reasons. First, it is possible to add commands to Hawk without recompiling Hawk. Second, by calling UNIX, Squid, and Hawk; commands built by users that implement circuit CAD algorithms do not have to implement I/O.

10.1.3. Disclaimer

Because Hawk is extensible and a research project, it is changing. Thus, the goal of this tutorial is to acquaint you with the principles of operation of Hawk so that you can learn the details of Hawk by yourself. If and when Hawk changes, you will be able to make sense of many of the changes without re-reading this tutorial.

10.1.4. Getting Started

Login on a computer in which Hawk is installed—currently ucboz and ucbic at a supported graphics terminal:

TerminalName	Manufacturer	Model	# of Colors	Note
j3	Jupiter	7	256	
kk	Tektronix	4113	16	Enables pop-up windows.
t5	Tektronix	4113	16	No pop-up windows.
AE	AED	512	256	Berkeley Evans Hall PROMs enables pop-up windows.

The AED and Jupiter terminals have a reliability problem. While operating Hawk on one of these terminals if the terminal should ring its bell, then *very calmly* wait until Hawk

has finished drawing—this will be signaled by a visible cursor. If your screen is quite messed up, type:

```
ctrl-l
```

Hawk will redraw your screen. Infrequently, the terminal will be "locked up". If this happens, pressing the *reset* function key in the upper left corner of the terminal's keyboard will have *no effect*. All you can do is turn the terminal off and then on again. This will log you out so you must log in again.

All of the terminals have either a mouse with three buttons or a tablet on top of which is a mouse—sometimes termed a *puck*— that has four buttons. All known mice have three buttons. It will be assumed that all pucks follow the Summagraphics convention for color coding puck buttons. This assumption is valid for the terminals in Berkeley's EECS Dept.

Type to the shell:

```
setenv MFBCAP ~ cad/lib/mfbcap
mkdir ~ /project
cd ~ /project
cat >path
● ~ cad/lib/hawk/technology/mosisNMOS ~ cad/lib/hawk
```

The directory named `~ /project` represents a *cell* or circuit. The file named `~ /project/path` represents a *view* of type *path* of the cell named `~ /project`. The file is a *stranger view*, because the file was created and edited by you instead of the Squid subroutine package. Again, if a file is created and edited solely by the Squid subroutine package, it is a *circuit view*.

The view type *path* is so named, because the function of it is the same as the function of the csh environment variable named `PATH`. It enables the client to use *partial* file names—those that do not begin with a slash in order to save typing and in order to be able to relocate the files that the partial file names refer to without having to change the partial file

names.

A view of type *path* is a list of full directory names. If you are logged in at ucbic, the list for `~/project/path` is:

```
'pwd' /cad/lib/hawk/technology/mosisNMOS /cad/lib/hawk
```

Partial file names are expanded into *full* file names— those that do begin with a slash— using this algorithm:

For each full directory name in list of full directory names contained in the view of type *path*,

```
fullFileName = strcat(fullDirectoryName, partialFileName);
switch accessMode
case editing:
  If fullFileName exists and is writable or
  it is permissible to create fullFileName,
  return;
case browsing:
  If fullFileName exists and is readable,
  return;
```

Now, it is evident why ● is the first element of `~/project/path`— anything you create will have priority over what is in Hawk's libraries.

If you are logged in at a terminal whose name is `terminalName`— see the table above for valid terminals— type:

```
~/cad/new/hawk -display terminalName -plotter vp
```

10.2. Basic Hawk

10.2.1. Screen Layout

The layout of windows on the screen is termed a *desktop* and is subject to change.

There will always be a window that is about as wide as the screen, but only one or a small

number of characters tall. This window is the *typescript window*. You should always watch this window for help and information in general. If the information is urgent as with a diagnostic, your terminal's bell will be rung. You must press any key once you have read the information before you may do anything else.

There will always be a window that is tall and thin or short and fat that is displaying a bunch of rectangles with a label on each. This window is the *layer menu window*. Each rectangle stands for a layer and the label on it is the layer's name. The rectangle shows the color and fill pattern that shapes on the rectangle's associated layer will be displayed in.

There may be several other windows. A window that contains a bunch of labels—one on top of another is usually a *command menu window* and the object displayed in it is just a special Squid circuit view that acts as a menu. In a menu, submenus are labels in all caps and commands are labels in mixed case.

Each window has a stack of objects in it. The top of the stack is what is displayed. A window can be empty.

Each window has several tiny rectangles around its border termed *tabs* or *light buttons*.

They are:

Location	Function
upper middle	pan upwards
lower middle	pan downwards
left middle	pan to the left
right middle	pan to the right
upper left	window containing tab becomes the current window
upper right	pop the object off the stack of objects in the window containing tab
lower right	display status of window containing tab

Currently, there is no help on light buttons so you will have to re-read this or watch *msgs*

for changes.

10.2.2. Graphical Input

The middle mouse button and the yellow puck button are the *point* button. By pressing it, you notify either Hawk or the command that is executing that you wish to point at something on the screen.

In general, the other buttons pop up windows when you press them. Only one pop up window can be displayed at a time. Thus, before a window is popped up, any window that is already popped up is erased.

Currently, the left (right) mouse button and the white (blue) puck button are bound to a window that is displaying a menu of user (Hawk) commands and submenus. Currently, the green puck button is bound to a window that displays the view type whose name is the value of your environment variable named `HA_PART_MENU_VIEW`, of the cell whose name is the value of your environment variable named `HA_PART_MENU_CELL`.

10.2.3. Invoking Commands

Again, a command is just a subroutine. To invoke a command, type a colon followed by the name of the subroutine that implements it. You can cancel a command when one is prompting you by pressing the `ESC` key. There are two other ways to invoke commands: pick a menu selection and press a key.

When a menu selection is picked, if the menu selection is displayed in a pop up window, the pop up window will usually be erased.

In general, a menu selection that is in all upper case is a *goto submenu* command. Go to a submenu. This just pushes the submenu's menu on the stack of objects of the window

displaying the menu. Go back to the previous menu by pointing at the pop tab. This just pops the submenu's menu off the stack of objects of the window displaying the menu.

To associate key **key** with a command named **command**, put a line in the **HAWK** section of your **.cadrc** file of the form:

ALIAS key command

Here are the names of the commands available by default.

Class	
Command	Function

Stretching	
strUL	Stretch Upper Left corners of selected rectangles.
strLR	
strLL	
strUR	

Shape Drawing	
rectActive	Draw rectangles for layout.
rectTerm	Draw rectangular pins for layout or schematics.
circleTerm	Draw circular pins for schematics.
arcTerm	Draw arc pins for schematics.
arcFrame	Draw arcs.
labelFrame	Draw labels.
setJustification	For labels.
setFont	For labels.
setHeight	For labels.
lineFrame	Draw vectors.
arrowFrame	Draw arrows.
lineActive	Draw paths for layout.
setWidth	For thick lines.

Selection	
delete	Delete elements in selected set.
addPtSel	Add elements to selected set.
addRectSel	"
subPtSel	Subtract elements from selected set.
subRectSel	"
desel	Clear selected set.
move	Move selected set elements.
copy	Copy selected set elements.
upsideDown	Copy selected set elements upside down.
sideways	Copy selected set elements sideways.
sel90	Copy selected set elements rotated CCW by 90 degrees.
sel180	Copy selected set elements rotated CCW by 180 degrees.
sel270	Copy selected set elements rotated CCW by 270 degrees.

Editing	
save	Save top view in current window's view stack.
pushByTyping	Push a view in the current window.
pushByPointing	Push the view in the current window in other windows.
pushBang	Re-push or re-edit.
pushInContext	Sub-push or subedit.

Windows	
copyWindo	Copy window.
createWindo	Create window.
updateWindo	Re-create window.
delWindo	Delete window.

Display Control	
HAvisible	Make visible selected layers only.
HAzoomin	Zoom in.
HAzoomout	Zoom out.
HAPAN	Pan.
HAfull	Show whole object.

View Control	
definedView	Defined view command.
physicalView	"Expand" command for layouts.
symbolicView	"Unexpand" command for layouts

Grid Control	
gridOn	Turn grid on.
gridOff	Turn grid off.
gridPitch	Set grid pitch.

Plotting	
justPlot	Plot current window.
plotAndSpool	And spool plot.
justDump	Plot screen.
dump&Spool	And spool plot.
slide	Temporarily have current window cover screen.

Layout Tools	
lyra	Run Lyra LRC.
violation	Probe Lyra violations.
critical	Probe Crystal critical path

Placement	
place	Place an instance.

Beaver Schematics	
selConnect	Connect selected terminals.
selDisconnect	Disconnect selected terminals.
selJazzNets	Jazz nets of selected terminals.
dangle	Jazz dangling terminals.

Framing	
frameMenu	Frame physical view in current window.
readRules	Re-read FRAME section of .cadrc file.

Property & Parameter List Editing	
aParmList	Make current plist actual parameter list of current instance.
fParmList	Make current plist formal parameter list of object in current window.
fPropList	Make current plist property list of object in current window.
aPropList	Make current plist property list of current instance.
geosPropList	Make current plist property list of current shape.
setInst	Set current instance.
setGeo	Set current shape.
updatePList	Change a property of current plist of current object.
delPList	Delete current plist of current object.
getPList	List current plist of current object.

A currently popular alias list for layout is:

```

begin HAWK
  ALIAS w HAZoomin
  ALIAS o HAZoomout
  ALIAS p HApAn
  ALIAS f HAFull
  ALIAS r rectActive
  ALIAS e delete
  ALIAS + addPtSel
  ALIAS - subPtSel
  ALIAS t addRectSel
  ALIAS _subRectSel
  ALIAS ^ x desel

```

```
ALIAS m move
ALIAS g copyWindo
ALIAS S save
end
```

Because of the way dynamic linking of command subroutines is implemented in Hawk, some commands are linked at the time Hawk is compiled and some are not. To force the latter ones to be linked, you must pick them from menus. Thus, you cannot use the keyboard to invoke the latter ones until you have picked them from menus at least once.

10.2.4. Manipulating a Single Window

Go the window manger menu **WINDOW**.

Create a window by pointing at the menu label *create* in the window manager menu. The user menu is erased and the window manager prompts you in the typescript window at the top of the screen. When creating the window, take care not to overlap any other windows. Hawk does not work correctly when windows overlap, but it does not force them not to. Yes, this is not great. This may change.

Set the current window by pointing at the appropriate tab in your window. Often, commands expect the current window to be set. In the beginning, you will probably forget to do this, but you will soon become accustomed to this.

Move and/or change the size of your window by pointing at the menu label *recreate* in the window manager menu.

Get the status of your window by pointing at its status tab.

Delete your window by pointing at the menu label *delete* in the window manager menu.

10.2.5. Selecting Layers

Hawk maintains a set of selected layers. Initially, none of the layers in the layer menu are selected. Point at the rectangle associated with a layer to select it. Hawk draws a rectangle around each selected layer. To un-select a selected layer, point at the rectangle associated with the layer.

10.2.6. Getting Help

Select the layer named **help**. Now, each time you point at a label in a menu, information about the command associated with the menu label will be printed in the typescript window until you un-select this layer.

This is terse help. A verbose help is planned.

10.2.7. Editing an Object

Create a window—it is empty. Point to the menu label **editByTyping** in the top-level of the user menu. Type:

```
foo physical
```

This will create a directory named **foo** in your directory named **~/project**. Also, a file named **physical** will be created in your new directory named **~/project/foo**. **foo** is a cell and **physical** is a type of view of it. This view of this cell is pushed on the empty stack of objects in your window. If the view is new, the view is made a Squid circuit view by default.

Point at random places in the window—each time you point, the coordinates of where you pointed at will be displayed in the typescript window. The numbers next to **dx** and **dy** yield the Manhattan distance between successive locations you point at. These numbers can

be used as a sort of ruler.

Display the status of your window.

10.2.8. Pan & Zoom

Whether or not you see a grid, zoom in by pointing at the menu label **zoomin** in the Hawk command menu. You will have to pop up the window containing the Hawk command menu. Don't forget to set the current window. If you did not, cancel the command or just pretend you did and the command will probably inform you that you did not. In any event, set the current window and point to the command again. Zoom in until you have a grid. There are grid-control commands in the window manager menu. Try them or use help to find out what they do.

Try the other commands in the Hawk command menu including **pan**, **zoomout**, and **full**. The panning tabs can also be used to pan the current window.

10.2.9. Shape Drawing in General

In general, once a shape drawing command is invoked, you will be prompted to point to define *control points* for the shape to be drawn. The shape will be drawn on the selected layers of the object in the current window. Each drawing command is a loop or *mode*. Each pass through the loop, you define one shape per selected layer.

For shapes such as paths that can be made up of an arbitrary number of control points, you must point to the same position twice or press the **ESC** key in order to finish off the shape being drawn. For shapes such as rectangles and arcs, the number of control points is a small constant so there is no need for such a convention. Most shape commands will highlight all control points entered so far so that it will be easy to choose the next one.

As an example, go to the layout menu by pointing at the menu label **SLIDE** in the top-level user menu. Select the **NP** layer. Point to the menu label **vectors** and point in your window to draw a few vectors on the selected layer. Cancel the command.

10.2.10. Multiple Windows

Go to the window manager menu and copy your window a few times. Make one of them display all of the vectors. Make the others display some of the vectors. You can use the others as magnifying glasses by setting one of them to be the current window, pointing at the pan command, and panning the current window by pointing at the window that is displaying all of the vectors.

10.2.11. More on Editing

Suppose you don't like what you have done. Invoke the command **re-edit** to get the previous version of your Squid circuit view which of course is empty.

To edit a different Squid circuit view, pick the pop tab and again invoke **editByTyping**.

10.2.12. Desktops

This section is a copy of the section on desktops in Chapter 5 of Ken Keller's PhD thesis.

In case you have not read all of Chapter 5, the user-supplied—don't worry, there is a default—subroutine named **HABegin** is called with all command line arguments that Hawk does not recognize directly. You already know that Hawk recognizes the arguments **-display** and **-plotter**.

A *desktop* is a configuration of windows independent of what is displayed in the windows. At this time, the default **HABegin** implements desktops. This subroutine's first argument is

a desktop name and its remaining arguments are a sequence of:

< cellName, viewName, accessMode >

triples that name circuit views to display in the windows of the desktop.

Desktop	Function
slides	Slide-making.
horizontalhop	Editing fat, short layouts.
verticalhop	Editing tall, skinny layouts.
one	One big window.
three	Two little windows stacked and a big window next to them.
zap	Four windows used for layout.

In summary, suppose you want three windows with each containing a different view of a cell and you only want to edit the last view. You would type to the shell to invoke Hawk:

```
~ cad/new/hawk -display terminalName -plotter vp -three
cell view1 r cell view2 r cell view3 w
```

10.3. Tools

10.3.1. Selected Set

There are a set of commands that manipulate a *selected set* of objects in the Squid circuit view displayed in the current window. Terminals, instances, and shapes can be selected. Instances can only be selected if the layer named `inst` is selected. Only shapes on selected layers can be selected unless the layer named `all` is selected at which time all shapes can be selected. A terminal is selected by selecting any one of its associated pins.

The menu labels `+point` and `-point` add and subtract respectively pointed to objects from the selected set. The menu labels `+rectangle` and `-rectangle` add and subtract respectively the *portion* of objects that intersect a selection rectangle you define by pointing twice. Thus, `+rectangle` behaves like a *rectangular saw*— cutting all objects into pieces that lie inside

and outside of the selection rectangle.

The menu label `clear` clears or empties the selected set.

A variety of commands operate on all selected objects in the current window. For example, all of the Beaver schematic capture tool's commands do this. The menu label `move` moves the objects in the selected set by the displacement you define by pointing twice. The menu label `erase` deletes the objects in the selected set. The menu labels `copy`, `upsideDown`, `sideways`, `90degrees`, `180degrees`, and `270degrees` *copy* the objects in the selected set to the Squid circuit view being displayed *in the current window*. Since the current window at the time of selection and at the time of copy may differ, *objects can be copied from one Squid circuit view to another*. Thus, Squid circuit views can be regarded as both menus and design objects. Remember that when you clear the selected set, the current window must be the window that was current when you started selecting. All rotations are counter-clockwise, upside-down means to flip about the x-axis, and sideways means to flip about the y-axis.

10.3.2. Hardcopy

The plotters available are:

PlotterName	Manufacturer	Model	# of Colors	Note
vp	Versatec	small	2	Works on uc cad or uc bic.
h7mfb	HP	7221a	4	Works on uc cad or uc bic. Starts plotting when you press "chart hold" button on plotter.

The program `~ cad/bin/mfb2lpr` must be installed on your machine for plotting to work.

To command the default `HABegin` to plot a set of views without being logged in at a graphics terminal, type:

```
~ cad/new/hawk -plotter plotterName -plotwindow cell1 view1 ... celln viewn
```

There are commands that plot the current window and the whole screen. See the upper half of the menu that can be reached by pointing at the menu label **SLIDES**.

10.3.3. Slide & Figure Making

There is a special menu that can be reached by pointing at the menu label **SLIDES** that has commands for drawing shapes whose functions are frame. For those who have not read about the Squid data model, a *frame* is just a shape that is not a wire and not a pin.

Plotting commands are also in this menu. The menu label **photograph** can be used to temporarily display the current window so that it covers the whole screen— then it can be photographed with a tripod-held camera or the Matrix Instruments film recorder by Ken Keller's desk. Pressing the return key will cause the window to resort to its original size.

When the technology `~cad/lib/hawk/technology/whiteSlides` is included in your `~/project/path`, a convenient set of layers is at your disposal when you are drawing with Hawk. The main layers are named **red**, **green**, **blue**, and **bw**. A shape drawn on one of the layers named for the three primary colors will always be plotted in the color whose name is the same as the layer's name— if you are using a color plotter. A shape drawn on the layer named **bw** will be plotted in white. The layers named **CONNECT**, **SOLDER**, and **SYMBOL** required by the Beaver schematic capture tool are also present in this technology.

10.3.4. Leaf Cell Layout

There are a small set of commands for layout of leaf cells in the menu reachable by picking the menu label **LAYOUT**. There is a command to draw rectangles and a command to draw rectangular terminals or pins. The selected set commands—especially **+rectangle**— are useful for modifying drawn rectangles. There are four commands— **upLeft**, **upRight**, **lowLeft**, **lowRight**— for moving the corners of the rectangles in the selected set.

Once you have drawn a portion of a layout or a whole layout and you want to check that it obeys layout rules, you can pick the menu label **lyra** to command the Lyra layout rule checker to analyze a rectangular area. Be sure that your **.cadrc** file has the correct technology declared in it. Lyra erases old violations and displays any new violations as "violation rectangles" on the layer named **lyrar**. If it is not evident what a violation rectangle means, pick the menu label **violation** and point to a violation rectangle—Lyra will display a sequence of characters that stand for the violation—see **lyra(cad.1)**.

Leaf cells should be drawn as views named **physical**. A physical view can be turned into a symbolic view by running the program **frame** or picking the menu label **FRAME**. For instructions on how to use **frame**, see Chapter 6 of Ken Keller's thesis. Be sure to edit your **.cadrc** file before attempting to run **frame**.

Leaf cells can have other leaf cells placed within them— that is physical views can contain instances of other physical views. Thus, it is not necessary to turn every physical view into a symbolic view. Turn a physical view into a symbolic view only when you will not be forming transistors anymore. If you want to rename a terminal, just draw a terminal with the desired name on top of the terminal to be renamed. In general, the terminal that is highest in the instance hierarchy will obscure all terminals on the same layer that intersect it.

To place an instance of a view, pick the menu label **SCHEMATIC** and then the menu label **place**. Once you have drawn leaf cells for transistor, contacts, etc., you may wish to place them in a view of a cell that will serve as a menu. By displaying this menu in a window next to the window that is showing the Squid circuit view that you are editing, you can copy menu items into the view you are editing.

Often, it is useful to edit a view that has been placed as an instance in the view being displayed in the current window. There are several ways to accomplish this. First, create a

new window and push the view to be edited in it. Second, pick the menu label **sub-edit** and point at the instance. The window will be redrawn, but all shapes except those in the view to be edited will be shown in the same color to indicate that they cannot be edited. Any changes to the view will be seen immediately in all instances of the view. Once you have finished modifying the view, picking the pop light button will redraw the window and you will see the results of your changes in color.

10.3.5. Floor Plan Layout

A floor plan layout is just a physical view with instances of symbolic views in it. There are currently no special commands for such views even though it is possible to construct hierarchical extraction, hierarchical layout rule checking, and routing aids. The rngroup may be developing such tools.

10.3.6. Schematic Capture

See Chapter 6 of Ken Keller's PhD thesis.

10.4. Wish List

This section is an evolving "wish list" that documents what users would like to see in Hawk. Having it here will serve to stimulate thought as to what is possible and prevent you from "making wishes" that have already been made.

Extend Squid to have an arrow shape type so that arrows can be operated on sensibly.

Polygons.

Selection of control points so that "move" becomes a general stretch.

Undo.

Redo.

Verbose help.

Comprehensive help.

Use stippling for better display of un-editable shapes.

Display un-editable shapes even when edit and display views are the same.

Automatic culling of shapes within instances.

Re-route command in Beaver.

Allow user routing in Beaver.

Bus support in Beaver.

Instance arrays in Beaver.

Editing labels.

Transform label justification.

String search for labels, terminals, etc.

Command table.

KIC-style attribute menu for interactively defining layers, colors of layers, etc.

A path command that solves the half-lambda problem.

Routing aids.

Release NLP so that Beaver can be used to drive simulators and Hawk can be used to display simulator output.

Release WISH so ctrl-z does not have to be typed in order to run shell commands.

"Video ls" to minimize typing of file names.

Have Ibex write its net list as a Squid view.

Simple stretchable cell capability.

Finish Python.

"make"-like support for re-evaluating schematics and symbolic views when lower levels of the instance hierarchy have changed.

True "desktop" support.

Support for black & white terminals.

Fang interface.

Per-object selected set—currently the selected set can only hold sub-objects from a single object.

Read-in of objects should be faster by replacing the parser built by yacc and lex.

ciftosquid that supports CIF files with hierarchy.

10.5. Bug List

10.5.1. Fatal

re-edit.

Placing an instance of a view that does not exist.

10.5.2. Annoying

Subtract from selected set by rectangle.

squidlyra is too slow on large areas.

10.6. Installing a Command

See Chapter 5 of Ken Keller's PhD thesis. In review, Hawk distinguishes Hawk-commands and client-commands—each command is in only one command-space. The menu selections associated with the subroutines you can write must be placed in a client-command menu. The information associated with the top-level, client-command menu is represented by the cell named `~ cad/lib/hawk/menus/userCommand`. There is a menu-making program named `menuview`. The command line:

```
menuview <~ cad/lib/hawk/menus/userCommand/src
~ cad/lib/hawk/menus/userCommand symbolic
```

causes the contents of the stranger view named `src` to be read by `menuview` and the top-level menu to be written into the circuit view named `symbolic`.

To make your own menu and install commands in it, you must create a directory to represent your menu, create and edit `src` in your directory, run `menuview`, create a window in Hawk for your menu, and finally edit your menu in the window. From this point on, you can pick menu selections from your menu.

To install a command in the top-level menu, you must create a directory `~/project/menus/userCommand` to represent your version of the top-level menu, copy `userCommand/*` to it, edit `path` so that it searches your directory and `~/cad/lib/hawk/menus/userCommand`, edit `src`, and re-run `menuview`.

To install a menu in your top-level menu, you must do the same thing. In addition, you must create a directory to represent your menu as explained above.

10.6.1. Simple Command

The subroutines named `fRects` and `oRects` in the object file named `~/cad/lib/hawk/menus/userCommand/geos/rects.h` implement rectangle drawing:

```
#define MAX(dragon,eagle) ((dragon) > (eagle) ? (dragon) : (eagle))
#define MIN(dragon,eagle) ((dragon) < (eagle) ? (dragon) : (eagle))

static void rects(filledP)
SQBool filledP;
{
    HAWindow currentWindow;
    SQGeo rect; /*Rectangle being drawn.*/
    SQGeo firstCorner; /*Highlight of first corner drawn.*/
    SQID layerGenerator;
    SQIntegerPoint lowerLeft,upperRight; /*Corners.*/
    SQIntegerPoint integerPointTriple[3]; /*Store for firstCorner.*/

    rect.geoType = sqRect;
    rect.function = geosF;
    rect.implements.term.name = termsName;
    rect.filledP = filledP;
```

```

firstCorner.geoType = sqLine;
firstCorner.function = sqFrame;
firstCorner.def.line.nPath = 3;
firstCorner.def.line.width = 0;
firstCorner.def.line.path = integerPointTriple;
firstCorner.filledP = sqTrue;
firstCorner.layer = "hawk";

for(;;) { /*Drawing commands are modes.*/

    currentWindow = HACurrentWindo();
    if(currentWindow.windoID == NULL) {
        HATypescript(sqTrue,"Please select the current window first.");
        return; }

    /*Prompt user to point at first corner of rectangle.*/
    HATypescript(sqFalse,"POINT to 1st corner of rectangle.");
    HAListen();
    /*Did he cancel?*/
    if(HAKeyTyped() == HAESC) return;
    if(HAMenuSelectionP()) return;

    SQEmptyBBSet(HAChangedRect());
    lowerLeft = *HACursorPositionL();
    integerPointTriple[1] = lowerLeft;
    integerPointTriple[0].x = lowerLeft.x;
    integerPointTriple[0].y = lowerLeft.y+HASQUNITSPERLAMBDA;
    integerPointTriple[2].x = lowerLeft.x+HASQUNITSPERLAMBDA;
    integerPointTriple[2].y = lowerLeft.y;
    SQPush ViewStk(currentWindow.editStk);
    /*Create highlight of first corner.*/
    SQ(sqCreate,sqGeo,&firstCorner);
    SQ(sqGet,sqGeo,&firstCorner,integerPointTriple,3,NULL,0);
    SQAddToBBSet(&firstCorner.bb,HAChangedRect());
    SQPop ViewStk();
    /*
    Just redraw the area of the object in the current window
    that the highlight covers.
    */
    HADisplayView(currentWindow.windoID,*HAChangedRect(),sqFalse,sqFalse);

    /*Prompt.*/
    HATypescript(sqFalse,"POINT to 2nd corner of rectangle.");
    HAListen();

    /*Cancel?*/
    if(HAKeyTyped() == HAESC OR HAMenuSelectionP()) {
        /*Yes.*/
        SQPush ViewStk(currentWindow.editStk);
        SQ(sqDelete,sqGeo,firstCorner); /*Delete highlight.*/
        SQPop ViewStk();
        /*Erase highlight.*/
        HADisplayView(currentWindow.windoID,*HAChangedRect(),sqFalse,sqFalse);
    }
}

```



```

    if(HAKeyTyped() == HAESC)
        continue;
    else return; }

/*Didn't cancel.*/
upperRight = *HACursorPositionL();

rect.def.rect.l = MIN(lowerLeft.x,upperRight.x);
rect.def.rect.r = MAX(lowerLeft.x,upperRight.x);
rect.def.rect.b = MIN(lowerLeft.y,upperRight.y);
rect.def.rect.t = MAX(lowerLeft.y,upperRight.y);
SQPush ViewStk(currentWindow.editStk);
SQBeginLayerGen(&layerGenerator);
/*For each selected layer.*/
while(SQGenLayer(layerGenerator,&rect.layer) != sqEndGen)
    if(HALayerSelectedP(rect.layer))
        SQ(sqCreate,sqGeo,&rect); /*Create rectangle drawn.*/

/*Delete & erase highlight.*/
SQ(sqDelete,sqGeo,firstCorner);
SQPopViewStk();

SQAddToBBSet(&rect.def.rect,HACheckedRect());
/*Redraw.*/
HADisplayView(currentWindow.windowID,*HACheckedRect(),sqFalse,sqFalse); }
}

static void fRects()
/*
Enable user to draw Filled RECTangleS
on selected layers
of Squid circuit view in current window.
*/
{
    rects(sqTrue);
}

static void oRects()
/*
Enable user to draw Outlined RECTangleS.
on selected layers
of Squid circuit view in current window.
*/
{
    rects(sqFalse);
}

```

10.6.2. Advanced Command

The subroutines named `lyra` and `violation` in the object file named `~cad/lib/hawk/menus/userCommand/lyra.o` implement the layout rule checking tool named Lyra [59]. Actually, the subroutines implement the *interface* to Lyra. Lyra is a large program written in the Lisp programming language. The first time the subroutine named `lyra` is called, it executes Lyra as a UNIX process so that it can communicate with Lyra via an inter-process communication object in the future. UNIX processes do not share the same address space so the inter-process communication object is not shared storage. Thus, if Lyra has a bug in it or exceeds its resource limits, Hawk will continue to execute and notify the user. Hawk and Lyra communicate via a simple language that is quite similar to CIF [8] called *Caesar format* [49].

When the user invokes the subroutine named `lyra`, the subroutine named `lyra` is called. If Lyra is not already executing on the subroutine's behalf, the subroutine executes Lyra. The subroutine prompts the user to point at the corners of a rectangle that brackets the area he wishes Lyra to analyze. Once the user has done so, `lyra`:

- Grows the area by **MAXRULE**.
- Calls Squid to retrieve all shapes that intersect the grown area.
- Communicates the shapes to Lyra in Caesar format via an inter-process communication object.
- Deletes any rectangles on the layer named `lyra` that are a result of past Lyra analyses. The `lyra` layer will be explained shortly.
- Waits for Lyra to fail or to communicate any layout rule violations in Caesar format. Each violation is a rectangle and a label. The rectangle brackets the area that contains the violation and the label explains the violation.
- Creates a rectangle on the layer named `lyra` for each violation rectangle and inserts a string-valued property named `lyra` on the created rectangle's property list whose value is the violation label.
- Calls Hawk to redisplay the grown area in all windows that display it.

When the user invokes the subroutine named `violation`, the subroutine prompts the user to point at a set of violation rectangles on the layer named `lyra`. Once the user has done so, the subroutine calls Hawk to display the values of the property named `lyra` on the pointed-to rectangles' property lists.

You may read the source file for `lyra.o` at your leisure. It is not stored in the stranger view named `lyra.c`. It is convenient to be capable of executing Lyra *interactively* from Hawk as a menu selection and in *batch mode* as a UNIX program. The source file is stored in its own directory, named `~ cad/src/squidLyra`, just as most UNIX program source is. The directory contains a command for producing `~ cad/lib/hawk/menus/userCommand/lyra.o` as well as the UNIX program named `~ cad/bin/squidlyra`.