T

Micro-D

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

ALE - A Custom Layout Methodology for
Bipolar Integrated Circuits

by

Norbert Diesing, B.A.Sc.

A thesis submitted to the Faculty of Graduate
Studies and Research in partial fulfillment
of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Carleton University
Ottawa, Canada
December 1987

© copyright
1987, Norbert Diesing

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-40650-X

The undersigned recommend to the Faculty of Graduate Studies
the acceptance of the thesis:

"ALE: A Custom Layout Methodology for
Bipolar Integrated Circuits"

submitted by Norbert Diesing in partial fulfillment of the
requirements for the degree of Master of Engineering.

Prof. C.H. Chan
Thesis Supervisor

Prof. M.A. Copeland
Chairman
Department of Electronics

December, 1987

ABSTRACT

In this thesis, an efficient, automated methodology for producing correct layouts of custom bipolar integrated circuits will be discussed. Schematic capture, a simulator interface, automatic layout, layout editing, and layout verification are described. Layouts can be automatically compacted to remove excess area. A correspondence between schematic and layout is maintained to allow incremental connectivity checking. Device construction can be verified and design rules incrementally checked. Layout extraction and postlayout simulation is automated, and a constraint system is provided to prevent parametric errors.

ACKNOWLEDGEMENTS

I would like to thank Bell Northern Research (BNR) for making their facilities available to allow this work to be researched. Prof. C.H. Chan of Carleton University provided valuable assistance throughout the project. Useful discussions were held with the staff of the BNR bipolar design groups and the staff of the BNR layout automation groups.

TABLE OF CONTENTS

Chapter 1. Introduction	1
1.1 The Layout Problem	3
1.1.1 Polygon Editors	7
1.1.2 Symbolic Layout	9
1.1.3 The ALE Methodology	11
Chapter 2. Overview of the ALE Methodology	13
2.1 Current Status of ALE	22
Chapter 3. The ALE Architecture	28
3.1 Object Oriented Programming	29
3.2 Computing Environment for an ALE Implementation	31
3.3 User Extendibility	33
3.4 Man Machine Interface (MMI)	35
3.5 Multi-Tasking	37
Chapter 4. ALE Technology File Definition	41
4.1 Technology Constants	42
4.2 Layer Definitions	43
4.3 Layer Display Definitions	44
4.4 Layer Connectivity Definitions	47
4.5 Design Rule Definitions	48
4.6 Device Construction Definition	51
4.6.1 Device Properties Definition	53
4.6.2 Property Constraints Definition	56
4.6.3 Symbol Shape Definition	56
4.6.4 Symbol Pins Definition	57
4.6.5 Automatic Layout Definition	58
4.6.6 Hierarchy Proposal	59
4.6.7 Hierarchy Constraints	61
4.6.8 Layout Constraints	62
4.6.9 Device Verification Definition	64
4.6.10 Parameter Extract Definition	67
4.6.11 Parameter Comparison Definition	68
Chapter 5. The ALE Methodology	70
5.1 The Need for Incremental Layout	70
5.2 Data Structure Summary	71
5.3 Schematic Editing	74
5.4 Schematic Simulation	75
5.5 Coupling Between Schematic and Layout	76
5.6 Library Devices	84
5.7 Automatic Layout Generation	85
5.8 Interactive Editing	87
5.9 Error Detection	89
5.9.1 Device Correctness	95
5.9.2 Device Parameter Correctness	98
5.9.3 Connectivity Correctness	98

5.10	User Constraints	100
5.11	Postlayout Simulation	107
5.12	Simulation Verification	110
5.12.1	Waveform Comparison	110
5.12.2	Simulation Errors	115
Chapter 6.	Automatic Layout	117
6.1	Automatic Floor Planning	117
6.2	Manual Adjustment of the Floor Plan	121
6.3	Primitive Device Generation	123
6.4	Nonprimitive Device Creation	130
Chapter 7.	Conclusions	132
7.1	Summary	132
7.2	Suggestions for Further Work	133
References		135
Glossary		139
Appendix A.	Complete Technology Definition for NPN	142
Appendix B.	Quadratic Placement Algorithm	147
B.1	The ALE Placement Algorithm	148
Appendix C.	Maze Routing Algorithm	153
C.1	The ALE Routing Algorithm	156
Appendix D.	Compaction Algorithm	160
D.1	The ALE Compaction Algorithm	160

LIST OF ILLUSTRATIONS

Figure 1.	A Custom Bipolar Layout	8
Figure 2.	The layout process in ALE	14
Figure 3.	Circuit Schematic of a NPN Current Mirror	15
Figure 4.	Layout of an NPN Current Mirror	18
Figure 5.	ALISP Example	26
Figure 6.	Epi Region Merging	79
Figure 7.	Allowable Differences Between Schematic and Layout Domains	81
Figure 8.	Automatic Layout Procedure	86
Figure 9.	Creating a NPN Device Construction Error	92
Figure 10.	Automatic Layout and Manual Editing for a NPN	97
Figure 11.	An Epi Region Illustrating Resistance Calcula- tion Difficulties	104
Figure 12.	Distance Between Signals	114
Figure 13.	Chip Schematic	118
Figure 14.	Chip Layout Tree	118
Figure 15.	Completed Floor Plan	122
Figure 16.	Some Examples of NPN Transistors	127
Figure 17.	Some Examples of NPN Emitter Rows	128
Figure 18.	Some Examples of PNP Transistors	129
Figure 19.	A Placement Produced with Modified Rectangle Constraints	152
Figure 20.	Routing a track through a NPN transistor	155
Figure 21.	A Layout and the Corresponding Metal Search Tree	157

CHAPTER 1. INTRODUCTION

Integrated Circuit (IC) layout is the process of converting a circuit schematic into a geometrical database suitable for manufacture [1]. When performed manually, this conversion process is time consuming, error prone, and requires a great deal of skill. Automating the layout process has been the goal of assorted layout methodologies, and many Computer Aided Design (CAD) programs implementing these methodologies.

Custom layout [2] is one methodology where integrated circuit layouts are created at a level of abstraction near the polygonal level.¹ The complexity of custom layout is therefore very high because every polygon of every device may be completely specified. Although most of the ICs designed today do not utilize custom layout techniques because of time and cost tradeoffs, there is still a demand in some applications for the flexibility and low production costs that custom layout can provide. However, because of the limited use and high complexity of custom layout, very

¹ The term custom layout is also used in standard cell methodologies to refer to standard cell layouts that are manually edited. In this usage, custom layout refers to manually optimizing cell placements and centreline interconnect. In this thesis, custom layout will refer to layout methodologies where every polygon of every structure can potentially be manually modified.

little custom layout automation exists. This lack of automation can be intensely frustrating when one is creating custom layouts, and provided the motivation for this thesis work.

This thesis will present an automated methodology for custom bipolar layout. The methodology is termed ALE as an acronym for Automated Layout Editor. In order for ALE to be effective, the following characteristics are desirable:

- A coupling between schematic and layout domains to allow incremental connectivity checking.
- All structures from devices to entire ICs to be automatically created.
- Manual layout editing to allow the automatic layout to be tailored to meet cost and performance criteria.
- All verification to be performed incrementally for high performance.
- Postlayout simulation and verification to allow a final check on layout performance.
- Parasitic analysis to ensure that parasitics do not affect the circuit functionality.
- An extendible and technology independent architecture to prevent early obsolescence.

ALE will be described in several steps:

- The layout problem and a few existing layout methodologies will be reviewed.

- An overview of ALE will be presented.
- A description of the programming environment in which ALE should be implemented.
- A description of the technology files which permit ALE to be tailored to a particular technology.
- A discussion of the ALE methodology.

The description of ALE will be followed by conclusions, suggestions for future research, and a glossary of terms. Also included are Appendixes that briefly describe some of the algorithms required for automatic layout.

1.1 The Layout Problem

In an integrated circuit, devices are constructed on a wafer using combinations of mask levels [3,4]. A mask level defines polygonal regions where a particular IC manufacturing process step will be performed. Mask levels may be physically represented as photographic plates that are used to photolithographically reproduce polygonal regions onto the wafer, but other representations and transfer techniques are possible. Regardless of the precise technique used in transferring the mask level data onto the wafer, at some point the data is represented in a computer as a polygonal database or layout.

Correctly creating this polygonal database is the layout problem. The criteria that define the correctness of a layout are:

- The layout must be design rule correct [3,4]. There are technology dependent rules that specify permissible spacings, overlaps, widths, and enclosures of mask layers as they relate to themselves or other mask layers.
- The layouts of all devices must be created in accordance with rules that define valid polygon configurations for the devices.
- The devices in the layout must be connected together according to the design intent.
- The parameters of devices in the layout must correspond to the design intent. Some examples of typical parameters are the size of a device and the driving ability of a device.
- Parasitic devices in the layout must not interfere with proper circuit operation.

Selecting the best layout for a circuit is very difficult, and has been found to be a NP complete problem [5,6,7]. Layout designers who manually create layouts are able to deal with this difficult problem in ways that are not well understood and thereby create high quality layouts. However, these layouts are very expensive in terms of

manpower and elapsed time, and there is therefore a strong incentive to automate the layout problem with computers.

With the current state of computers and algorithms, simplifying assumptions must be made in order for a computer to create reasonable layouts. Most automated methodologies break the layout problem down into placement and routing phases [1,8]. Heuristics are then applied to both the placement and routing problems to achieve a final layout. Some methodologies predefine a placement and therefore avoid the placement problem completely. Other methodologies create the placement through a predefined procedural definition.

Some examples of methodologies which attempt to deal with the layout problem are:

- Custom (Polygon level, Symbolic, ALE)
- Standard Cell
- Programmable Logic Array (PLA)
- Gate Array
- Module Generator
- Silicon Compiler

These layout methodologies all have a distinct place in integrated circuit layout, as does custom layout. Custom layout is used when:

- New primitives for other layout styles are required, i.e. the creation of the contents of standard cells.

- A circuit requires careful optimization and tight design tolerances in order to function according to specification. This often occurs in analogue designs.
- The integrated circuit volume will be very large. Because the cost of an IC is proportional to its area, the large volume allows the smaller area of a custom layout to compensate for the greater engineering costs and time delay associated with a custom layout.

Most layout methodologies have a fixed layout style that simplifies automation. For example, both gate array and PLA styles [9,10] have predefined layouts which are tailored with a few mask levels to achieve the desired circuit function. The standard cell layout style positions predefined pieces/of layout (standard cells) in rows, and routes connections in the channels between rows [11]. Because of rules governing the construction of the standard cells, rules governing the way in which the cells can be placed, and rules governing the way in which channels can be routed, standard cell layout is considered a fixed layout style.

Alternately, custom layout has an unfixed layout structure [4]. There is an infinite number of ways in which a set of arbitrary polygons can be arranged to create a layout that is a correct representation of a circuit. There is also an infinite number of ways in which a set of polygons can be arranged to incorrectly represent a circuit. All polygon

configurations are possible in custom layout. The difficult task is not only in creating a correct layout, but in verifying that the layout is correct.

This flexibility in polygon configurations is required to solve the problems addressed by custom layout, but the same flexibility makes custom layout a slow process prone to error. Because of these drawbacks, custom layout is used only in niche applications. However, if a layout methodology existed which allowed custom layouts to be created quickly and correctly, many of the objections to custom layout would vanish. An example of a small piece of a custom bipolar layout is shown in Figure 1.

1.1.1 Polygon Editors

Various strategies to automate custom layout have been attempted. The least automated is a polygon editor that uses a graphics screen and a set of editing commands to allow the layout designer to capture a layout and then modify it interactively [4]. These editors typically do not incorporate any technology information other than display customization, and are little more than computerized drawing boards.

In this form of polygon layout the degree of automation is minimal and the potential for error high. After the original layout has been produced, verification programs are

Figure 1. A Custom Bipolar Layout

used to ensure that the layout is correct. Much manual editing and computer time is expended during this step until the layout is considered correct by the verification programs. Because of limitations in existing verification programs, some final manual checks are performed to guarantee correct circuit operation.

This degree of manual labour is unacceptable for any reasonable speed of design turnaround. In order to reduce the manual labour, either the complexity of the polygonal layout problem must be managed, or polygonal layout must be abandoned in favour of a higher level of abstraction.

1.1.2 Symbolic Layout

One higher level approach is symbolic layout where low level devices or parts of devices are represented on a graphics screen by symbols [12]. The layout designer graphically places these symbols and interconnects them with centre-lines. The program automatically expands the symbols into their actual graphical representation, and can compact the layout to minimum area while obeying design rules [13,14].

Because symbolic layout programs contain design rule information and some technology construction information, they are significantly more advanced than basic polygonal level editing. They allow layouts to be produced that have correctly constructed devices and that are design rule

correct, and therefore can greatly reduce the verification burden on the designer. Unfortunately, symbolic layout programs do not usually produce layouts that are guaranteed connectivity correct without further layout verification.² This is because these programs do not deal with the circuit schematic (or some similar representation) that defines the required connectivity of the circuit.

Symbolic layout is better than manual polygonal layout in terms of throughput, but is restrictive in several respects:

- The set of allowable device configurations is limited because the automatic symbol expansion is not as flexible as the imagination of a custom designer.
- The compactor is not as good at optimizing area as a custom layout designer with patience and time. This is a problem because symbolic layout programs do not often provide the ability to edit the compacted layout without a one way migration to a polygon editor.
- The lack of connectivity checking is a drawback because other verification programs need to be run before the layout can be considered correct.

² Some exceptions to this rule are Topologizer [15] and Dumbo [16].

1.1.3 The ALE Methodology

The ideal custom layout tool is one which can be a valuable assistant in the schematic capture and simulation stage.¹ It can automatically layout the circuit and achieve a quality as good as that which a custom designer can achieve. This layout would be guaranteed to perform as the designer's original circuit simulation indicated.

Such a custom layout tool is not currently practical, and possibly will never be practical. However, with modern software techniques and modern computers it is possible to get close. The ALE methodology is one such attempt. ALE is not yet close to a final implementation as the amount of programming work is staggering. Nevertheless, the concepts and ideas required to create a system such as ALE will be discussed.

Parts of ALE have been implemented as a prototype, but as the ALE concept has evolved, certain notions and ideas have changed. Therefore, the prototype implementation is no longer representative of current concepts. For this reason, the ALE methodology will not usually be explained in terms

¹ Ideally the tool would also create the circuit implementation of the design from a higher level description. ALE addresses the physical layout implementation aspect of custom design and therefore circuit compilation will not be discussed.

of the prototype implementation, but in terms of how a new implementation should be developed.

ALE has been created with the specific needs of the custom analogue bipolar designer in mind. These needs are largely a superset of the needs of custom layout in other technologies. Therefore the ALE methodology can be applied to most technologies, although only bipolar will be discussed here.

CHAPTER 2. OVERVIEW OF THE ALE METHODOLOGY

ALE's major goal is to produce custom layouts. However, in order to ensure layout correctness, both schematic capture and a simulator interface is required. Schematic capture is needed to allow a direct coupling between schematic and layout that can then be used for incremental connectivity checking. The simulator interface is needed to allow the performance of both the schematic and layout to be verified.

The procedure that ALE follows in producing a layout is illustrated in Figure 2. This process is explained briefly in the next few pages as an introduction, and will be explained in greater depth in subsequent sections.

The first step is for the designer to capture the circuit schematic or netlist. In an iterative procedure, this schematic is simulated and modified until the circuit performs to the designer's specifications. This iterative phase uses only the designer's expertise as ALE has no inherent design capabilities. ALE does not include any direct simulation abilities but provides a generic simulator interface so that many different simulators can be integrated into ALE. An example of a simple circuit schematic is the NPN current mirror of Figure 3.

Figure 2. The layout process in ALE

7

During the schematic capture and simulation step, the designer also produces a simulation template that ALE will subsequently use in verifying the layout's performance. A simulation template is a simulator setup that ALE passes to a simulator to produce simulation output. This simulation output is converted into an ALE standard format which can then be used to compare a circuit schematic simulation to a postlayout simulation.

Once simulation is complete, the layout process can begin. ALE takes the circuit schematic hierarchy and transforms it into a layout hierarchy. The organization of the layout hierarchy is not necessarily the same as the schematic hierarchy because ALE can automatically perform some reorganizations to allow more efficient layouts. These changes are proposed by expert routines (Section 4.6.6) and checked for validity. If the reorganization is not valid, it is not performed.

The designer also has the capability to manually perform this reorganization. ALE checks this reorganization and warns the designer if the changes are likely to be

* The transformation from circuit schematic hierarchy to layout hierarchy is actually performed incrementally as the schematics are being captured, but it is conceptually easier to think of it as a batch process. More details are provided in Section 5.5.

invalid. In general, ALE will permit the designer to do things that are likely to be incorrect. Warnings are issued, but the changes are not forbidden.' Allowing this flexibility does not affect the final layout quality as ALE maintains all items that it considers to be in error, and these can be queried at any time.'

Once a layout hierarchy has been produced, ALE can automatically create the corresponding layout. ALE cannot always finish the layout, and cannot always construct the layout in a way that meets specified constraints. When this occurs, the user is warned with ALE's error reporting capabilities. Figure 4 contains the layout of the current mirror from Figure 3.

After a layout has been automatically produced, the user often wants to manually tailor it further. This is done by using layout editing commands and compaction capabilities.

-
- Allowing the user to do things which currently appear invalid can be viewed as an invitation to disaster. However, what appears invalid at one point in time may no longer be invalid if other changes are performed. This technique of temporary errors is often used in polygon level editing to effect intricate changes. For this reason, the user cannot be forbidden from creating errors.
 - Errors are stored as graphical objects which are stored in the layout along with all other graphical objects. The errors have text associated with them which describes the error in user understandable terms. This will be explained in greater detail in Section 5.9.

ties: All layout changes can be incrementally checked by ALE for design rule correctness, device construction correctness, connectivity correctness, parameter correctness, and for adherence to high level constraints. Any detected errors are reported to the user as graphical figures with associated text so that they can be corrected.

Once the layout has been modified to the user's satisfaction and all errors detected by ALE have been corrected, the layout can be extracted and simulated. The extraction phase is relatively minor since throughout the layout process ALE has been extracting various circuit parameters to ensure that they match those specified on the circuit schematic.

An extracted netlist includes parasitic capacitances and resistances as well as parasitic active devices such as transistors or diodes. The extracted netlist is simulated using the simulation template that the designer has created. The simulation outputs for the circuit schematic and extracted netlist are then compared for equivalence. Equivalence can be defined in many ways, and the user is provided

Many circuit parameters are too computationally intensive to extract interactively. These parameters are extracted as background processes that are interrupted whenever the user issues a command. Full details are in Section 3.5.

with the capability to customize what ALE considers equivalent.*

In this step ALE not only compares simulation outputs but it also checks parasitics. With current simulator technology it is not possible to simulate large circuits and include all parasitics be they passive or active. This limitation is due to the algorithm complexity required to guarantee simulation convergence for large circuits, and due to computer time and computer memory limitations. Because of this limitation, ALE does not simulate all extracted parasitics, but prunes them with a set of heuristics. Heuristics for pruning passive parasitics are normally based on their magnitudes. Active parasitics cannot be effectively pruned without knowing the precise way in which they will be biased, and this is not known until after the simulation is performed. Therefore, ALE initially prunes all active parasitic devices. The validity of this assumption is tested by examining the postlayout simulation to determine which active parasitics might have turned on. These are then marked with a property as being dangerous parasitics so that a subsequent simulation will include them.

* This customization capability is provided by user created LISP functions and is explained in Section 3.3.

If the results from the circuit simulation and the postlayout simulation are not equivalent, ALE has the capability to attempt to isolate where in the layout the problem exists. The basic strategy in this comparison is to compare the parasitics on the two netlists (which are identical in terms of device connections and device parameters) in an attempt to determine significant differences. This will be explained in greater depth in Section 5.12.

When a problem is isolated in the layout which causes the postlayout simulation to fail, high level constraints on the layout may be specified to ensure the layout's performance. These constraints are then checked by ALE during layout editing and when violated, ALE adds error markers to the layout.

Alternately, the simulation mismatch may be due to a poor design which results in changes to the circuit schematic being necessary. When the circuit schematic is modified, ALE can automatically determine the changes required to the layout. This is possible because ALE maintains a continual correspondence between circuit schematic and layout, even when a reorganization for a more efficient layout has been performed, and can therefore incrementally track connectivity errors.

Once the layout is error free and the simulation outputs match, the chip is ready for fabrication. A graphical database compatible with programs used to transform layouts into manufacturable mask data is produced, and the primary task of ALE is complete.

However, design rules and valid device constructions will change as the technology in which the chip is manufactured matures. At some point it may become economically feasible to try to take advantage of these changes by modifying the layout. Because ALE utilizes compaction and a knowledge of device construction, these changes can be almost completely automated.

2.1 Current Status of ALE

The previous section briefly describes the capabilities of a mature ALE. Much of this capability has not been implemented in the ALE prototype that currently exists.

The prototype ALE is coded in Pascal with all technology files coded in ALISP. ALISP is a version of LISP specifically created for ALE to capture technology information such as design rules, layer name definitions, and layer connectivity definitions. Because of the relative simplicity of the original technology information, ALISP began as a very rudimentary implementation with no user defined functions.

For example, capturing a design rule for the spacing between the p+ diffusion and the n+ diffusion was captured in ALISP as:

```
(ADDMINSPACING 'P+ 'N+ 4)
```

As the ALE prototype and the concepts behind it progressed, the simple LISP interpreter became more and more elaborate until it evolved into a reasonably full implementation of LISP with some special capabilities for layout manipulation. Unfortunately, the ALISP implementation is now showing the stress of the way in which it has evolved and is becoming difficult to enhance and support. As well, important capabilities such as windowing, property inheritance, multi-tasking, tailorable user interface, and rapid user interaction are lacking in ALISP.

The lack of multi tasking and rapid user interaction are symptomatic of the mainframe environment that the ALE prototype was developed in. Any new work on ALE should be done in a more suitable environment such as a dedicated workstation. Modern workstations generally provide multi-tasking, and can handle rapid user interaction if they are dedicated to a single user. Some of the other deficiencies are best solved by using a language other than ALISP as the core of the system (see Section 3.2).

Some of ALISP's special capabilities are:

- The inclusion of built in data types for:

Polygons: A collection of vertices that represent physical areas on a mask level.

Layers: Groups of polygons on a particular mask level.

Cells: A group of layers and cell placements.

Cell References: The instantiation of a cell with translation and orientation within another cell. This datatype allows hierarchical layout.

Centrelines: Line segments with width that are used to represent connectivity in a slightly more abstract way than polygons.

Connections: A data type that connects polygons and centrelines to each other and to nodal potentials.

Netlist data types: Netlist data types include nodes, device instances, and pins. They are used to represent the connectivity of circuit schematics within ALE.

Layer Attributes: Properties of layers such as design rules and display colours.

- The ability to manipulate polygons and layers using geometrical operators such as AND, OR, ANDNOT, INSIDE, OUTSIDE, and CUTTING.
- The ability to read a netlist from a file and process that netlist with inquiry and traversal functions.
- Provision of relative placement for creating devices.
- Ability to export a polygon database that can be used in creating masks.

An example of some ALISP code is shown in Figure 5.

The prototype ALE can create most of the devices that will be required in a custom layout and provides a noninteractive mode that can create a library of standard devices and their simulation models. This includes difficult devices such as split collector PNPs of which some examples are shown in Figure 18 on page 129. Most of the low level extraction and some of the low level design rule checking has been implemented, but the high level functions are not completed. Connectivity checking, device verification, and constraint checking have not been implemented. The quadratic placement algorithm has been studied and prototyped. However the router and global compaction have not yet been implemented.

The prototype ALE can best be described as an automatic device generator with interactive editing capabilities that allows rudimentary compaction, automatic relative placement, and manual routing abilities.

The ALE prototype provided a foundation for establishing the ideas discussed in this thesis and as such has served its purpose. It will never be able to provide many of the features discussed in this thesis - that must be left to

```
; Function MAKENPNEM produces a minimum size NPN emitter.
; The first argument to the function is the name of the
; cell to be returned. The second argument is the
; emitter node.
```

```
(DEFUN MAKENPNEM (NAME EMNODE)
  (LOCALVAR CHEIGHT CWIDTH CONTACT)

  (SETQ CHEIGHT (MINWIDTH 'CONTACT))
  (SETQ CWIDTH
    (ROUNDUPTOGDS (/ (MINAREA 'CONTACT) CHEIGHT)))
  (SETQ CONTACT
    (LAYER 'CONTACT
      (RECTANGLE 0 0 CWIDTH CHEIGHT EMNODE)))

  (CELL NAME
    CONTACT
    (BIAS 'NPLUS CONTACT (MINENCL 'NPLUS 'CONTACT))
    (BIAS 'METAL CONTACT (MINENCL 'METAL 'CONTACT))))
```

Figure 5. ALISP Example

a new implementation in a more suitable computing environment.

CHAPTER 3. THE ALE ARCHITECTURE

In order for a methodology such as ALE to be effective, certain assumptions about the computer environment in which it functions need to be made. These include:

- Multi-tasking, and multi-windowing
- Colour graphics
- Dedicated processing
- Mouse or graphics tablet for graphical pointing

These requirements are best met by a dedicated workstation that can provide the immediate response and high level of interaction demanded. Mainframe computers are best suited to batch processing, not interactive tools.⁶ The ALE methodology attacks a computationally intensive problem and does not shirk from using large amounts of compute power. This computing is always done in an attempt to guarantee that the final layout will operate correctly. Most workstations currently on the market do not have the required compute performance, but with the processing capability of workstations typically doubling every year, in a few years the workstations will be adequate.

⁶ Whenever a group of people want something simultaneously from something that can do only one thing at a time, most people will end up waiting in a queue. This is the behaviour of a mainframe which has to serve hundreds of people "simultaneously." A dedicated workstation has only one user and thus can give immediate response.

3.1 Object Oriented Programming

ALE utilizes object oriented programming concepts, and is heavily based on concepts from LISP. A good description of the object oriented paradigm is found in [17,18] but a brief description will also be given here. /

Any program implemented in an object oriented style could also have been implemented using conventional programming languages and styles. However, the same could be said in comparing high level languages and assembler languages. The object oriented style is one that allows code to be organized in a useful and supportable manner, one that encourages reuse of existing code, and one that simplifies programming projects involving teams.

An object is an encapsulated piece of data. The individual data elements are called instance variables or slots. Encapsulation is an important programming concept because it allows an object to contain all the required information to allow it to perform operations on itself such as display, modification, deletion, etc. These operations are called methods.

An object is always based on more primitive objects except for the low level primitive object. Each object inherits data such as methods or instance variables from its

ancestors, although these can be overridden by methods or instance variables on the object itself.

Some object oriented languages have the capability of defining new instance variables during execution. This is a useful capability as it allows objects to have their own property lists. Flavors and CommonLoops are two of the more important object oriented LISPs, and both have this capability [19,20].

Properties have always been a part of LISP, [21,22] and including them in the object oriented concept permits a great deal of flexibility. The usefulness of properties can be illustrated with the example of an NPN transistor. Most transistors are fairly generic, probably minimum geometry, and with a default set of electrical characteristics. Some transistors are not generic and must have their differences from the norm defined.

Using the object oriented paradigm and inheritance of properties, the library symbol for the transistor (an object) will have a descendant type of object known as an instance. This instance will represent the particular NPN on the schematic. The library symbol contains properties that describe the behaviour of the default NPN. In order for a particular instance of an NPN to have behaviour that differs from the default, the properties that describe how the NPN

differs from the default have to be added to the instance. These will override the properties on the library symbol.

In ALE, most data is represented as objects. The encapsulation of all information relating to an object in a single spot, as encouraged by object oriented programming, simplifies maintenance, program enhancements, and reduces source code size [17].

3.2 Computing Environment for an ALE Implementation

Originally ALE was implemented in a hybrid of Pascal and ALISP and resided in a mainframe environment. Now that the concepts for ALE have properly gelled, this is not adequate. The computing environment in which ALE is implemented must have the following features:

Object oriented:

This is needed to allow good program organization, to achieve an increase in development speed, and to take advantage of the built in inheritance features which would require substantial programming effort in other languages.

User extendible:

Allows the language itself to be used as the input for technology files, and allows a great deal of functionality to be included in the technology files at little development effort. As well, user extendibility allows the program to

evaluate code that the program itself has written. This will be important for advanced ALE applications.

Multi-tasking:

Needed to allow ALE to behave as a highly interactive program even though there are some computationally intensive tasks. Multi-tasking achieves this by computing the large processing tasks in background. Multi-tasking can be provided by virtually any computer language if the host operating system has the capability.

Multi-Windowing:

An important capability in all modern graphics systems. This allows features such as editing the schematic of a circuit in one window while editing the layout in another window. Multi-windowing is also a capability that is often provided by utilizing the services of the host operating system.

Properties:

Permits storage of parameters on devices. Properties add another dimension to user extendibility. If the user wishes to add new types of data into the system, properties are one mechanism that does not require source code modification.

The most popular object oriented languages are based on existing languages such as C, Pascal, and LISP. Each of these languages is capable of multi-tasking and multi-windows if the resident operating system supports these abili-

ties. However, while property lists are conceptually possible in object oriented C or Pascal, they already exist in object oriented LISP. As well, LISP is the only language which truly supports user extendibility as described previously.

This makes object oriented LISP the preferred language choice for ALE. Historically, LISP was considered a poor language for a system language because of low execution speed and poor development tools. Today, LISP environments are among the best in terms of development tools, and execution speed is no longer a serious problem because LISP can now be compiled.

The prototype implementation of ALE uses Pascal as its primary language and ALISP as its secondary language. In further implementation this would be reversed. An object oriented LISP would be the primary language, and Pascal or C used to solve computationally intensive problems such as design rule checking, placement, and compaction.

3.3 User Extendibility

ALE permits any user written LISP functions to be loaded into the ALE environment. These functions become part of ALE and are evaluated in the same way as any other ALE commands. To the user, these new functions become indistinguishable from predefined ALE commands.

All access to the ALE database is performed through a set of database manipulation routines. This includes the ALE program itself, background tasks, and extensions to ALE that the user creates himself. Giving the user access to the internal ALE database is a powerful feature and must be used carefully. This capability results in ALE being an open extendible tool, and allows the user to implement functionality that the basic ALE does not provide.

Some of the areas in ALE that require user extendibility are:

- Automatic device building where a procedural method can be defined that will be followed in creating the layout of devices.
- Defining device parameters and the way in which they will be extracted.
- Defining what polygon configurations represent valid device configurations.
- Defining new types of constraints that are to be checked.
- Creating a simulator interface. A simulator input file is created from either the ALE schematic representation, or the ALE layout representation. The simulator is then run on this file and the results processed into an ALE waveform format.
- Comparison of the circuit schematic simulation and the postlayout simulation for equivalence. Depending on the

circuit, the required performance may be based on rise times, absolute voltages, distortion, drive capability, etc.

- Determining what devices in the layout prevent the schematic and postlayout simulations from being equivalent.
- Defining what reorganizations are potentially valid layout structures. This is done by traversing the netlist^A and checking for specific device configurations. For example, if two PNPs have a common base potential, then it is often possible to place them in the same epi region.
- Checking that potential reorganizations are valid. For example, if a PNP ever operates in a saturating mode then placing that PNP in the same epi region with any other devices is potentially catastrophic. This check is performed by looking at the output of the simulation template to check the operating points of the devices in question.

3.4 Man Machine Interface (MMI)

Details of the ALE Man Machine Interface (MMI) are secondary in this work. Although in the final analysis, the MMI is what makes a tool successful, providing the user with an adequate set of MMI customization tools allows MMI details to be treated superficially. This section briefly describes the ALE MMI.

The MMI of ALE is based on a multi-window system with fixed menus, popup menus, form filling, keyboard commands, keyboard shortcuts, and command aliasing. The complete MMI is tailorable by the user.

A menu is a graphical object with areas that contain text or icons. When a pointing device (i.e. a mouse) is used to move a cursor into an active area of a menu, the active area is highlighted. A mouse click will evaluate the command that is associated with the highlighted active area.

Fixed menus are menus that are permanently displayed on the screen, and whose commands can be selected at any time. A popup menu appears on the screen in response to some user action such as a mouse click and will disappear after a command is selected.

Form filling is a user interface concept where the program displays a form on the screen. Each line of the form has a prompt field and an entry field. The prompt field displays text which informs the user what should be entered in the entry field. Default values can be used, and error checking for allowable values is performed.

A keyboard shortcut is a single key that has been associated with a command string. When the key is pressed, the command string is evaluated.

Command aliasing is a way of changing the names of commands. The default name may be undesirable to the user. By specifying an alias, or substitute name, a preferable command name may be created.

All commands to the system, whether entered as a keyboard shortcut, or from a menu, are expanded into a textual form that is passed to the ALE command interpreter. This expanded form of a command is identical to that which can be typed at the keyboard at any time after entering a command entry mode.

3.5 Multi-Tasking

ALE must operate in a multi-tasking environment. ALE is designed to perform most CPU intensive tasks in an incremental manner such that the user is not normally aware of the action. However, tasks such as extracting some types of device parameters, performing batch simulations, and some types of error checking cannot be performed in a short enough period of time to avoid perceptibly slowing the user down.

ALE deals with this by creating a queue of pending tasks that will be evaluated in background.¹⁰ Background

¹⁰ Because much of ALE is implemented in LISP, the queue can be implemented as a list of function calls that must be evaluated. The lack of a distinction between program

tasks will only be evaluated when the user is between commands, and suspended at other times.¹¹

This concept of background tasks within an application is similar but more powerful than that in the Magic incremental Design Rule Checker (DRC) [23] where checking of regions that have been modified is performed in the pauses between user commands. In Magic, if the user enters a command the check is aborted and started over after the command is finished. ALE can suspend a task while a user command is being evaluated and continue the task when the command is finished.

In order for this scheme of background tasks to be possible, tasks must not attempt to modify the database at the same time, or read the database while any other process is modifying it. This is accomplished by allowing tasks to disable interrupts from other tasks or user commands while initializing from the database, and while writing data back into the database.¹²

and data in LISP is a great advantage here, and strengthens LISP as a language choice in which to implement ALE.

- ¹¹ Background tasks are implemented by using the services provided by the operating system under which ALE is operating.
- ¹² This communication can usually be performed by using the inter-process communication capabilities of the computing environment.

Initializing a background task requires a separation of the data that the background task will manipulate from the data that user commands may modify. For example, if the required task is to compute the resistance of a polygon configuration using finite element analysis, a copy of all involved polygons must be copied into a private area to prevent unpredictable results if the polygons are altered by a user command while the resistance calculation process is suspended.

In this example, modifying these polygons results in the resistor extraction becoming meaningless because the calculation is for an out of date polygon configuration. To avoid wasting computer time due to this behaviour, whenever a user command completes, a check is performed to see whether the background processes that were being evaluated are still valid.¹³ Any invalid background tasks are terminated as further calculations are a waste of CPU time. Any other jobs that are waiting in the queue are also checked for validity and terminated if they are no longer valid.

When removing tasks from the queue and creating background tasks, ALE monitors the number of page faults and the

¹³ This is implemented by adding a description to the background task queue that describes the data elements that will invalidate a background task if modified. This description takes the form of a LISP expression which will be evaluated after each user command.

amount of processing required to suspend background processes and begin executing a user command. If the amount of resource becomes too high, ALE will begin suspending the most recently created background tasks until earlier background tasks finish. Thus ALE is able to monitor its efficiency in dealing with the work load, and temporarily lighten it by delaying some tasks until the work load lightens. This allows the speed of user interaction to remain as high as possible.

While tasks are being evaluated in background or waiting in a queue, it is quite possible that the user will want to observe the results of a task before the computation completes. In order to deal with this problem, any properties that the background task will modify are tagged as currently being computed. This tagging occurs during the background process setup time. If a user command attempts to access one of these properties, then a message is issued, and the user is given the option of aborting the command or waiting for the background process to compute the results. If waiting is the desired behaviour, then the priority of that particular background task is increased to maximum so that the results will be available as soon as possible.

CHAPTER 4. ALE TECHNOLOGY FILE DEFINITION

The purpose of technology files is to allow ALE to be tailored to a particular process technology. In ALE, this customization includes:

- technology constants
- layer name definitions
- layer display definitions
- layer connectivity definitions
- design rule definitions
- device definitions
- constraint definitions

The ALE technology file is constructed as LISP expressions. This simplifies processing the file, as each statement can be evaluated as a LISP instruction. Much of the user configurability in the technology file is achieved by imbedding LISP expressions as part of the technology statements.

Examples in this chapter are from a simple bipolar technology. The only devices that will be discussed are an NPN, and an epitub which is an epi region that contains other devices. This chapter is not complete. To limit the scope of the discussion, some items have been neglected or treated superficially. This chapter is not intended as a complete technology definition but rather to illustrate some

of the offered technology constructs, and to demonstrate some of the flexibility offered by the LISP approach to technology files.

4.1 Technology Constants

Every technology has basic constants which are important in creating or verifying layouts. The ALE technology file allows these constants to be entered and accessed via LISP global variables. The syntax is:

```
(SYSVAR varname varvalue)
```

Where `varname` is the global variable to be defined and `varvalue` is the value to be assigned to the global variable. `Varvalue` may be any LISP expression.

A new function called `SYSVAR` is used instead of the standard LISP `SETQ` to provide a measure of protection to these variables. `SYSVARs` are special variables which cannot be reassigned with a `SETQ` nor hidden by local variables. If an attempt is made to reassign or hide the variable, an error diagnostic is produced.

Example:

; Set up global variables

```
(sysvar xj_p+ 1.3)      ; p+ junction depth (u)
(sysvar xj_n+ 0.9)      ; n+ junction depth (u)
(sysvar nd_epi 1.5E16)   ; epi doping (1/cm**3)
(sysvar na_p+ 2.0E18)    ; p+ doping (1/cm**3)
(sysvar maxIbaseperu2 0.5) ; Max base contact current (ma/u**2)
(sysvar maxIcollperu2 0.5) ; Max coll contact current (ma/u**2)
(sysvar maxIemperu2 0.5) ; max emi contact current (ma/u**2)
```

4.2 Layer Definitions

The next part of a technology file contains a definition of the valid mask layer names. The syntax is:

```
(DEFLAYER layername)
```

Where **layername** is the name of the layer to be defined. This name will become a technology constant.

One predefined layer is **substrate** and is therefore not defined by the **DEFLAYER** statement. **Substrate** represents the physical substrate of the IC and is constantly updated to encompass all other layers.

Example:

; Define the valid layernames

```
(deflayer nburlay)      ; N buried layer
(deflayer dp+)          ; Deep p+ layer
(deflayer epi)          ; Epi tubs
(deflayer dn+)          ; Deep n+ layer
(deflayer p+)           ; p+ layer
(deflayer n+)           ; n+ layer
(deflayer contact)      ; Contact between n+ or p+ and metal
(deflayer met1)         ; First level of metal interconnect
(deflayer via)          ; Contact layer between met1 and met2
(deflayer met2)         ; Second level of metal interconnect
(deflayer passiv)       ; Window openings in passivation
```

4.3 Layer Display Definitions

Mask layers are displayed on the graphics screen with the parameters colour, line style, and fill pattern. Some mask layers may not always be displayed. These display attributes are defined with the following syntax:

```
(SETCOLOUR colourname redlevel greenlevel bluelevel)
(COLOUR layername colourname)
```

```
(SETLINESTYLE linestylename linestyledef)
(LINESTYLE layername linestylename)
```

```
(SETFILLPATTERN fillpatternname fillpatterndef)
(FILLPATTERN layername fillpatternname)
```

```
(VISIBILITY layername T|NIL)
```

Where colourname is a particular colour name. Some predefined colours such as blue, red, green, and cyan exist. Redlevel, greenlevel, and bluelevel define respectively, the intensity of red, green, and blue in the colour.

Linestylename is a particular line style name. Some predefined line styles such as solid, dash, and dot exist. **Linestyle**def is a sixteen bit number that represents the line style.

Fillpatternname is a particular fill pattern name. Some predefined fill patterns such as solid, ldiag, and rdiag exist. **Fillpattern**def is a sixteen element list of sixteen bit numbers that together represent the fill pattern.

Example:

; Define the colours of the layers

```
(setcolour gray 12 12 12)
(colour nburlay green)
(colour dp+ gray)
(colour epi brown)
(colour dn+ blue)
(colour p+ red)
(colour n+ cyan)
(colour contact orange)
(colour met1 magenta)
(colour via green)
(colour met2 gray)
(colour passiv red)
```

; Define the line styles of the layers

```
(setlinestyle dashdot (bin. "1110010011100100"))
(linestyle nburlay dash)
(linestyle dp+ solid)
(linestyle epi solid)
(linestyle dn+ solid)
(linestyle p+ solid)
(linestyle n+ solid)
(linestyle contact solid)
(linestyle met1 solid)
(linestyle via dashdot)
(linestyle met2 ldash)
```

; Define the fillpatterns of the layers

```

(setfillpattern clear (list 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(setfillpattern lrdiag (list (bin "1000000110000001")
(bin "0100001001000010")
(bin "0010010000100100")
(bin "0001100000011000")
(bin "0001100000011000")
(bin "0010010000100100")
(bin "0100001001000010")
(bin "1000000110000001")
(bin "1000000110000001")
(bin "0100001001000010")
(bin "0010010000100100")
(bin "0001100000011000")
(bin "0001100000011000")
(bin "0010010000100100")
(bin "0100001001000010")
(bin "1000000110000001"))
(fillpattern nburlay clear)
(fillpattern dp+ clear)
(fillpattern epi clear)
(fillpattern dn+ clear)
(fillpattern p+ ldiag)
(fillpattern n+ rdiag)
(fillpattern contact clear)
(fillpattern met1 clear)
(fillpattern via clear)
(fillpattern met2 clear)

```

; Set visibility of mask layers

```

(visibility substrate nil)
(visibility nburlay t)
(visibility dp+ t)
(visibility epi t)
(visibility dn+ t)
(visibility p+ t)
(visibility n+ t)
(visibility contact t)
(visibility met1 t)
(visibility via t)
(visibility met2 t)

```


4.4 Layer Connectivity Definitions

Mask layers electrically connect when certain overlaps occur. This is defined with the following syntax:

```
(CONNECT layername1 layername2 [contactarea])
```

Where layername1 and layername2 are the names of the layers that will be connected together. Contactarea is an optional argument which is an boolean expression relating mask layers. If included, contactarea defines the only areas where layername1 and layername2 can connect if they overlap. If contactarea is not included, then layername1 and layername2 connect wherever they overlap.

Example:

```
; Define layer connectivity

; Note that contact and passiv are assigned connectivity
; along with all other mask layers. This is for
; consistency and simplicity.

; dp+ connects to substrate where there is no nburlay
(connect dp+ substrate (&~ substrate nburlay))

; dp+ always connects to p+
(connect dp+ p+)

; Epi connects to n+ only where there is no p+
(connect epi nplus (&~ epi p+))

; nburlay always connects to epi
(connect nburlay epi)

; dn+ always connects to epi
(connect dn+ epi)

; dn+ connects to n+ only where there is no p+
(connect dn+ n+ (&~ dn+ p+))
```

```

; n+ always connects to contact
(connect n+ contact)

; p+ connects to contact only where there is no n+
(connect p+ contact (&~ p+ n+))

; contact always connects to met1
(connect contact met1)

; met1 always connects to via
(connect met1 via)

; via always connects to met2
(connect via met2)

; passiv always connects to met2
(connect passiv met2)

```

4.5 Design Rule Definitions

Simple design rules require checks only between original mask layers. However most technologies require checks based on layers derived through boolean operations from original layers. Rule checking must therefore have provision for derived layers. The syntax used in ALE is:

```

(DRCSETUP drcsetupname
  <((SETQ derivedlayer layerexp) |
    (ruletype layerexp1 [layerexp2] range [option]*)

```

Where drcsetupname is the name assigned to this particular DRC setup. This name becomes a technology constant.¹⁴

¹⁴ Assigning names to DRC setups allows various setups to be defined at any time.

SETQ is the standard LISP assignment operator which allows derived layers to be assigned names.¹⁵

Layerexp, layerexp1, and layerexp2 are boolean expressions which relate original mask layers, or previously defined derived layers together to create new derived layers.¹⁶

Ruletype is one of the following: WIDTH, AREA, SPACING, ENCL, LENGTH, or OVERLAP. This defines the check to be performed.

Range is a list of the form:

[[lessop lower] [greaterop upper]]

This defines a distance range in which an error will be flagged. Lessop is one of < or <=, and greaterop is one of >, or >=. Lower is the lower bound of the range, and upper is the upper range.

¹⁵ These derived layers are stored only for the duration of a DRC and then released.

¹⁶ The previously derived layers can be the results of a DRC operation. This allows conjunctive rules.

¹⁷ Selections can be performed on original mask layers on the basis of property values. This is the same operator as that in Section 4.6.9.

Option is one of the following: SHORTED, NOTSHORTED, OPPOSITE, NOTOPPOSITE, PARALLEL, NOTPARALLEL, or OUTPOLYGONS.¹¹

Example:

; Design rules for the simple bipolar technology

(drcsetup rules

; epi rules

(width epi (< 10))
(spacing epi (< 5))
(encl epi nburlay (< 5))

; n+ buried layer rules

(width nburlay (< 8))
(spacing nburlay (< 15))

; dn+ rules

(width dn+ (< 5))
(encl epi dn+ (< 6))

; p+ rules

(width p+ (< 5))
(spacing p+ (< 6))
(encl epi p+ (< 5))

; n+ rules

(width n+ (< 4))
(area n+ (< 16))
(spacing epi n+ (< 6))
(spacing n+ (< 4))
(spacing n+ p+ (< 4))
(encl p+ n+ (< 3))
(spacing n+ p+ (< 6))

; contact rules

¹¹ These options modify the behaviour of the DRC operation and behave in an analogous manner to similar options in other design rule checkers.

```

(width contact (< 4))
(area contact (< 20))
(encl p+ contact (< 3))
(encl n+ contact (< 2))
(spacing n+ contact (< 4))

```

```

; met1 rules

```

```

(width met1 (< 5))
(spacing met1 (< 5))
(encl met1 contact (< 2))

```

```

; via rules

```

```

(width via (< 5))
(area via (< 30))
(encl met1 via (< 1.5))

```

```

; met2 rules

```

```

(width met2 (< 5))
(spacing met2 (< 5))
(encl met2 via (< 2.5))

```

```

; passivation rules

```

```

(width passiv (< 80))
(spacing passiv (< 62))
(encl met2 passiv (< 4))
)

```

4.6 Device Construction Definition

Primitive device construction is one of the more intricate elements of the ALE system. Each device is defined by a single command in the technology file, but this command is complex and contains several subsections. The device command will therefore be described in sections. The principle device in the examples is an NPN with the complete description for the NPN device definition in Appendix A.

The basic structure of the device definition is:

```
(DEVICE devicename
  (PROPERTIES ... )
  (PROPCONSTRAINTS ... )
  (SYMBOLSHAPE ... )
  (SYMBOLPINS ... )
  (AUTOLAYOUT ... )
  (HIERPROPOSE ... )
  (HIERCONSTRAINTS ...)
  (LAYCONSTRAINTS ...)
  (DEVICEVERIFICATION ... )
  (PARAMETEREXTRACT ... )
  (PARAMETERCOMPARE ... )
)
```

Where:

devicename: The device name of the device being defined.

PROPERTIES: This section defines the form that will be displayed when a schematic symbol for the device is placed, or the parameters on the symbol are altered.

PROPCONSTRAINTS: This section defines a set of constraints that the input parameters must meet before being accepted by the system.

SYMBOLSHAPE: Describes the graphical shape of the schematic symbol for the device.

SYMBOLPINS: Defines the pins on the schematic symbol.

AUTOLAYOUT: Defines the procedure for automatically laying out the device.

HIERPROPOSE: The procedure to be followed in determining whether a reorganization should be proposed to allow more efficient implementation of the layout.

HIERCONSTRAINTS: Defines the procedure to verify that a proposed reorganization of devices into a new level of hierarchy in the layout is valid.

LAYCONSTRAINTS: Define the default constraints that are to be placed on the layout of this device.

DEVICEVERIFICATION: Defines the procedure to verify that the layout of the device has been correctly constructed, or that the layout of the device is still correct.

PARAMETEREXTRACT: Defines the names of the parameters to be extracted, and the method to be used in extracting them.

PARAMETERCOMPARE: Defines how the parameters on the devices in the layout will be compared to the devices in the schematic.

4.6.1 Device Properties Definition

This section defines the form that will be displayed when an instance of a schematic symbol for a device is created or altered. The prompts, variable names, variable types, default variable values, and permissible variable values or ranges are defined. The syntax is:

```
(PROPERTIES  
  <(propname prompt class type defaultvalue [permvals])>*)
```

Where **propname** is the property name that will be set and **prompt** is the text displayed to the user as a prompt. **Class** is a text string used to classify the purpose of the prop-

erty. Some valid examples are: Performance, Auto-Layout, Verification, or Constraint.

Type is the type of the property. Some examples are: integer, string, real, boolean, function, or lispexp.¹¹ Defaultvalue is the default value to be assigned to the property on initial creation. Permvals is optional and if specified defines the permissible values of the property. The syntax of permvals is:

(<value>*) | (RANGE lowend upend)

Where value is an allowable value for the property. A list of values creates an enumeration of the allowable values of the property. RANGE specifies that a range for the values of the property is to be specified. Lowend and upend define respectively the lower end and the upper end of the permissible range.

¹¹ Allowing a LISP expression as a property value is extremely useful in capturing information which is not fixed, and can only be determined through evaluating an expression. The LISP expression can also be used to capture a list of values on a single property.

Example:

```
(device npn
  ...
  (properties
    (size "Size of transistor in multiples of 1X"
      "Performance"
      'real
      1
      '(range 1 20))
    (emitype "Strip or separate emitter NPN"
      "Auto-Layout"
      'string
      'separate
      '(strip separate))
    (rowcount "Number of emitter rows"
      "Auto-Layout"
      'integer
      1
      '(range 1 5))
    (maxIbase "Maximum base current (mA)"
      "Verification"
      'real
      .1
      '(range .001 10))
    (maxIcoll "Maximum collector current (mA)"
      "Verification"
      'real
      3
      '(range .01 100))
    (maxIem "Maximum emitter current (mA)"
      "Verification"
      'real
      3
      '(range .01 100))
    (SPICEcreate "Function which creates the SPICE line"
      "Simulation"
      "function"
      'NPNSPICEcreate)
  )
  ...
)
```

4.6.2 Property Constraints Definition

This section defines a set of constraints that the input properties must meet before being accepted by the system. Each constraint contains a LISP expression and an error message. If the LISP expression evaluates to a true value, then the error message is displayed to the user. The user is then given the opportunity to correct the property values. The syntax is:

```
(PROPCONSTRAINTS <(boolexp errmsg)>*)
```

Example:

```
(device npn
  ...
  (propconstraints
    ((< size rowcount)
      "Transistor size not larger than the number of rows)
    ((& (> rowcount 1)
      (= emitype 'separate)
      (~(integerp size)))
      (concat "Size of transistor with multiple rows and "
        "separate emitters must be integer.)))
  )
  ...
)
```

4.6.3 Symbol Shape Definition

This section defines the shape of the symbol to be displayed in the schematic editor. The syntax is:

```
(SYMBOLSHAPE <(DRAW <(xcoord ycoord)>*) |
  (CIRCLE xcoord ycoord radius) |
  (ARC xcoord ycoord rad startangle stopangle)>*)
```

Where **DRAW** specifies that a list of coordinate pairs is to follow that defines a shape to draw. **CIRCLE** specifies that a circle is to be drawn at the specified location with the specified radius. **ARC** specifies that an arc is to be drawn at the specified location with the specified radius and start and stop angles.

Example

```
(device npn
  ...
  (symbolshape (draw (0 0) (2 0))
    (draw (2 -3) (2 3))
    (draw (2 1) (4 3) (4 5))
    (draw (2 -1) (4 -3) (4 -5))
    (draw (4 -3) (3 -1) (2 -2) (4 -3))
  )
  ...
.)
```

4.6.4 Symbol Pins Definition

Symbol pins define the locations on the symbol where electrical connections can be made. The syntax is:

```
(SYMBOLPINS
  <(pinname (xcoord ycoord) pintype)>*
```

Pinname is the name given to the pin that will be created at the specified location. The pin has a **pintype** which may be one of **IO**, **INPUT**, or **OUTPUT**.

Example:

```
(device npn
  ...
  (symbolpins (b (0 0) 'io)
               (c (4 5) 'io)
               (e (4 -5) 'io)
  )
  ...
)
```

4.6.5 Automatic Layout Definition

The automatic layout section of the device command is a list of LISP expressions which are evaluated sequentially. These LISP expressions are a procedural description that describe how to create the device. The syntax is:

```
(AUTOLAYOUT <lispexp>*)
```

Example:

```
(device npn
  (autolayout
    ; Define local variables
    (localvar emipointers emilist npnlayout basewidth)

    ; Create all emitter rows and store them on emipointers
    (do$ ((i 1 (+ i 1)))
      ((> i (propref 'rowcount)))
      (setq emipointers
        (append emipointers
          (npnemi i
            (propref 'size)
            (propref 'emitype)
            (propref 'rowcount)
          )
        )
      )
    )
    ; Step through all emitter rows and find the widest one.
```

```

; This allows the width of the base to be determined.
(setq basewidth
  (+ (apply 'max (mapcar 'findwidth emipointers))
    (* 2 (encl 'p+ 'n+))))

; This loop steps through all created emitter rows. On
; the first emitter row the npnlayout is set to the
; emitter row itself. Each subsequent emitter row has a
; base contact placed between itself and the previous
; emitter. After stepping through all emitter rows,
; the layout is surrounded by a polygon of p+, and a
; polygon of n+.

(do$ ((emilist emipointers (cdr emilist))
      (currenti (car emilist) (car emilist)))
  ((= emilist nil)
    (gensurround 'epi
      (gensurround 'p+
        npnlayout
        '((purpose npnbase))
        (noderef 'b))
      nil
      (noderef 'c)))
    (cond ((= currenti (car emipointers))
      (setq npnlayout currenti))
      (t (placebelow npnlayout
        (npnbase basewidth)
        '(on p+)))
    )
  )
)
...
)

```

4.6.6 Hierarchy Proposal

This section defines the netlist pattern matching that determines which devices should be grouped together in a new level of hierarchy to allow efficient layout implementation. The most likely application of this hierarchy creation in bipolar design is the grouping of devices together into a single epi region (See Figure 6 on page 79.)

This grouping can save large amounts of area, but is also inherently risky in that any device in the epi region which enters saturation or is biased incorrectly can cause the entire circuit to malfunction. To minimize the risk, a check is performed on all devices that potentially could become part of a grouping to ensure that their operating points will not cause problems. The syntax is:

```
(HIERPROPOSE <lispexp>*)
```

Example:

```
(device epitub
  ...
  (hierpropose
    ; The following lisp expressions traverse a netlist and
    ; select the PNPs that have a common base potential.
    ; These PNPs are individually tested for entering
    ; saturation during the template simulation. If they do
    ; not enter saturation and have a common base potential,
    ; they may be grouped into an epi region.

    (localvar pnpepi pnptub)

    (do$ ((nodelist (getnodelist) (cdr nodelist))
          (node (car nodelist) (car nodelist)))
      ((null nodelist))
      (setq pnptub nil)
      (do$ ((pinlist (getpinlist node) (cdr pinlist))
            (pin (car pinlist) (car pinlist)))
          ((null pinlist))
          (& (eq (getpinname pin) 'b)
              (eq (getdevtype (getdesig pin)) 'pnp)
              (~saturatep (getdesig pin))
              (progn (setq pnpepi (getnode pin))
                      (setq pnptub
                          (append (list desig) pnptub))))))
      )
    (& (listp pnptub)
      (> (length pnptub) 1)
      (maketubdesc pnptub pnpepi))
  )
)
```

4.6.7 Hierarchy Constraints

This section defines the procedure to test whether a proposed grouping of schematic devices into a new level of hierarchy in the layout is valid. The grouping is proposed either automatically, or by the user from the schematic editor. (See Section 5.5) The syntax is:

```
(HIERCONSTRAINTS <(boolexp errmsg)>*)
```

Where boolexp is a LISP expression that checks the proposed grouping. If boolexp returns a non-NIL value then errmsg is the textual error message that will be output.

A simple example which tests whether two PNPs may be placed in the same epi region follows. The check verifies that the base potentials of the PNPs are the same and that none of the transistors enter saturation. The saturation check is performed by examining the output of the simulation template.

Example:

```
(device epitub
  (hierconstraints
    ; Verify that all selected devices are PNPs
    ((mapcan '(lambda (x) (neq (devtype x) 'pnp))
      (schemselected))
      "All selected devices must be PNPs"))
    ; Base potential of all PNPs must be the same
    (((lambda (l) (mapcan '(lambda (x)
      (neq (devnode (car l) 'b)
        (devnode x 'b))) l))
      (schemselected))
      "PNP base nodes must all be at the same potential"))
    ; Make sure that none of the PNPs enter saturation
    ((mapcan '(lambda (x) (saturation x)) (schemselected))
      "No devices may enter saturation"))
  )
  ...
)
```

4.6.8 Layout Constraints

This section defines the constraints on the layout of a device. These constraints are associated with the device definition as properties and can be overridden by constraints placed on a particular device instance. These constraints could have been defined directly as default properties on the device, but the special syntax is clearer. The syntax is:

```
(LAYCONSTRAINTS <(constname (boolexp polylist errmsg))>*)
```

- Where `constname` is the name of the property under which the constraint will be placed and is followed by a list of error

triplets. If `boolexp` returns non-NIL when evaluated, then the polygons from `polylist` are copied onto the error layer with the error message `errmsg` associated with them.

The first example is a constraint that might be applied to an epi tub.

Example 1:

```
(device epitub
  ...
  (layconstraints
    ; This constraint prevents the voltage drop in a epi
    ; polygon from exceeding 0.4 V. This will prevent many
    ; parasitic transistors from turning on.
    (maxepidrop ((> (deltaVoltage (layerref 'epi)) 0.4)
                  (layerref 'epi)
                  "Voltage drop in epi > 0.4 V"))
    )
  ...
)
```

The second example is a constraint that might be applied to a bonding pad.

Example 2:

```
(device pad
  (layconstraints
    ; The first constraint is a generic constraint that
    ; applies to all pads - All pads must be on the border
    ; of the chip
    (padposition
      (~ (locationp self 'border)
        (layerref 'passiv)
        "Pads must be located on the border of the chip"))
    )
  )
  ...
)
```

4.6.9 Device Verification Definition

The device verification definition section defines what polygon configurations are invalid for a particular device. The syntax of this section is a list of error triplets where each triplet identifies a different type of error. The syntax is:

```
(DEVICEVERIFICATION <(boolexp polylist errmsg)>*)
```

If boolexp returns non-NIL when evaluated, then the polygons from polylist are copied onto the error layer with the error message errmsg associated with them.

Example:

```
(device npn
  (deviceverification
    ((notnode (layerref 'epi) (noderef 'c))
      (notnode (layerref 'epi) (noderef 'c))
      "All epi in a NPN must be connected to the collector")

    ((layerref 'dn+ '(\ (purpose epicon)))
      (layerref 'dn+ '(\ (purpose epicon)))
      "All dn+ in an NPN must have purpose epicon")

    ((notnode (layerref 'dn+) (noderef 'c))
      (notnode (layerref 'dn+) (noderef 'c))
      "All dn+ in an NPN must be connected to the collector")

    ((interact (layerref 'dn+) (layerref 'p+))
      (interact (layerref 'dn+) (layerref 'p+))
      "dn+ may not interact with p+ in an NPN")

    ((notsurround 'epi (layerref 'dn+))
      (notsurround 'epi (layerref 'dn+))
      "All dn+ in an NPN must be surrounded by epi")

    ((layerref 'n+
      '(\ (| (purpose 'npnem) (purpose 'epicon))))
      (layerref 'n+
      '(\ (| (purpose 'npnem) (purpose 'epicon))))
      "All n+ in an NPN must have purpose npnem or epicon")

    ((notnode (layerref 'n+) (noderef 'e))
      (notnode (layerref 'n+) (noderef 'e))
      "All n+ in an NPN must be connected to the emitter")

    ((notsurround 'epi (layerref 'n+))
      (notsurround 'epi (layerref 'n+))
      "All n+ in an NPN must be surrounded by epi")

    ((notsurround 'p+ (layerref 'n+ '(\ (purpose 'npnem))))
      (notsurround 'p+ (layerref 'n+ '(\ (purpose 'npnem))))
      "All npn emitters must be surrounded by p+")

    ((interact (layerref 'n+ (purpose 'epicon))
      (layerref 'p+))
      (interact (layerref 'n+ (purpose 'epicon))
      (layerref 'p+))
      "epi contacts may not interact with p+")

    ((layerref 'p+ '(\ (purpose 'npnbase)))
```

```

(layerref 'p+ ' (~ (purpose 'npnbase)))
  "All p+ in an NPN must have purpose npnbase")

((notnode (layerref 'p+) (noderef 'b))
 (notnode (layerref 'p+) (noderef 'b))
  "All p+ in an NPN must be connected to the base")

((notsurround 'epi (layerref 'p+))
 (notsurround 'epi (layerref 'p+))
  "All p+ in an NPN must be surrounded by epi")

((layerref 'contact ' (~ (| (purpose 'npnbasecon)
                             (purpose 'npnemcon)
                             (purpose 'npncollcon))))
 (layerref 'contact ' (~ (| (purpose 'npnbasecon)
                             (purpose 'npnemcon)
                             (purpose 'npncollcon))))
 (concat "All contacts in an NPN must have purpose "
  "npnbasecon, npnemcon, or npncollcon"))

((notsurround 'metal (layerref 'contact))
 (notsurround 'metal (layerref 'contact))
  "All contacts must be surrounded by metal")

((notsurround (| (layerref 'p+) (layerref 'n+)) 'contact)
 (notsurround (| (layerref 'p+) (layerref 'n+)) 'contact)
  "All contacts must be surrounded by n+")
)
...
)

```

This example is somewhat simplified in that it does not contain any checks to guarantee that an emitter or base actually exists. These checks could be added to this section, or alternatively the parameter comparison section could be used. The logical place is here in the device verification section.

4.6.10 Parameter Extract Definition

The parameter extraction definition defines the physical parameters that are to be extracted for a device. Once again the syntax is a list of LISP expressions which will be evaluated, but convention dictates that these expressions will set properties on the layout of the device. The syntax is:

```
(PARAMETEREXTRACT <lispexp>*)
```

For complicated parameters which cannot be computed in a small amount of computer time, the LISP expression for a particular parameter will be the setup processing to allow the parameter to be calculated as a background task. An example of such a parameter might be the 3dB rolloff point of a transistor which must be computed by calling a simulator.

Example:

```
(device npn
  ...
  (parameterextract
    (setprop 'size (/ (area (layerref 'n+
                           npn1xsize))
                      '(purpose 'npnem)))
    (setprop 'maxIbase
      (* (area (layerref 'contact
                    maxIbaseperu2))
        '(purpose 'npnbasecon)))
    (setprop 'maxIcoll
      (* (area (layerref 'contact
                    maxIcoilperu2))
        '(purpose 'npncollcon)))
    (setprop 'maxIem
      (* (area (layerref 'contact
                    maxIbaseperu2))
        '(purpose 'npnemcon)))
  )
  ...
)
```

4.6.11 Parameter Comparison Definition

This section defines how parameters are to be compared between layout and schematic. The syntax is:

```
(PARAMETERCOMPARE <(boolexp polylist errmsg)>*)
```

If `boolexp` returns non-NIL when evaluated, then the polygons from `polylist` are copied onto the error layer with the error message `errmsg` associated with them. This is another instance of the use of error triplets. A call to the function `compare` is often used as a basis for `boolexp`. The syntax is:

```
(COMPARE proptype comparisontype [comparisonvalue]>*)
```

Where `propname` is the property whose value will be compared between schematic and layout. `Comparisontype` is the type of comparison that will be performed. Valid entries are `tolerance`, `absdiff`, `extractedgreater`, or `extractedless`. `Comparisonvalue` is a number that is used to specify the amount of allowed tolerance when the `comparisontype` equals `tolerance`.

Example:

```
(device npr
  ...
  (parametercompare
    ((compare size tolerance .05)
     (layerref 'n+ '(purpose 'npnem))
     "Size of NPN not within tolerance"))
    ((compare maxIbase extractedgreater)
     (layerref 'contact '(purpose 'npnbasecon))
     "Base not able to support required current"))
    ((compare maxIcoll extractedgreater)
     (layerref 'contact '(purpose 'npncollcon))
     "Collector not able to support required current"))
    ((compare maxIem extractedgreater)
     (layerref 'contact '(purpose 'npnemcon))
     "Emitter not able to support required current"))
  )
  ...
)
```

CHAPTER 5. THE ALE METHODOLOGY

5.1 The Need for Incremental Layout

After a designer has created a circuit schematic and verified its performance, the schematic needs to be transformed into a layout. This layout must be a correct connection of the polygonal representation of the devices in the schematic. Unfortunately, there is often the need to update the schematic while layout is in progress. This need can arise due to:

- Circuit errors being discovered.
- Circuit requirements being changed.
- The demands of the particular layout implementation requiring a change to the circuit schematic to maintain a correspondence between schematic and layout.

It is important that a custom layout methodology be able to gracefully deal with changes to the schematic, and not require the layout to be completely redone. This is acceptable in some layout methodologies because the layout is produced by a completely automated process with no manual editing for further optimization. These layout methodologies are used when the final IC production cost is relatively unimportant compared to design turnaround time.

In custom layout the inverse tradeoffs are often made because final IC production cost is much more important than design turnaround time. For this reason, any automatic layout done in custom layout may be manually optimized to recover the layout area that automated algorithms are not currently able to salvage. Completely regenerating a portion of layout when the schematic is changed is usually not acceptable because of the degree of manual optimization that may have been invested in modifying the layout to perfection. Therefore, a requirement imposed on a custom layout methodology is that when a schematic is modified, the changes that need to be made to the layout to maintain its correspondence to the schematic be automatically maintained.

A schematic is a graphical representation of a circuit useful to circuit designers. A layout is a graphical representation of a circuit useful for IC manufacture. Both a schematic and layout represent the circuit but in different domains. For that reason, and because of the requirement for close coupling between schematic and layout, both schematic and layout editing are implemented in a single program in the ALE methodology.

5.2 Data Structure Summary

The data structure in which the schematic is stored is called a schematic tree. The corresponding data structure

for the layout is called a layout tree. Both are hierarchical data structures that represent the current view of the schematic or layout. The schematic and layout trees are coupled together to allow a correspondence to be maintained between them.

Each node in the schematic tree and layout tree represents a level of hierarchy in the schematic domain or layout domain. These nodes refer to information such as the current graphical representation of the node, the property list associated with the node, and the corresponding node in the parallel representation. Because the layout is not always implemented in exactly the same manner as the schematic due to reorganizations for layout efficiency, or because of schematic editing, there may not always be a corresponding node in the other representation.

For example, if a device is deleted from the schematic, the corresponding device in the layout is no longer required. One possibility would be to automatically remove that device from the layout. While this solution is simple to implement, it is highly undesirable from the point of view of layout designers who do not like their hard work deleted automatically. The layout device is therefore left without a corresponding node in the schematic, although a machine readable explanation is attached to the layout node that describes why there is no schematic node. The explana-

tion in this case is simply that the "schematic node was deleted."

Another case where they may not be corresponding nodes between layout and schematic is when the schematic has its hierarchy reorganized for the layout implementation. This is explained in the next section, but the same technique of attaching explanations to nodes with no corresponding nodes is used.

Full details of the data structures are left for subsequent work. This is not because of any great complexity but simply for brevity and because the details are somewhat irrelevant to the ALE concept. The object oriented paradigm allows these structures to be implemented once, and then used through an interface provided by the objects. The rest of ALE can then be implemented without any knowledge of the object's underlying implementation.

However, for efficiency reasons, the data structures for layout should take advantage of data concepts such as K-D trees or quad trees [24]. Mixing these tree structures with enhanced corner stitching notions [25] or constraint graphs [26,27] should also be explored as it could be a very effective way to speed compaction by incrementally maintaining a compaction data structure in parallel with the regular geographical data structure.

Details regarding storage of data between editing sessions are not discussed here and are not necessary as the file format is completely secondary to the methodology.

Briefly, a variety of techniques are possible:

- Commercial LISPs provide a capability for preserving data sets so that they can be subsequently reloaded. This is by far the simplest as it takes advantage of existing language functionality:
- Another simple technique is to output a LISP expression that describes the current database. This is fairly easy to implement at the cost of a large storage file with long read and write times.
- A custom binary format can also be implemented at the cost of substantial programming complexity, but resulting in a small storage file with short read and write times.

5.3 Schematic Editing

The process of schematic editing in ALE is similar to that of many other schematic capture tools. Devices are represented as symbols and interconnected with wire segments. Properties associated with the symbols may have their values displayed graphically on the symbol or queried textually.

ALE schematic capture uses the property inheritance scheme described in Section 3.1. Property inheritance allows the behaviour of default devices to be changed simply

by changing the properties on the library symbol. Properties on schematic devices typically capture information such as the size of a device, the maximum capacitance on a terminal, or the maximum current into a terminal. Properties such as the 3dB rolloff point of a transistor are well within the realm of possibility for capture and checking. During layout verification, devices from the schematic and layout can be compared for equivalence strictly on the basis of their properties.

ALE incrementally updates the layout domain as the schematic is being edited. Modifications are not made to the polygons of the layout but to the structure of the layout tree. As devices are added to the schematic, nodes that will contain the implementation for these devices are automatically added to the layout tree. This allows the layout tree to track the schematic tree and allows ALE to provide incremental connectivity checking abilities. These will be described in Section 5.5.

5.4 Schematic Simulation

Once a schematic has been captured, it is time for simulation. Since ALE does not include a simulator, an interface between ALE and circuit simulators must be provided.

There is a wide variety of simulators currently available with an equally variable set of input formats. In order

for one of these formats to be created by ALE, a system property is set to the name of a LISP function that will create the simulator input file, call the simulator, and postprocess the results into a standard format.

This LISP function first traverses the schematic and writes to the simulation file the interconnections of all devices and the properties needed to model them. Stimuli information, which describes how the circuit will be driven throughout the simulation is then taken from a text file and appended to the simulation file.²⁰ The LISP function then starts the simulator as a background task. After the simulation completes, the LISP function converts the simulation results into a standard storage format. This storage format is used when comparing different simulations for equivalence, and when analyzing the biasing of devices.

5.5 Coupling Between Schematic and Layout

Both the schematic and layout domains are hierarchical descriptions of the building blocks that compose the circuit and the nodal potentials that connect them. The upper levels

²⁰ Stimuli information can also be captured graphically, by creating symbols that represent sources or probes and capturing the connections of these sources and probes in a special level of schematic hierarchy that does not have a correspondence to layout. The behaviour of the sources and probes would be captured as properties on their instances.

of the hierarchy contain functional blocks, i.e. nonprimitive devices, and the lowest levels contain the primitive devices that comprise the functional blocks. Nodal potentials connect functional blocks and primitive devices across hierarchy boundaries.

The primary difference between the schematic and layout domain is that the schematic is constructed to allow the designer to visualize the circuit and to capture the circuit connectivity, and the layout is constructed to allow the circuit to be manufactured. To the user, the domains have different graphical representations and there may be differences in the hierarchy of the domains. However, all devices in one domain normally have some mapping into the other domain.

Nodal potentials directly correspond between the two domains, but the precise way in which the interconnections between devices is implemented is usually different. This is not significant from a basic connectivity point of view, but can become quite important for postlayout simulation when parasitic capacitances and resistances are used.

For primitive devices, there is usually a one to one mapping between the schematic and layout domain. For example, if a transistor symbol is selected in the schematic,

then a corresponding transistor can be selected in the layout.

Nonprimitive devices such as a flip-flop may be represented as a box in the schematic, with a lower level of the schematic hierarchy containing the actual transistor connections that make up the flip flop. In the layout, the flip-flop could be represented as a cell, corresponding to the flip-flop symbol, and the transistors inside the cell corresponding to the transistors in the flip-flop. This also provides a one to one mapping between layout and schematic, but the mapping is not always as straight forward.

In the flip-flop example, the layout designer could have decided that the layout could be created more efficiently by ignoring the hierarchy of the flip-flop, and laying out the flip-flop as though its component transistors were not at a different level of hierarchy.

The layout designer could also have decided that an extra level in the layout hierarchy would lead to a better layout. This might occur when two PNPs have the same base potential. Typically, these PNPs could be laid out in the same epi region and thereby save layout area. In the ALE methodology, this can be done by creating an extra level of hierarchy containing the devices that should be in the same epi region. This example is shown graphically in Figure 6.

Figure -6. Epi Region Merging: At the top are two PNPs in separate epi regions. At the bottom are the same PNPs in the same epi region.

These two examples illustrate that a one to one mapping between the schematic and layout domains does not always exist. However, in order for ALE to be able to create correct layouts, it must always be able to guarantee that the layout domain is equivalent to the schematic domain. Equivalence exists when there is a legal mapping between the schematic and layout domains.

To simplify enforcing the requirement that the layout domain and schematic domain be equivalent, a restriction on the differences between layout domain and schematic domain is enforced. There may be at most one extra level of hierarchy added or subtracted in the layout domain relative to the schematic domain. This restriction is quite arbitrary and is not strictly necessary, but does simplify matters while still permitting a substantial amount of flexibility. This restriction is illustrated graphically in Figure 7.

To communicate to ALE the fact that a flip-flop is to be implemented in a flat manner, the designer graphically selects the flip-flop and then selects the command:

Flat-Layout

This causes ALE to check whether the rules for layout to schematic equivalence would be violated and outputs a message if there would be a violation. If the proposed modification is acceptable, the layout hierarchy is then modified. The layout node for the flip-flop is deleted and its descendant gates instantiated in its place. This is case 4 of Figure 7 where B has been deleted and B's descendants have taken its place. The schematic node for the flip-flop now has no corresponding node in the layout, and this is noted with the explanation "Layout node flattened."

Alternately, the user may decide that the two PNPs should be grouped together in their own level of layout hierarchy. To communicate this to ALE, the designer selects both PNPs and then selects the command:

Create-Hier

This causes a form to be displayed for the user to fill out. The form and the appropriate responses are:

Layout Node Name: EPI1
Layout Node Type: EPITUB

This indicates to ALE that the name of the new level of hierarchy will be EPI1 and that its kind is an EPITUB. ALE first verifies that the domain equivalence rules would not be violated if the new level of hierarchy is created. An expert routine defined in the HIERCONSTRAINTS section of the EPITUB device_definition is then evaluated to ensure that the selected PNPs can legally be laid out in the same EPI region (Section 4.6.7). This is case 3 of Figure 7 where a new node G is created that acts as a parent to D and E. There is no corresponding schematic node for G and this is noted with the explanation "Layout node added."

The user cannot directly alter the layout tree with deletion and digitization commands in the way that the schematic tree can be modified during schematic editing. All changes to the layout tree must be performed indirectly through changes to the schematic or by the commands for creating and flattening hierarchy nodes in the layout tree. Preventing the user from directly modifying the layout tree is done because the schematic is usually intended as describing the design, while the layout is intended to implement the schematic. By preventing the user from directly modifying the layout tree, this design flow is enforced.

The one exception to this rule is deleting devices in the layout that have been previously deleted from the schematic. Because these devices had their correspondence to the schematic removed when the schematic symbols were deleted, there must be a way to also delete them from the layout tree. This behaves as a special case of the delete command during layout editing.

Because changes to the schematic tree also change the layout tree, layout errors may be introduced that ALE will subsequently discover. The user must then modify or move items in the layout to correct these errors.

5.6 Library Devices

Library devices are predefined sections of layout that have a correspondence to a schematic symbol. Placing a library device in a schematic allows the predefined layout to be used, and prevents the automatic creation.

When a device is added to the schematic, all of its corresponding lower hierarchy elements are also automatically added if the device is not a library device. If a user has an opamp, which is not a library device, and the opamp is duplicated several times in the circuit, the opamp's structure will also be duplicated in all of its instances. This replication also occurs in the layout tree. Therefore, the opamp's layout will be created repeatedly in

each of its instances. This is often desirable for optimum layout compactness, but not in terms of storage and compute efficiency. In order to have the opamp laid out only once, it is necessary to layout its structure independently, and make it a library device.

When ALE produces a layout tree, it uses library devices if they exist. If the user needs to customize a library device in a particular location then a single command will copy the library device into a new nonlibrary device that the user can then customize.

5.7 Automatic Layout Generation

Figure 8 shows a flowchart that describes the procedure ALE uses in automatically generating a layout. These steps will be explained briefly in this section and in greater detail in Section 6.0.

- If the device being laid out is not a primitive device, and the lower level devices are not yet laid out, ALE performs area estimation of the lower level devices.
- Based on these estimates the layout is floor planned from top level down to lowest level. The floorplan contains relative positions of devices, and approximate positions of inputs and outputs.
- The layout is then created in a bottom up manner using the floorplan information.

- The floorplan is manually adjusted if required.
- The low level devices of the hierarchy are then automatically laid out using parameterized device descriptions.
- The created devices are then placed using a quadratic placement algorithm based on a wire length metric and simultaneous solution of a constrained set of nonlinear equations.
- Because the placement algorithm does not guarantee complete design rule correctness, a compactor repairs any violations.
- A maze router automatically routes the placed devices as well as possible [28]. Unfortunately, maze routers are not guaranteed to complete 100% of the routing. Therefore the user may have to perform some manual editing to finish the layout.
- Since the router potentially introduces spacing violations when squeezing routing lines through constricted layout areas, the compactor is automatically run to repair these violations.

5.8 Interactive Editing

At this point, the automatic layout phase is finished. If the layout is incomplete or unsatisfactory, the user must intervene, interactively with editing commands. Most of the editing commands provided in ALE are typical of many layout editors. Other commands incrementally utilize the automatic

layout algorithms to implement powerful editing capabilities. The set of editing commands provide the ability to:

- Reshape polygons or centrelines
- Delete polygons or centrelines
- Digitize polygons or centrelines
- Alter properties on polygons
- Shift devices in the layout by incrementally running the quadratic placement algorithm
- Compact specific areas of the layout to remove design rule violations or remove extra area
- Route signals between individual devices by using the router

Whenever a layout change is made, the connectivity of the layout is incrementally updated. All polygons and centrelines in the layout refer to a list of connections. These connections electrically connect two geometrical entities together. For example, if a metal polygon overlaps a contact polygon, a connection will be created that points to these two polygons. The connection defines that the two polygons are connected together, but does not propagate nodal potentials between the polygons. Therefore, it is possible to have two polygons that have a connection between them but also have different nodal potentials. The ALE connectivity checks will subsequently detect this problem,

2 of/de 2

MICROD

(among other types of connectivity errors) and output error markers into the layout.

Connections can cross hierarchy boundaries if needed. This is difficult to implement, but provides the required flexibility. Normally, layout methodologies insist that electrical connections which cross hierarchy boundaries be formalized into pins. ALE has no such restriction. The physical mask regions of the IC know nothing of pins, and will connect together when overlapped regardless of which level of hierarchy the polygons originally came from. The description that defines the configurations that will cause connections between mask layers is defined in Section 4.4.

5.9 Error Detection

To allow a correct layout to be produced, any changes made to the layout can be checked by ALE for correctness. This error checking is initiated whenever ALE automatically creates pieces of layout, but can also be initiated by the user entering a command.

ALE incorporates automatic error detection to allow fast interactive correction when:

- The user makes an error
- ALE is unable to complete a layout and requires user assistance.

All detected errors are stored as graphical objects as well as a textual explanation that describes the error. The combination of text and graphics greatly simplifies the user's task in isolating problems.

To allow old errors to be automatically removed when errors are corrected, the error graphics need to be associated with the object that is in error, and the type of verification that discovered the error. For example, if a polygon is discovered to have a DRC violation associated with it, then an error marker is added to the layout, but a pointer to the error marker is also added to the polygon so that when a DRC check is subsequently performed on the polygon, the old error marker can be removed.

This concept of error pointers also applies to device verification errors, device parameter errors, and all other device errors. However, these errors are associated with devices rather than polygons and are therefore attached to nodes of the layout tree. Whenever error checks on a device are redone, the old error markers on the layout node are first removed.

ALE detects several major types of errors. These are:

- Design rule check (DRC) errors. These are errors of spacings, widths, overlaps, enclosures, etc., between mask layers. A layout automatically produced by ALE will never

contain DRC errors, but when the user modifies the layout, the danger always exists that errors will be introduced.

- Device construction errors are errors introduced when a user modifies a structure and it no longer is a valid representation of the desired structure. For example, if a user moved the emitter of an NPN out of the base of the NPN as shown in Figure 9, the structure would still be design rule correct, but would no longer be a valid NPN.
- Connectivity errors occur when the existing layout does not correspond to the provided netlist. This can be due to missing connections that would occur if ALE were not able to complete the routing of a signal. Alternately, when a user modifies the connections on devices in the layout, or changes the circuit schematic, errors may also be introduced.
- Parameter errors are errors in comparing the extracted parameters of a device in the layout with the parameters attached as properties to the corresponding device in the schematic.
- The last major type of error that ALE detects is a constraint violation. ALE permits the user to specify a wide variety of constraints that must be satisfied in order for the layout to be acceptable. Some examples of the type of constraints that might be added are listed in Section 5.10.

Error checking is performed incrementally when the user requests a check. Only objects which have changed need to be checked. For all checks other than design rule checking, this is performed on a device basis. If the device, or any of its subdevices have been modified, then all the checks that are associated with each modified device are redone.

Design rule checking is done differently. Whenever an area is modified, the altered polygons are added to a list of shapes requiring checking. When the user requests a check, this list is traversed and the design rule checks performed.

Unfortunately, the polygons that have been modified and are on the list are not the only ones that require checking. The interaction of these polygons with unaltered polygons in the layout needs to be checked as well. To allow this to operate in a reasonable manner, the design rule checking process must share the ALE internal data structure.²¹

This sharing of data structures is not strictly necessary as all shapes that must be included as part of a check

²¹ The DRC is typically performed as a background task and because the DRC process can be interrupted at any time by a user command, the DRC process must protect itself from corruption during the short periods when it is accessing the database by locking out all interrupts. This will prevent a user command from modifying data that is in the process of being accessed by the error check.

can be determined as part of the DRC setup cost. However, the setup cost would be quite significant and would be very noticeable to the user because no commands could be entered during this setup process. Therefore, if possible, these computations should be deferred to the actual background task.

Many workstation operating systems provide for this sharing of data between processes. However, the problem of data that is being verified suddenly changing due to user ~~commands~~ must be addressed. In other ALE background processes this is solved by copying the background process data from the ALE database to a separate area. For design rule checking, which shares the ALE data structure, the procedure is also quite simple.

Whenever a polygon is modified by a command,²² it must subsequently be checked and therefore must be added to the list of shapes that need to be verified. If the polygon is already part of the list of entities waiting to be checked, then nothing need be done, as the polygon will be checked due to its previous membership in the list.²² When a polygon is deleted from the layout, then nothing need be done unless

²² This requires attention in programming to ensure that when polygons are modified, that they can still be referred to by the same database entry.

the polygon is already a member of the DRC list. If it is a member, then it is removed.

5.9.1 Device Correctness

In polygon editing layout methodologies, verifying that the layout of individual devices is correct is left for a batch verification which also performs design rule and connectivity checks. The cycle time for modifying the layout and getting feedback from the verification will typically take a substantial amount of user time and computer time, and is therefore a slow cumbersome process.

Alternately, symbolic layout tools attempt to guarantee device construction by automatically producing polygonal representations. These polygonal representations cannot be edited without losing the symbolic representation. This prevents completely customizable devices and is a significant drawback.

ALE eliminates the cumbersome behaviour of batch verification programs by performing error checks incrementally, while still permitting the user to modify the layout of devices in a completely arbitrary manner. The NPN in the upper half of Figure 10, may have been created automatically by ALE but by using polygon editing commands, the user could modify this NPN until it looks like the lower half of Figure 10. ALE automatically verifies that the NPN

still represents a valid NPN by using a description of invalid polygon configurations that is read from a technology file. An explanation of the syntax, and an example is presented in Section 4.6.9.

The correct properties must be assigned to all polygons in order to be able to ensure device correctness. For example, all the polygons on the n^+ layer in the NPN must have the property "purpose" defined. The permissible values of this property are "npnem" or "epicon." If the properties are not defined properly, device verification errors will be detected by ALE.

Device verification can also include a first stage of voltage parameter checking. In some bipolar technologies, there are different design rules depending on what voltage a device must tolerate without breakdown. In order to accommodate this, properties are attached to devices that define the required voltage level. Device verification can then ensure that all the polygons in a device have the appropriate voltage properties based on the voltage property attached to the device itself.²² Design rule checks which use the voltage property from the polygons can then verify that the polygons are spaced correctly.

²² Verifying that the voltage properties on the devices are correct can be performed as part of device verification by examining the output of the simulation template.

5.9.2 Device Parameter Correctness

Once a particular device has been verified as being constructed properly, the parameters on that device can be verified. This is simply a matter of executing the parametercompare section on the definition of the device. If errors are discovered, appropriate error markers will be automatically output into the layout.

5.9.3 Connectivity Correctness

ALE tracks the differences in connectivity between the netlist and the layout, and can inform the user of the differences at any time. This is possible because of the layout tree and its correspondence to the schematic. Because the layout tree directly represents the schematic, when the layout tree has been completely populated with geometries, then by definition the entire circuit has been laid out.²⁴

The difficult parts in layout to schematic verification are:

- Determining which layout devices correspond to which schematic devices.
- Determining whether all the devices from the schematic have been instantiated in the layout.

²⁴ Some extra device may exist in the layout due to deletions from the schematic that were not done in the layout.

- Determining whether there are devices in the layout that do not exist in the schematic.

ALE completely eliminates these problems by using the layout tree. This is because devices in the layout tree directly correspond to devices in the schematic. There is no possibility of there not being enough devices in the layout as the layout tree can never be a subset of the schematic tree. This is because the layout tree has been automatically constructed from the schematic tree. However, because devices that have been deleted from the schematic may still be in the layout, there is the possibility of there being extra layout devices. Traversing the layout tree and determining which layout devices have no corresponding schematic device and whose reason for not having a correspondence is "Schematic node was deleted" are in error and have an error marker output to that effect. All other layout devices are guaranteed to either have a direct correspondence with a schematic device or a valid reason for not having a correspondence (Section 5.5).

ALE incrementally updates its internal connectivity description whenever a layout change is made but does not propagate nodal values from interconnect into devices. When devices are constructed properly, the nodal values are guaranteed correct. Allowing devices to take on the nodal values of the interconnect would be throwing away informa-

tion that is known to be correct, and is therefore not done. Ensuring that devices have been constructed properly and have the correct nodal values is performed by device verification.

Because of these simplifications, the connectivity checking process degenerates to one of checking that the connections between devices are correct. This can be easily done by traversing all the electrical nodes in the layout and making sure that they are not split. A split node is one that is implemented in more than one piece and is symptomatic of a node that was not completely routed. A check also needs to be made to ensure that interconnect does not connect devices at different nodes. This is also easily done because of the connection data structure which connects interconnect and devices together and was briefly described in Section 5.8. If the connection connects different nodal potentials together, then it is invalid.

5.10 User Constraints

A powerful ALE capability allows the user to specify constraints on the layout. A layout may be design rule correct, device construction correct, and connectivity correct, but still fail to function correctly. Assuming that the reasons for circuit failure are not due to incorrect simulation or modelling as these issues are not addressed

here, the problems are due to other differences between the layout and schematic. These differences are not in the connections of devices, as these are guaranteed correct by the connectivity checking, but are due to parasitic resistances, parasitic capacitances, parasitic devices, and device matching errors.

Every piece of diffusion or interconnect has resistance and capacitance associated with it. Interacting pieces of diffusion often create parasitic devices such as diodes and transistors. The schematic entered by the designer does not capture any of these parasitic devices and they are ignored when checking connectivity. However, since these parasitics can potentially affect circuit operation they must be checked before a layout can be determined to be error free. These checks are performed in two ways. The constraint system described in this section is one method. The other way is through simulation verification and is described in Section 5.12.

The constraint system provides the ability to specify limits on the amount of resistance between two parts of a node, on the amount of capacitance on a node, among others. ALE monitors these constraints as layout progresses, and will flag violated constraints as errors. During automatic layout, ALE considers some of these constraints and attempts to create layouts that do not violate them.

The constraints in this section are expressed as LISP expressions that represent how the constraints are stored. The user interface to create these constraints is based on a form filling system and is much more user friendly. Some examples of constraints and their LISP description follow.

- The bonding pads of an IC are the regions through which connections from the IC to the outside world are made. Bonding pads typically are placed on the periphery of the layout. As well, the side of the layout that the bonding pads will be positioned on, and the relative position of a bonding pad on a particular side is often predefined.

Example:

```
; Constraint limits the position of a specific pad
; to the left border of the chip.
(padleftside
  (¬(locationp self 'leftborder)
    (layerref 'passiv)
    "Pad must be located on the left border of the chip")
)
```

```
; Constraint restricts the position of a pad to be over
; another pad
(padrelpad2
  (¬(abovep self 'pad12)
    (layerref 'passiv)
    "Pad must be located over pad12")
)
```

- The resistance of an epi tub between the epi contact to the epi region just below the emitter of a particular NPN must be below a certain value to control transistor parameters.

This type of constraint is extremely difficult to verify because the topology of an epi region can be very arbitrary. The epi tub in Figure 11 shows some of the difficulties. In order to compute the resistance between the epi contact at point A to the epi region just below the NPN emitter at point B, requires the processing of the resistance of a curving path with regions of varying conductivities. One technique that can perform this computation is the method of finite elements.²³

Example

```
; This constraint limits the resistance between
; the epi contact of an epi tub and the base of an NPN
; transistor within that epi tub.
(epibaser
  (> (resistance
      (layerref 'n+' (purpose epicon))
      (layerref 'p+' (purpose npnbase) (device t2)))
    500)
  (append (layerref 'n+' (purpose epicon))
    (layerref 'p+' (purpose npnbase) (device t2)))
  "Resistance from epi contact to base of t2 > 500 ohms")
)
```

- A similar constraint might be that the maximum voltage difference between any epi terminals in an epi tub must be below a certain value. This would allow detection of

²³ Computing resistances through finite element analysis is very computationally intensive. To prevent this from affecting the interactive nature of ALE, these tasks are processed in background as described in Section 3.5.

an epi tub in which there is a significant danger of parasitic transistors turning on.

In order to implement this type of constraint, all terminals in question must be annotated with their maximum current usage. This permits the maximum possible voltage differences between particular nodes to be computed if the resistance between them is known from the previous finite element analysis.²⁴

Example:

```
; This constraint limits the voltage between
; the epi contact of an epi tub and the base of an NPN
; transistor within that epi tub.
(epibasev
  (> (voltage
      (layerref 'n+ '(purpose epicon))
      (layerref 'p+ '(purpose npnbase) (device t2)))
    0.4)
  (append (layerref 'n+ '(purpose epicon))
    (layerref 'p+ '(purpose npnbase) (device t2)))
  "Voltage drop between epi contact and t2 base > 0.4 V"
)
```

- The maximum coupling capacitance between two nodes required to be below a certain value is an example of another important type of constraint. In sensitive

²⁴ This constraint can also be checked through a constraint which examines the output of postlayout simulation. However this requires a simulation to be available which implies that the layout has already been completed. Being able to check a constraint of this nature while layout is progressing is extremely important to minimize layout changes at a later date.

analogue circuits, capacitances of this type can couple noise into nodes and affect the performance of the circuit.

Capacitance between two nodes is of two kinds, vertically between mask levels at different potentials, and horizontally between mask layers at different potentials. Computing these capacitances involves summing many small internodal capacitances to achieve a lumped value that can be compared against the constraint.²⁷

Example:

```
(caplim
  ((< (capacitance self (noderef 'n2)) 10E-12)
    (append (layerref '* (noderef self))
            (layerref '* (noderef 'n2)))
    "Capacitance between nodes too high")
)
```

- Input stages of op-amps generally require close matching to minimize offset voltages. To achieve this close matching, it is important that the transistors in the input stages are placed closely together. A user constraint limiting the permitted distance between transistors may be specified.

²⁷ This is one of the few constraints that cannot be attached to devices in the schematic but have to be attached to nodes in the schematic.

Example:

```
; Constraint restrains position of a device to be within
; 100 u of another device.
(devdist
  (((< (distance self (device t3)) 100)
    (append (layerref 'epi '(purpose epicon))
      (layerref 'epi '(purpose epicon) (device t3)))
    "Devices greater than 100u apart")
  )
```

5.11 Postlayout Simulation

After the user is satisfied with the layout and ALE no longer detects any errors, ALE can produce a netlist for simulation. This netlist contains parameterized devices, as well as parasitic resistances, parasitic capacitances, and parasitic devices such as transistors and diodes. Extraction of capacitances and resistances is performed using the same techniques as in constraint checking. Parasitic device extraction is performed by user written extraction routines.

Using tailorable heuristics defined in the simulator interface, many of the insignificant resistors and capacitors that will not affect circuit operation can be automatically pruned. This pruning is necessary both to save simulation time, or as is often the case, to avoid exceeding the capacity of the simulator. This pruning is performed by comparing the extracted resistances and capacitances to minimum values and discarding those that fall below the minimum. The minimum values of resistance and capacitance

for a particular node are based on the category of the node. Typical categories might include sensitive, moderately sensitive, insensitive, low current, medium current, and high current.

Simulating parasitic active devices is more difficult than inactive devices such as capacitors and resistors. Modelling active devices requires much more computer time than inactive devices. As well, many simulators have a much lower limit on the number of active devices that can be simulated than inactive devices. These problems makes pruning of active parasitics imperative. However, pruning of parasitic active devices is essentially impossible without precise knowledge of the voltages that will stimulate these devices.

The solution chosen in the ALE methodology is to:

- Prune all active parasitics that have not been marked as potentially dangerous,¹¹ and heuristically prune the inactive parasitics from the simulation netlist.
- Thoroughly simulate the pruned netlist.
- Postprocess the simulation output while checking all active parasitic devices for entering a mode in which they might affect circuit operation, i.e. check to see

¹¹ This marking may be performed manually by the user, or automatically by a later step.

whether any of the parasitic transistors or diodes could have turned on or entered a breakdown mode at any point in the simulation. All of these potentially dangerous parasitics are marked as such so that when the simulation is redone, they will become part of the simulation.

- If any new parasitics have been marked as potentially dangerous, inform the user and resimulate the layout.

Many parasitic problems are not due to a single parasitic device, but due to a chain of parasitics turning on. ALE is capable of dealing with only one parasitic in a chain at a time but if the recommended procedure is followed, ALE will eventually resolve all parasitic problems. For example:

- The first time an extracted netlist is simulated and postprocessed, some parasitics will be marked as potentially dangerous.
- Resimulating the netlist with these parasitics will allow the detection of parasitics in the next link of the chain.
- Repeating the simulation and parasitic marking phases until no more parasitics are marked will discover all parasitic chains.

This is a computationally intensive loop to perform, but there are certain things that make it more palatable. By far the most common parasitic in bipolar circuits is a single parasitic diode or parasitic transistor turning on.

This limits the amount of repeated simulation to process the parasitic chain because the chain is only one or two links long. Secondly, because active parasitic devices prevent a design from functioning properly, virtually any cost in discovering these problems is worthwhile.

5.12 Simulation Verification

The results of simulating the extracted layout with the simulation template can be automatically compared with the results of simulating the circuit schematic with the same simulation template. This allows ALE to perform a final check on the behaviour of the layout.

5.12.1 Waveform Comparison

The output of a simulator is usually a set of tables of a swept independent variable versus the resulting dependent variables.²² These tables represent waveforms and are transformed by the ALE simulator interface into an ALE waveform format which can be used during waveform comparison.

Because a simulation template will usually contain multiple sweeps of variables, the waveform format must take

²² The independent variable is usually time, but could also be frequency or some parameter such as resistance. The dependent variables are usually voltage or current, but other representations such as distortion could also occur.

this into consideration. As well, since the simulator template contains information describing how the outputs of the simulations are to be compared, this is also written into the waveform file. The syntax of the waveform file is:

```
(WAVESET wavesetname
  (PROPERTIES (SOURCE LAYOUT|SCHEMATIC)
    [(propname propval)]*)
  <waveform>*)
```

Wavesetname is a name given to the set of waveforms and is usually the name of the simulation template. Any number of properties can be associated with the set of waveforms, with one mandatory property. This property is **SOURCE** and indicates whether the waveforms originated from a schematic simulation or from a postlayout simulation.

The description of a particular waveform has the following syntax:

```
(WAVE (NAME wavename)
  (VARIABLES <(varname DEP|IND units)>*)
  [(PROPERTIES <(propname propval)>*)]
  [(COMPARE <compareexp>*)]
  (DATA <(depvarname depvarval <indvarname indvarval>*)>*))
```

Wavename is the name given to this particular waveform. **VARIABLES** indicates that a list defining the valid variable names in this waveform follow. **IND** indicates that the variable is the independent one. Only one independent variable is allowed in a waveform. Units is a piece of text which

describes the units of varname with typical values of usec, V, or mA.

A waveform may also have a list of properties and values associated with it. There are no mandatory properties on a waveform, but the capability is provided for user extendibility or for use in the waveform comparison.

COMPARE precedes a list of LISP expressions that will perform the waveform comparison. The LISP expressions are completely general. and Section 7.2 indicates some enhancements that might be made to the general concept of simulation verification. A typical compareexp might look like:

```
(testwavedist (wavevector 'VOUT 'LAYOUT)
              (wavevector 'VOUT 'SCHEMATIC')
              1E-6
              1E-3)
```

Wavevector is a wave lookup function which can index into a waveform based on the varname and ORIGIN passed as its two arguments. A list of pairs is returned with the following format:

```
(<(indvar depvar)>*)
```

These pairs of indvar and depvar represent the values of the independent variable and the dependent variable being looked up. A typical list of pairs might look like:

```
((0.0 1.0) (1E-6 1.1) (2E-6 2.5) (3E-6 2.6))
```

Testwavedist is a LISP function which takes two waveform vectors, measures the distance between them and returns a list of ranges where the distance between the waveforms exceeds the specified value.

Because the units of the independent variable and dependent variable will almost always be different, the definition of distance between waveforms is defined specially. In Figure 12 there are two kinds of distances indicated, the distance between waveforms on the independent axis (Time) and the dependent axis (Voltage). The distance between the waveforms is taken as the minimum of these distances and is called the independent distance or dependent distance depending on which value was minimum.

There are two values specified in the testwavedist function call. The first of these is the maximum independent distance, and the other the maximum dependent distance. The measured distance is compared against the appropriate maximum value to determine whether the waveforms are equivalent at that particular point.

This is not the only way to determine the equivalence of waveforms, but is one of the simplest. The generic LISP interface for the comparison allows any style of comparison to be implemented as needed.

5.12.2 Simulation Errors

After the waveforms have been compared, a list of differences will have been accumulated. This can be output to the user as a graphical display that shows the areas of the waveform that differ. The user can then manually determine what caused the problem, or ALE can make an attempt at discovering the problems.

ALE's ability to isolate these problems is based entirely on the layout having no design rule errors, device construction errors, connectivity errors, or parameter errors. This is guaranteed once all ALE detected errors have been fixed (Section 5.9). At this point the only significant differences between schematic and layout are due to parasitics. There are many things that can be done in determining which parasitic circuit elements are causing the differences. Only a few of the simple ones will be briefly described here.

- If a node that is in error has an active parasitic on it that has turned on during the time span of the error, then the active parasitic is possibly the problem.
- In the schematic, a node is represented as a single potential. In the layout, the node contains distributed resistances and capacitances which can make the potential of the node vary from one point to another. If the difference in potential within a single node is greater

than the maximum allowable voltage difference, then the distributed RC on the node is possibly the problem.

If simple procedures like these cannot isolate the problems then expert routines that attempt to understand the behaviour of the circuit are certainly possible. This has not been explored in this thesis work, but is an intriguing research subject.

CHAPTER 6. AUTOMATIC LAYOUT

This chapter describes the automatic layout process that was outlined in Section 5.7.

6.1 Automatic Floor Planning

Floor planning is the process of determining approximate sizes and positions of devices in all levels of hierarchy, determining the approximate positions of inputs and outputs (terminals) at all levels of hierarchy, and determining a global routing plan [29,30,31]. Floor planning information can be created automatically or manually by the user. All floor planning information is stored as constraints on the layout.

An example will be used to allow a proper explanation of the floor planning process. Figure 13 shows the schematic of a simple chip and Figure 14 contains the corresponding layout tree. The top level of the chip contains two nonprimitive devices, A and B, as well as four primitive pad devices, PAD1, PAD2, PAD3, and PAD4. Nonprimitive device A contains transistors A1 and A2, while nonprimitive device B contains transistors B1, B2, and B3. The layout for device B has already been created, either by manual editing or by previous automatic layout.

The floor planning task is to create a floor plan which describes approximately where A, B, and the four pads will be placed in the top level. The locations of A1 and A2 in A must also be defined, as must the position of the inputs and outputs on A. Because B is already defined, no definition needs to be performed, but the position of the inputs and outputs on B must be respected.

The first step in floor planning is to estimate the area of all blocks which have not yet been laid out. The area of lower level blocks must be determined before the area of higher level blocks can be computed, and therefore area is computed in a depth first manner.

In a recursive manner the area of all the subelements of A must be found before the area of A can be found. All elements of A are primitive devices which allows their areas to be determined either heuristically or by actually laying them out. For simplicity in this discussion, the areas are computed by invoking the automatic layout routines on the devices, and returning their area and aspect ratio.

Based on the areas and aspect ratios of A1 and A2, an estimate of the area of A can be made. One simple way is to take the total area of all subdevices and multiply this by a factor to account for the wasted area used by routing inter-

connections. After predicting the area and aspect ratios of the pads in the top level, all devices placed in the top level will have predicted areas and aspect ratios. The next task is to determine relative positions of all devices. This procedure is recursive and begins at Chip, the top level.

Relative positioning is performed by using a quadratic placement algorithm. This algorithm is explained in Appendix B, and is also used for final placement. The quadratic placement algorithm uses nonlinear programming to find a solution and thus has the capability of explicitly dealing with many specified constraints. The resulting placement minimizes a cost function that includes wire length, overlaps, and constraint violations.

This placement phase also identifies where terminals should be placed on lower level blocks that have not yet been laid out. This is determined by examining the placement to discover where the connections of each node must eventually be routed, and heuristically determining the positions on the devices that would minimize the final routing length. This input and output positioning information is added as constraints to the lower level block.

After all lower levels of hierarchy have been traversed, the initial floorplanning is complete. More optimization can be performed through a stage of successive

refinement. Briefly, this could take the results of the first floor planning and use it as a starting point for subsequent floor planning.

6.2 Manual Adjustment of the Floor Plan

After the floor plan has been automatically created, the user can adjust it as desired. This is implemented by graphically displaying the floor plan and allowing the user to modify it with editing commands. For example, estimated areas, aspect ratios, relative positioning of blocks, locations of terminals on lower level blocks could all be modified.

After user modification, ALE can take the new floor plan and apply successive refinement to it. This allows the user to make coarse changes such as moving a device to the other side of the layout and allow ALE to compute the consequences. An example of the completed floor plan for the example of Figure 13 is shown in Figure 15.

Once a floor plan is created, the layout process can begin. This is done in a bottom up process, starting at primitive devices and attempting to respect aspect ratio restrictions and input output location restrictions.

6.3 Primitive Device Generation

The part of the technology file which defines how devices can automatically be created is the automatic layout section of the device command. This contains a list of LISP expressions which are evaluated sequentially and together represent a procedural description that describes how to create the device. An example is shown in Section 4.6.5. This example describes the construction of a NPN which may have multiple rows of emitters interspersed with base contacts. The emitter rows may be constructed from strip emitters or separate emitters. This NPN has been simplified for illustration purposes so that there is no epi contact, connections between the rows of the emitters, or an associated epi region. There is also no usage of floor planning information to determine where inputs and outputs should be positioned.

The NPN creation function begins by creating the individual emitter rows. These emitter rows are created by the LISP function `nbnemi` which uses as arguments, properties on the schematic symbol for the particular device being created. After the individual emitter rows are created, the maximum width of the rows is computed to allow the width of the base contacts to be defined. Then as the final step, base contacts are placed between the emitter rows to complete the device. Some examples of the possible NPNs that

can be constructed using this definition are shown in Figure 16.

A possible implementation for `nprnemi` and its related functions follows.

```

; Function npnemirow creates a row of NPN emitters.
(defun npnemirow (rowindex size emitype totrows)
  (localvar height width num rowlayout)

  ; Compute height, width, and number of nominal 1X emitter
  (setq height (+ (minwidth 'contact)
                  (* 2 (minencl 'n+ 'contact))))
  (setq width (+ (/ (minarea 'contact) (minwidth 'contact))
                 (* 2 (minencl 'n+ 'contact))))
  (setq num 1)

  ; If the emitter row is to be a strip emitter then the
  ; width of the emitter is simply a factor times the
  ; default width
  (cond ((eq emitype 'strip)
         (setq width (* width (/ totrows size)))))

  ; If the emitter row is to have separate emitters then
  ; each emitter is a 1x emitter and each row must have
  ; an integer number
  (t (setq num (roundup (/ totrows size)))
      (& (eq rowindex totrows)
          (setq num (- num (- (* num totrows) size))))))

  ; Create and place each emitter in the row beside the
  ; previous base contact
  (setq rowlayout nil)
  (do$ ((i 1 (+ i 1)))
        ((> i num) rowlayout)
        (cond ((= i 1)
                (setq rowlayout
                      (ncon width height (noderef 'e)))
                (t (placeright rowlayout
                               (ncon width height (noderef 'e))
                               '(on p+)))
                )
        )
  )
)

; Function ncon creates a n+ region with a contact inside it.
; Both polygons are associated with the passed nodal potential.
(defun ncon(width height node)
  (localvar n)

  (setq n (rectangle 'n+ width height node))
  (list n (bias n (- (minencl 'n+ 'contact)) node))
)

```


Some emitter rows produced by this implementation are shown in Figure 17. PNP transistors are substantially more complicated structures but can be dealt with in the same general manner. Some examples of PNP transistors are shown in Figure 18.

Figure 18. Some Examples of PNP Transistors

6.4 Nonprimitive Device Creation

Any devices whose structure is defined by the user during schematic editing, is a nonprimitive device. Because these devices are not predefined in the technology file there is no associated automatic layout description. A simple default automatic layout procedure is applicable to these devices.

Example:

```
(autolayout  
  (quadratic-place)  
  (maze-route)  
)
```

Another example is a nonprimitive device that was added to the schematic by using the hierarchy manipulation capabilities described in Section 5.5. The example described in that section was the grouping of devices into an epi tub. Because the EPITUB in this example is a predefined device, the automatic layout description is attached to the device in the technology file. This description will utilize the automatic placement and routing capabilities, but will also add whatever polygons are required to finish the layout. In this case, a n buried layer polygon and an epi polygon might be added.

The placement phase respects the floor plan and constraints previously created. The placement phase is actu-

ally successive refinement of the floor plan. The floor plan is a placement that was created using estimated information before the lower level devices existed. The actual placement modifies the floor plan to accommodate the shape and terminal positions of the actual devices.

Because the quadratic placement cannot guarantee that the placement will meet all design rules, the placement is spaced or compressed as needed by calling a compactor. The basic requirements for a compactor are briefly described in Appendix D but the details are left to future research.

Once the placement phase is completed, the devices must be connected together with interconnect. This is done by using a maze router. A potential maze router is described in Appendix C.

After the maze routing is complete, a compaction is again in order as the maze router is not forbidden from creating design rule violations in the attempt to successfully complete routing. Any routing that the router was not able to complete is displayed as error regions to allow the user to determine the problem areas and repair them.

CHAPTER 7. CONCLUSIONS

7.1 Summary

A methodology for the layout of custom bipolar integrated circuits has been described. A subset of the capabilities of the methodology have been implemented in a prototype.

The methodology addresses many of the time consuming elements of custom layout as well as ensuring correctness of the resulting layout. However, the methodology does not sacrifice final IC cost in achieving these goals. Time is saved, correctness is ensured, and cost is minimized by:

- Coupling the schematic and layout domains to allow incremental connectivity checking.
- Automatic layout of all layout structures from devices to entire circuits.
- Full manual layout editing to allow the automatically created layout to be tailored to meet cost and performance criteria.
- Device verification to ensure that polygonal structures represent the intended devices.
- Incremental connectivity checking.
- Incremental design rule checking.
- Postlayout simulation and verification.
- Parasitic analysis.

- Extendibility and technology independence through the LISP language.

These capabilities combine to provide a rich environment that allows the designer full automation in initial layout creation, while maintaining complete layout flexibility.

Additional capabilities are provided by allowing the layout to track technology enhancements by automatically reworking and compacting the layout when design rules change. This increases the lifetime of a design by allowing the layout to stay current with the technology.

7.2 Suggestions for Further Work

As much of this thesis has been devoted to describing a methodology that has been only partly implemented, further work should naturally be devoted to implementing the full methodology.

Although ALE depends strongly on the placement, routing, and compaction phases as parts of automatic layout, these subjects have been treated superficially in order to limit the scope of this thesis. Great opportunities present themselves for algorithm work in all areas of automated custom layout. The areas of placement and routing are espe-

cially appealing as little work has been done in their specific application to custom integrated circuits.

The area of simulation verification is also interesting. The technique for simulation verification presented here where a circuit simulation is compared against a postlayout simulation is powerful but extensions are possible. One of the most interesting is to capture parts of the specification of the circuit in LISP expressions which could then be compiled into a simulation template and verification expressions. This would allow ALE to "understand" performance criteria such as distortions and slew rates, and be able to generate simulations that would simulate these criteria and automatically check the circuit performance after simulation. This procedure would occur both for schematic simulation and postlayout simulation.

REFERENCES

- [1] J. Soukup, "Circuit Layout", *Proceedings of the IEEE*, 1981, pp 1281-1304
- [2] S.L. Hurst, "Custom-Specific Integrated Circuits", *Marcel Dekker Inc.*, 1985, pp 258-284
- [3] C. Mead, and L. Conway, "Introduction to VLSI Systems", *Addison-Wesley*, 1980, pp 38-59
- [4] A.B. Glaser, and G.E. Subak-Sharpe, "Integrated Circuit Engineering", *Addison-Wesley*, 1977, pp 291-333
- [5] B.T. Preas, and P.G. Karger, "Automatic Placement: A Review of Current Techniques", *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, 1986, pp 622-629
- [6] S. Sahni, and A. Bhatt, "The Complexity of Design Automation Problems", *Proceedings of the 17th ACM/IEEE Design Automation Conference*, 1980, pp 402-411
- [7] W.E. Donath, "Complexity Theory and Design Automation", *Proceedings of the 17th ACM/IEEE Design Automation Conference*, 1980, pp 412-419
- [8] A.A. Szeplieniec, "Integrated Placement/Routing in Sliced Layouts", *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, 1986, pp 300-307
- [9] A.D. Lopez, and H.F.S. Law, "A Dense Gate Matrix Layout Method for VLSI", *IEEE Transactions Electron Devices*, Aug. 1980, pp 1671-1675
- [10] Y.C. Chang, S.C. Chang, and L.H. Hsu, "Automated Layout Generation Using Gate Matrix Approach", *Proceedings of the 24th ACM/IEEE Design Automation Conference*, 1987, pp 552-558
- [11] C. Sechen, and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package", *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, 1986, pp 432-439
- [12] M.A. Elmasry, "Digital Bipolar Integrated Circuits", *Wiley-Interscience*, 1983, pp 281-315
- [13] N. Weste, "MULGA - An Interactive Symbolic Layout System for the Design of Integrated Circuits", *The Bell System Technical Journal*, Jul-Aug 1981, pp 823-857

- [14] A.E. Dunlop, "SLIP: Symbolic Layout of Integrated Circuits with Compaction", *IPC Business Press*, Nov. 1978, pp 387-391
- [15] P.W. Kollaritsch, and N. Weste, "TOPOLOGIZER: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout", *IEEE Journal of Solid-State Circuits*, June 1985, pp 799-804
- [16] W. Wolf, J Newkirk, R. Mathews, and R. Dutton, "DUMBO, a Schematic to Layout Compiler", *Third Caltech Conference on VLSI*, 1983, pp 379-393
- [17] B.J. Cox, "Object Oriented Programming - An Evolutionary Approach", *Addison-Wesley*, 1986
- [18] E.R. Tello, "Object-Oriented Programming", *Dr. Dobb's Journal*, Mar. 1987, pp 126-134
- [19] D.A. Moon, "Object-Oriented Programming with Flavors", *OOPSLA '86 Proceedings*, 1986, pp 1-8
- [20] D.G. Bobrow, K. Khan, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops - Merging Lisp and Object-Oriented Programming", *OOPSLA '86 Proceedings*, 1986, pp 17-29
- [21] P.H. Winston, and B.K.P. Horn, "LISP - Second Edition", *Addison-Wesley*, 1984
- [22] Dept. of Computer Science - UBC, "LISP/MTS User's Guide", *Dept. of Computer Science - UBC*, 1974
- [23] G. Taylor, and J.K. Ousterhout, "Magic's Incremental Design-Rule Checker", *Proceedings of the 21st ACM/IEEE Design Automation Conference*, 1984, pp 160-165
- [24] J.B. Rosenberg, "Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries", *IEEE Transactions on CAD*, Jan. 1985, pp 53-67
- [25] J.K. Ousterhout, G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor, "MAGIC: A VLSI Layout System", *Proceedings of the 21st ACM/IEEE Design Automation Conference*, 1984, pp 152-159
- [26] Y.E. Cho, "A Subjective Review of Compaction", *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, 1985, pp 396-404

- [27] Y.Z. Liao, and C.K. Wong, "An Algorithm to Compact a Symbolic Layout with Mixed Constraints", *Proceedings of the 20th ACM/IEEE Design Automation Conference*, 1983, pp 107-112
- [28] C.Y. Lee, "An Algorithm for Path Connections and Its Applications", *IRE Transactions on Electronics*, Sept. 1961, pp 346-365
- [29] W.M. Dai, and E.S. Kuh, "Hierarchical Floor Planning for Building Block Layout", *Proceedings of the IEEE International Conference on Computer Aided Design*, 1986, pp 454-457
- [30] R.H.J.M. Otten, "Automatic Floorplan Design", *Proceedings of the 19th ACM/IEEE Design Automation Conference*, 1982, pp 261-267
- [31] T. Adachi, H. Kitazawa, M. Nagatani, and T. Sudo, "Hierarchical Top-Down Design Method for VLSI Chip", *Proceedings of the 19th ACM/IEEE Design Automation Conference*, 1982, pp 785-791
- [32] M.A. Breuer (Ed.), "Design Automation of Digital Systems, Vol. 1, Theory and Techniques", *Prentice-Hall*, 1972, pp 213-282
- [33] J.P. Blanks, "Initial Placement of Gate Arrays Using Least Square Methods", *Proceedings of the 21st ACM/IEEE Design Automation Conference*, 1984, pp 670-671
- [34] R. Forbes, "Heuristic Acceleration of Force-Directed Placement", *Proceedings of the 24th ACM/IEEE Design Automation Conference*, 1987, pp 735-740
- [35] J.P. Blanks, "Near Optimal Quadratic-Based Placement for a Class of IC Layout Problems", *IEEE Circuits and Devices Magazine*, Sept. 1985, pp 31-37
- [36] N.R. Quin, and M.A. Breuer, "A Force Directed Component Placement Procedure for Printed Circuit Boards", *IEEE Transactions on Circuits and Systems*, June 1979, pp 377-388
- [37] K.J. Antreich, F.M. Johannes, and F.H. Kirsch, "A New Approach for Solving the Placement Problem Using Force Models", *Proceedings IEEE International Symposium on Circuits and Systems*, 1982, pp 481-486
- [38] L. Sha, and R.W. Dutton, "An Analytical Algorithm for Placement of Arbitrarily Sized Rectangular Blocks",

Proceedings of the 22nd ACM/IEEE Design Automation Conference, 1985, pp 602-608

- [39] J.H. Hoel, "Some Variations on Lee's Algorithm", *IEEE Transactions on Computers*, Jan. 1976, pp 19-24
- [40] R.L. Joseph, "An Expert Systems Approach to Completing Partially Routed Printed Circuit Boards", *Proceedings of the 22nd ACM/IEEE Design Automation Conference, 1985, pp 523-528*
- [41] J.F. Lee, and D.T. Tang, "On VLSI Compaction with Mixed Grid and Edge Constraints", *Proceedings of the IEEE International Conference on Computer Aided Design, 1986, pp 308-311*
- [42] J. Weinstein, "A New Two-Dimensional Compaction Algorithm for Symbolic Layout", *IEEE Custom Integrated Circuit Conference, 1987, pp 605-609*

GLOSSARY

ALE: Automated Layout Editor.

Automated Layout Editor (ALE): A program implementing a custom layout methodology which is highly automated.

Automatic Layout: The computer process of creating the layout of a circuit without human intervention.

CAD: Computer Aided Design

Circuit Schematic: A graphical representation of the interconnections of devices in a circuit.

Computer Aided Design (CAD): The process of designing with the assistance of computer programs.

Compaction: The process of expanding or collapsing a layout so that the layout meets specified design rules and constraints.

Connectivity Checking: The process of checking that the connectivity of devices in a layout match that of the circuit schematic.

Constraints: Limits on various layout parameters such as capacitances, resistances, voltage differences, and locations of devices.

Design Rule Checking (DRC): The process of verifying that the manufacturing constraints on the layout of a circuit are met. This includes spacings, widths, enclosures, and overlaps of mask levels.

DRC: Design Rule Checking

Device Construction Verification: The process of verifying that the devices in a layout are constructed in a valid manner. This differs from design rule checking in that design rule checking only determines if a polygon configuration is manufacturable, not whether it performs a particular primitive device function.

Error doublet: A two element LISP list used in checking some error situations. The syntax is:

(boolexp errmsg)

If `boolexp` returns non-NIL when evaluated then an error has occurred. `Errmsg` is the error text.

Error triplet: A three element LISP list used in creating error shapes with the syntax:

```
(boolexp polylist errmsg)
```

If `boolexp` returns non-NIL when evaluated, the polygons from `polylist` are copied onto the error layer with the error message `errmsg` associated with them.

Floor planning: The process of establishing approximately where devices will be placed, and where routing will go. This is done before any layout has been performed.

IC: Integrated Circuit

Integrated Circuit (IC): A circuit which has been manufactured as a single entity. IC usually refers to circuits which have been manufactured on a single piece of silicon.

Layout: The polygonal database that represents a circuit schematic.

Layout Editing: The process of modifying the layout of a circuit in order to correct errors, or to optimize it.

Layout Verification: The process of verifying that the layout of a circuit is correct. This includes design rule checking, device construction verification, connectivity checking, constraint satisfaction, and post layout simulation verification.

Mask Level: A set of polygonal regions where a particular IC manufacturing process step will be performed.

Maze Routing: A type of algorithm that routes a layout by exploring it in a manner analogous to maze running.

Nonprimitive Devices: A device that contains hierarchy and may require placement beyond relative placement, as well as routing.

Placement: The process of positioning devices in a layout. These devices are positioned in a manner that minimizes a cost function that is related to final layout area.

Polygonal Database: The mask levels of an IC in a polygonal representation.

Post Layout Simulation: The process of simulating the extracted netlist of a layout to determine the layout's actual performance.

Post Layout Simulation Verification: The process of verifying that the post layout simulation using the simulation profile matches that simulated using the simulation profile on the circuit schematic.

Primitive Device: A device that contains only polygonal structures and can be constructed via a procedural definitions.

Quadratic Placement: A type of placement where the cost function that is being minimized has a quadratic nature.

Routing: The process of connecting devices in a layout with interconnect.

Schematic Capture: The process of entering a circuit schematic into a computer.

Simulator: A computer program used to emulate the behaviour of a circuit.

Simulator Interface: A portion of computer code whose purpose is to create a file that a simulator can read, to have the simulator run a simulation, and to process the results of the simulation.

Simulator Profile: A simulator setup which is used to simulate both the circuit schematic and the layout to allow the simulation results to be compared for equivalence.

APPENDIX A. COMPLETE TECHNOLOGY DEFINITION FOR NPN

The complete device definition for the NPN transistor is presented in this Appendix to allow a cohesive examination.

Example:

```
(device npn
  (properties
    (size 'Size of transistor in multiples of 1X"
      "Performance"
      'real
      1
      '(range 1 20))
    (emitype "Strip or separate emitter NPN"
      "Auto-Layout"
      'string
      'separate
      '(strip separate))
    (rowcount "Number of emitter rows"
      "Auto-Layout"
      'integer
      1
      '(range 1 5))
    (maxIbase "Maximum base current (mA)"
      "Verification"
      'real
      .1
      '(range .001 10))
    (maxIcoll "Maximum collector current (mA)"
      "Verification"
      'real
      3
      '(range .01 100))
    (maxIem "Maximum emitter current (mA)"
      "Verification"
      'real
      3
      '(range .01 100))
    (SPICEcreate "Function which creates the SPICE line"
      "Simulation"
      "function"
      'NPNSPICEcreate)
  )
  (propconstraints
    ((< size rowcount)
      "Transistor size not larger than the number of rows)
    ((& (> rowcount 1)
      (= emitype 'separate)
```

```

    (-(integerp size)))
    (concat "Size of transistor with multiple rows and "
            "separate emitters must be integer.))
  )
  (symbolshape (draw (0 0) (2 0))
               (draw (2 -3) (2 3))
               (draw (2 1) (4 3) (4 5))
               (draw (2 -1) (4 -3) (4 -5))
               (draw (4 -3) (3 -1) (2 -2) (4 -3)))
  )
  (symbolpins (b (0 0) 'io)
              (c (4 5) 'io)
              (e (4 -5) 'io))
  )
  (autolayout

    ; Define local variables
    (localvar emipointers emilist npnlayout basewidth)

    ; Create all emitter rows and store them on emipointers
    (do$ ((i 1 (+ i 1)))
          ((> i (propref 'rowcount)))
          (setq emipointers
                (append emipointers
                        (npnemi i
                              (propref 'size)
                              (propref 'emitype)
                              (propref 'rowcount))
                        )
          )
    )

    ; Step through all emitter rows and find the widest one.
    ; This allows the width of the base to be determined.
    (setq basewidth
          (+ (apply 'max (mapcar 'findwidth emipointers))
             (* 2 (encl 'p+ 'n+))))

    ; This loop steps through all created emitter rows. On
    ; the first emitter row the npnlayout is set to the
    ; emitter row itself. Each subsequent emitter row has a
    ; base contact placed between itself and the previous
    ; emitter. After stepping through all emitter rows,
    ; the layout is surrounded by a polygon of p+, and a
    ; polygon of n+.
    (do$ ((emilist emipointers (cdr emilist))
          (currenti (car emilist) (car emilist)))
          ((= emilist nil)
           (gensurround 'epi-
                        (gensurround 'p+

```

```

                                npnlayout
                                '((purpose npnbase))
                                (noderef 'b))

                                nil
                                (noderef 'c)))
(cond ((= curremi (car emipointers))
      (setq npnlayout curremi))
      (t (placebelow npnlayout
                     (npnbase basewidth)
                     '(on p+))))
)
)
(deviceverification
  ((notnode (layerref 'epi) (noderef 'c))
   (notnode (layerref 'epi) (noderef 'c))
   "All epi in a NPN must be connected to the collector")

  ((layerref 'dn+ '(- (purpose epicon)))
   (layerref 'dn+ '(- (purpose epicon))))
   "All dn+ in an NPN must have purpose epicon")

  ((notnode (layerref 'dn+) (noderef 'c))
   (notnode (layerref 'dn+) (noderef 'c))
   "All dn+ in an NPN must be connected to the collector")

  ((interact (layerref 'dn+) (layerref 'p+))
   (interact (layerref 'dn+) (layerref 'p+))
   "dn+ may not interact with p+ in an NPN")

  ((notsurround 'epi (layerref 'dn+))
   (notsurround 'epi (layerref 'dn+))
   "All dn+ in an NPN must be surrounded by epi")

  ((layerref 'n+
   '(- (| (purpose 'npnem) (purpose 'epicon))))
   (layerref 'n+
   '(- (| (purpose 'npnem) (purpose 'epicon))))
   "All n+ in an NPN must have purpose npnem or epicon")

  ((notnode (layerref 'n+) (noderef 'e))
   (notnode (layerref 'n+) (noderef 'e))
   "All n+ in an NPN must be connected to the emitter")

  ((notsurround 'epi (layerref 'n+))
   (notsurround 'epi (layerref 'n+))
   "All n+ in an NPN must be surrounded by epi")

  ((notsurround 'p+ (layerref 'n+ '(- (purpose 'npnem))))
   (notsurround 'p+ (layerref 'n+ '(- (purpose 'npnem))))

```

```

    "All npn emitters must be surrounded by p+")

((interact (layerref 'n+ (purpose 'epicon))
  (layerref 'p+))
 (interact (layerref 'n+ (purpose 'epicon))
  (layerref 'p+))
 "epi contacts may not interact with p+")

- ((layerref 'p+ '(- (purpose 'nnpbase)))
  (layerref 'p+ '(- (purpose 'nnpbase))))
  "All p+ in an NPN must have purpose npnbase")

((notnode (layerref 'p+) (noderef 'b))
 (notnode (layerref 'p+) (noderef 'b))
 "All p+ in an NPN must be connected to the base")

((notsurround 'epi (layerref 'p+))
 (notsurround 'epi (layerref 'p+))
 "All p+ in an NPN must be surrounded by epi")

((layerref 'contact '(- (| (purpose 'nnpbasecon)
  (purpose 'nnpnemcon)
  (purpose 'npncollcon))))
 (layerref 'contact '(- (| (purpose 'nnpbasecon)
  (purpose 'nnpnemcon)
  (purpose 'npncollcon))))
 (concat "All contacts in an NPN must have purpose "
  "nnpbasecon, npnemcon, or npncollcon"))

((notsurround 'metal (layerref 'contact))
 (notsurround 'metal (layerref 'contact))
 "All contacts must be surrounded by metal")

((notsurround (| (layerref 'p+) (layerref 'n+)) 'contact)
 (notsurround (| (layerref 'p+) (layerref 'n+)) 'contact))
 "All contacts must be surrounded by n+")
)
(parameterextract
 (setprop 'size (/ (area (layerref 'n+
  (purpose 'nnpnem)))
  npnlxsize))
 (setprop 'maxIbase
  (* (area (layerref 'contact
  (purpose 'nnpbasecon)))
  maxIbaseperu2))
 (setprop 'maxIcoll
  (* (area (layerref 'contact
  (purpose 'npncollcon)))
  maxIcollperu2))
 (setprop 'maxIem
  (* (area (layerref 'contact

```

```

                                '(purpose 'npnemcon)))
                                maxIbaseperu2))
                                )
                                (parametercompare
                                ((compare size tolerance .05)
                                (layerref 'n+ '(purpose 'npnem))
                                "Size of NPN not within tolerance"))
                                ((compare maxIbase extractedgreater)
                                (layerref 'contact '(purpose 'npnbasecon))
                                "Base not able to support required current"))
                                ((compare maxIcoll extractedgreater)
                                (layerref 'contact '(purpose 'npncolicon))
                                "Collector not able to support required current"))
                                ((compare maxIem extractedgreater)
                                (layerref 'contact '(purpose 'npnemcon))
                                "Emitter not able to support required current"))
                                )
                                )

```

APPENDIX B. QUADRATIC PLACEMENT ALGORITHM

ALE requires a placement algorithm that can handle arbitrarily sized rectilinear blocks as well as constraints on positioning. These constraints could include requiring two devices to be placed very close to each other in the same orientation for best matching, or requiring certain other devices to be placed as equidistant as possible from another block for thermal gradient reasons.

These requirements eliminate many exchange algorithms which make the assumption that all devices are square and equal in size [32]. The basic principle in these exchange algorithms is to define an initial placement on a square grid and then exchange the location of devices until no further improvement can be made. These algorithms cannot effectively be modified to handle arbitrarily sized devices with arbitrary orientation.

Another class of algorithms are the constructive algorithms [32]. These algorithms start from a seed placement that can be defined automatically or manually by the user. Devices are added to the placement one at a time with the goal of clustering highly connected devices closely together. These algorithms perform well as the initial placement for an exchange algorithm, but without the

exchange step the algorithm performance is rarely acceptable.

B.1 The ALE Placement Algorithm

One of the remaining classes of placement algorithms is numerical solution [33,34]. The placement problem can be formulated as a set of simultaneous equations which can be solved to result in a placement. This type of algorithm is very CPU intensive but can offer quality results for an otherwise intractable problem.

In these formulations a wire length metric is usually the primary minimization consideration. The wirelength metric is one of the predominant factors that human layout designers use when constructing a custom layout because in custom layouts there is usually a direct correlation between wire length and final layout area.

Computing the exact wire length is usually not practical when producing a placement and so various approximations are made. One possibility is to assume that, if given a set of devices that should be connected together, each device is directly connected by a wire to every other device. The wire length can then be computed on a centre of device to centre of device basis using either Euclidean or rectilinear distance. Frequently the square of Euclidean distance is

used to avoid discontinuity problems in solution and to reduce computational complexity.

The assumption that in a set of connected devices, each device connects directly to every other device is in general poor, but many heuristics have been developed based on the total connectivity of the node to make the result reasonably accurate.

The degenerate point solution where all coordinates are placed at the same X-Y location, (which certainly minimizes the wire length metric) is avoided by imposing solution constraints which tend to spread the solution out. One possible constraint requires that the sum of the square of the X coordinates of the blocks equals a constant (likewise with the Y coordinates). The correlation between X and Y coordinates must also equal zero to avoid a solution along a line [35]. Other possible constraints based on spring analogies to repulse blocks that are not connected have also been used [36,37].

The justifications for some of the constraints that have been used in the past to avoid the degenerate solution do not have a physical analogy and usually avoid the problem of overlapping blocks by repositioning overlapping blocks in a post processing step. However, because the initial solution ignored the overlap problem, the perturbation

required to achieve a solution with no overlap is substantial and does not produce very good results. In general, constraints which do not have a good physical justification should be avoided.

A recent paper by Sha and Dutton [38] describes a set of constraints which models devices as rectangles with rounded corners. The rectangles are modelled as rounded to keep all equations continuous and are referred to as modified rectangles. The formulation of simultaneous equations in this paper avoids overlaps of the modified rectangles and does not use any physically questionable constraints. This implementation also optimizes the orientation of devices which is a great advantage over other formulations.

Some postprocessing is still required to eliminate corner to corner overlaps but this is greatly reduced from the other formulations and can be handled by a standard compactor.

This placement algorithm has been implemented in the prototype of ALE with promising results. The best results were achieved by initially solving the placement using the constraints of Quinn [36] and then using the nonoverlap constraints on the modified rectangles to complete the solution. An example of a placement produced with this technique is shown in Figure 19.

Enhancements to the algorithm that have not yet been implemented are:

- Using the position of terminals on the devices to better optimize orientation
- Permitting rectilinear blocks instead of only rectangular blocks
- Constraining the absolute position of devices
- Constraining the relative position of devices
- Constraining devices to be equidistant from another device
- Constraining the aspect ratio of the resulting placement.

NETLIST-AS F-2083.827 OVERFACT-250

Figure 19. A Placement Produced with Modified Rectangle Constraints

APPENDIX C. MAZE ROUTING ALGORITHM

A class of routing algorithm that is currently very popular is the channel router. With channel routers, the entire routing area is broken up into rectilinear regions that are called channels. The first step is a global routing step to determine which channels will contain specific interconnects. The channels are then routed individually so that interconnect is continuous across channel boundaries. As the channel size gets smaller, and the number of channels gets very large, the global routing gets very difficult, and ensuring that interconnect is continuous across channel boundaries also becomes very difficult.

Custom layout involves devices of arbitrary size and aspect ratio that are placed in a fairly arbitrary manner. If the channel routing model is used, this leads to many small channels which is a situation where channel routing performs poorly. A more appropriate routing algorithm is a maze router.¹⁰ This type of router is typically used in Printed Circuit Board (PCB) layout or gate array layout.

¹⁰ In custom layout, the case of many small channels applies most strongly when creating low level functional blocks. When routing the top level of the IC the channel routing philosophy often becomes useful, and a channel router could be used effectively.

In order to operate effectively in ALE, one requirement imposed on the routing algorithm is that it not restrict itself to routing through areas that can be interconnected without design rule violation. If routing introduces rule violations, the compactor can automatically repair them. Routing in a manner that does not create violations is far preferable to routing that does create violations, and this should be considered by the cost function that the router employs. As an example of this type of routing, a NPN could have an interconnect threaded through its base. The NPN could then be recompact to remove any design rule violations. This example is shown in Figure 20.

In ALE there are no predefined terminals or pins on the devices. Therefore the router must be able to determine the areas in which routing may be performed. As well, the router may be required to route in diffusions instead of metal or poly which puts some difficult restrictions on the areas in which the routing may be performed.

Within these restrictions, the most suitable type of routing algorithm is a maze router [39]. The maze router has not been implemented, but the following section captures some of the important ideas.

Figure 20. Routing a track through a NPN transistor

C.1 The ALE Routing Algorithm

The first step is to take the portion of the layout hierarchy to be routed and construct a line search data construct. This data construct is a connected graph that avoids all the routing obstacles. An example of this search structure is shown in Figure 21 for a routing layer (like metal) which can be routed over any other layer. This structure shows all the possible paths that a piece of metal can take. A search structure is created for all routing layers.

An important feature is that arcs are put into the graph even if a track cannot be routed in that location, without causing spacing violations. The compactor would be expected to repair these violations after the routing is finished.

Initially, the routing algorithm should take the signals one at a time, define goal segments in the search data structure which represent unconnected subnodes and run the maze of the search data structure until all unconnected subnodes are found.

The maze running is heuristically governed to avoid excessive backtracking. This is done by visualizing the maze running as creating a rectilinear spanning tree that connects the unconnected subnodes. Thus each part of the maze running is goal governed, [40] i.e. connect this

subnode to that track. Because of these goals it is possible to initially prune routes that are likely to be poor candidates for a solution. These pruned paths are stored so that eventually, if no good route is found, these pruned paths can be investigated.

If a signal is successfully routed, there is some attempt to find more solutions. Again this is heuristically governed to limit the amount of effort. The precise heuristics have not yet been determined, but some obvious ones are total number of explored paths, total number of dead ends, or amount of CPU time.

If more than one solution is found, then they are scored to determine which is the best. The scoring function is based on the length of the route, the number of vias, and resistance/capacitance of the routing. Another scoring criteria quantifies the amount that the rest of the layout has to be disturbed to implement the route without violations, i.e. Figure 20. As well, the likelihood that the routing will block signals that still need to be routed could be quantified into the scoring function.

Having found one good solution (hopefully) the search data structure is updated to reflect the routing and the router can begin work on the next node.

If no solution is found then the portions of the node that could not be routed are left alone and intervention from the layout designer will be required.

The routing algorithm will be usable in a batch operation or one in which the layout designer will interactively route one node at a time. In this mode ALE will be able to display all the possible routes, the scoring assigned by the heuristics, and allow the layout designer to select one of these routes.

APPENDIX D. COMPACTION ALGORITHM

Compaction in the prototype ALE has been implemented to a limited extent. Currently, the data structure maintains all the visible neighbours of a data element and so is able to move data elements one at a time to obey design rules. This is currently used effectively in the interactive editing part of the prototype ALE.

The compaction algorithm must be able to work on an entire circuit or a localized portion as specified by the layout designer. It should take maximal guidance from the initial layout and disturb its relative placement as little as possible. This is not particularly important when the placement has been produced by the ALE placement algorithm, but critically important when the placement has been produced by the layout designer because typically it is a safe assumption that if a layout designer has put something somewhere, it was put there for a good reason. It should also be possible for the layout designer to introduce constraints to direct the compaction [41].

D.1 The ALE Compaction Algorithm

Most compaction algorithms currently in existence build a constraint graph in one of the Cartesian directions, solve it for the longest path, and then do the same for the other

Cartesian direction [26]. This can be repeated until there is no further improvement.

This is the basic algorithm that should initially be implemented in ALE although the ideas from Weinstein [42] which allow some 2-D elements to be introduced into the basic 1-D compaction steps will be added. In Weinstein, a constraint graph is built in both Cartesian directions before compaction is performed. This prevents the order of compaction from having any effect on the final result. As well, this step tends to increase the locality of the algorithm so that it pays more attention to the initial placement than other compactors.

Most compaction algorithms are constraint based, and adding extra user constraints is certainly possible. Some user constraints can conceivably add cycles to the constraint graph, but there are techniques in many compaction algorithms to deal with this problem.

END

09.08.88

FIN