Corner-based Geometric Layout Rule Checking for VLSI Circuits

By

Michael Helmut Arnold

B.S. (Michigan State University) 1978

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: [illegible] 10/3/85
Chairman Date

[illegible]

[illegible]

Corner-Based Geometric Layout Rule Checking
for VLSI Circuits

CORNER-BASED GEOMETRIC LAYOUT RULE CHECKING FOR VLSI CIRCUITS

Michael Helmut Arnold

Abstract

Layout rule checking is traditionally done through sequences of region-operations, and a few experimental systems use pixel-based processing. This dissertation examines these approaches in detail, and then proposes corner-based checking as an efficient and flexible alternative. In corner-based checking contextual rules, specifying conditions at corners matching patterns, are applied to the design. A rule compiler is used to convert the user-readable rule description to an efficient, indexed, internal form prior to checking. Hierarchical and incremental check algorithms that eliminate redundant checking are also developed. These algorithms greatly enhance the effectiveness of layout rule checking. Measurements from several systems implementing corner-based checking and the hierarchical and incremental algorithms demonstrate their viability and effectiveness.

Corner-based checking has several advantages. First, it checks all rules in a single pass over the data. This avoids the I/O bottleneck that is common in the multi-pass region-operation systems. The rule-based nature of corner-based checking provides inherent flexibility: variants of design rules that would require the coding of new operations in region-operation systems can often be accommodated by modifying the rule specification. Corner-based rules also permit directional context, which is notoriously difficult to establish in region-operation systems. Finally corner-based systems associate violations with points in a design rather than edges or regions. The consequent simplicity of piecewise processing facilitates hierarchical and incremental checking.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3803, monitored by the Naval Electronic System Command under Contract No. N00039-78-G-0013-0004.

Acknowledgments

I would like to thank Ann Lanfri, Chong Lee, and Bill Weir of MCV and Metheus Corporations for their continuing cooperation and support in my research. I am also in debt to Greg Cardell, Myron White and especially Keith Billings for their ideas and work on Leo and Leo45, the Metheus corner-based systems. I would also like to thank Jack Vernetti (for his help with the plots) and all the others who have helped make my association with Metheus productive and enjoyable.

I would like to thank Howard Landman and Stephen Pope for their help and comments with the development of the Lyra rulesets, and the many others at Berkeley and in industry who have enriched my research by their attention.

This research was funded, in part, by Defense Advanced Research Projects Agency.

Last, but not least, I am grateful to my research advisor, John Ousterhout. The corner-based idea was born out of a sequence of discussions with John in which I volunteered half-baked ideas and John shot them down (John is a crack shot). In addition to eventually leading to this dissertation, these meetings taught me the value of simplicity and precision in computer science. This is the single most valuable lesson I learned while at Berkeley, and I know it will continue to bear fruit in the years to come.

Table of Contents

CHAPTER 1

Introduction

This thesis presents my work in automatic design rule checking for integrated circuits (DRC). It introduces corner-based design rule checking as an efficient and flexible alternative to traditional region-based checking, that is well suited for hierarchical and incremetal checking. A general formalism for corner-based rules is developed that can handle all-angle data and complex conditional rules, and its implementation is considered. A rule compiler is introduced to preprocess the input rule description for efficient processing. Hierarchical and incremental check algorithms are also developed. The identification of violations with points rather than edges or regions, makes corner-based checking particularly well suited for hierarchical and incremental systems. Several corner-based systems, are discussed. In addition, background material on the nature of the design rule checking problem, and other work in the area, is provided.

This chapter gives an overview. It briefly describes design rules and design rule checking, discusses the nature and scope of my research, outlines the most important ideas arising from and/or validated by the research, and outlines the rest of the thesis.

1.1. Design Rules and Design Rule Checking

Integrated circuits are specified in terms of geometric mask patterns, or artwork, for each of the layers in the circuit; see Figure 1.1. Design rules specify tolerances on these patterns. Tolerances typically govern width and spacing on conducting lines and various extensions and enclosures on circuit constructs such as transistors and contacts; see Figure 1.2. Design rules stem from the limitations of the circuit fabrication process. They are an abstraction of these limitations that acts as the interface between circuit design and process engineering. They free the circuit designer from the intricate details of the fabrication process and its limitations: he need only make sure he obeys the design rules.

It is important that design rules are checked automatically. Current designs typically contain one million or more separate geometric figures, making for very tedious and error-prone manual checking. Manual checking is certain to result in missed design rule violations. Such violations necessitate expensive and time consuming additional fabrication cycles. In addition it is very difficult to locate design rule violations by probing a finished circuit. Some violations are likely to go completely undetected, and result in degraded performance and a lower yield of working parts throughout the lifetime of the product. The only acceptable solution is complete automatic design rule checking of the mask artwork prior to fabrication.

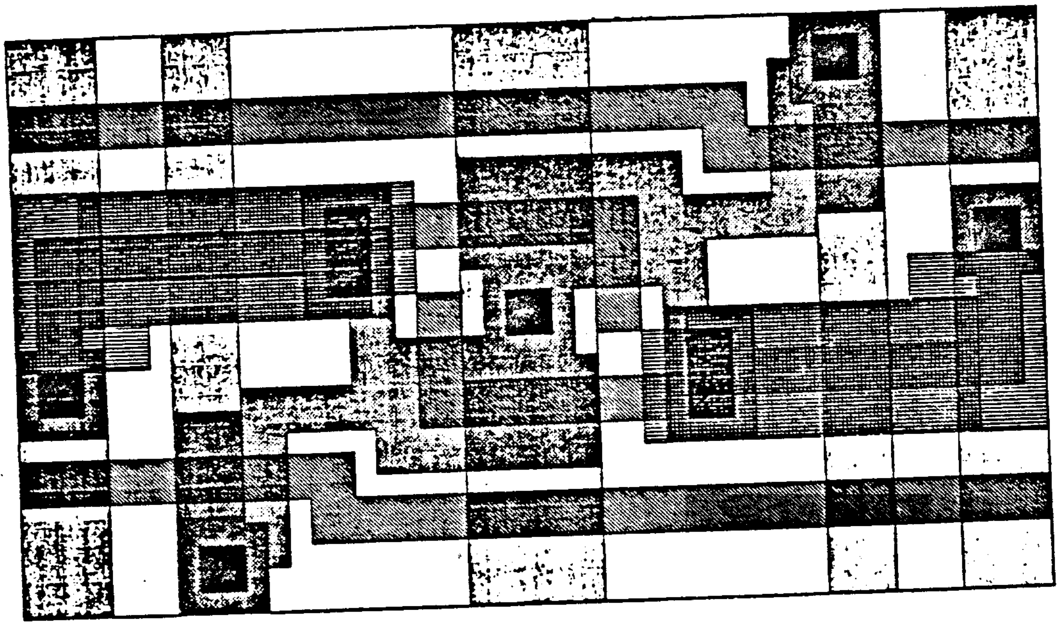


Figure 1.1. - Mask Artwork. This is the mask artwork for the basic register cell used in the RISC-II microprocessor chip developed at Berkeley. The RISC chip contains over 4000 such cells, comprising approximately one quarter of the total chip area. Each type of shading corresponds to a distinct mask layer in the design.

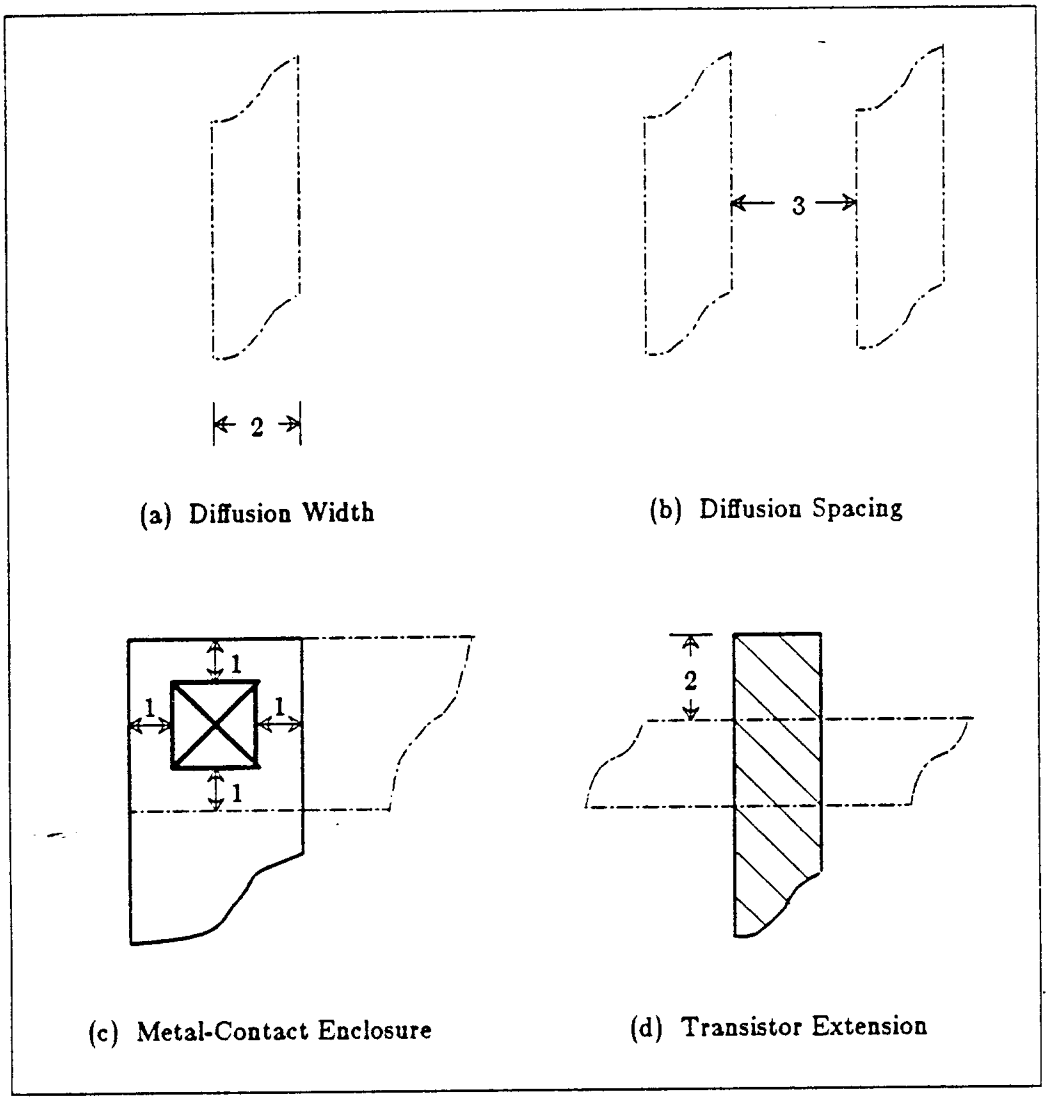


Figure 1.2. - Typical Design Rules. These examples are taken from the Mead-Conway rules for nMOS. Parts (a) and (b) give minimum widths and spacings (respectively) for lines on a particular layer. Parts (c) and (d) specify dimensional constraints on the formation of contacts and transistors (respectively). A design rule set contains anywhere from two dozen to over two hundred such rules. Some rules are more complicated; examples will be given in the next chapter.

To be useful, a design rule checker must meet several requirements. First it must be accurate. Since a single design rule violation can render an entire design nonfunctional, a design rule checker must miss no violations. In addition, a design rule checker that hides genuine violations in a sea of false alarms is almost as bad. This can happen if the checker inaccurately handles just one commonly occurring mask configuration. Thus a design rule checker must be accurate: it must miss no genuine violations, and generate few false alarms.

A useful design rule checker must also be flexible enough to check a variety of design rules. There are many integrated circuit technologies in use, and for each technology a number of fabrication lines. Each fabrication line has its own characteristic limitations, and hence its own design rules. In addition, as fabrication lines are refined and new ones introduced, design rules change over time. The natures of design rules are also diverse. Conditional design rules exist, that depend on factors as varied as the geometric context of a mask feature, the length of conducting lines, the electrical characteristics of a particular node, and even the intended function of the signal carried by a line. To be useful over time, and in more than a narrow context, a design rule checker must be flexible enough to check a variety of rulesets involving a variety of types of rules.

A design rule checker must also perform reasonably. As design complexity continues to grow, design rule checkers are faced with large and ever increasing amounts of data to process. A design rule check on a large design typically takes many hours or even days to complete. Such checking is expensive in terms of computer resources. In addition it makes design rule checking a batch process that is typically deferred until the end of the design cycle. Design rule violations detected so late can be quite difficult to fix: a significant amount of mask artwork in the vicinity of a violation may need to be altered to make room for the fix. Expensive design rule checking also inhibits design refinements, since any change in the design will require another costly design rule check. With the size and complexity of designs continually increasing, it is clearly important to look for efficient ways to do design rule checking.

In fact, the basic technique used almost universally for design rule checking was born more than a decade ago when designs were much smaller and less complex than today. This technique is poorly suited for processing designs of today's scale. In this approach, rule checking is implemented through sequences of *region-operations* on one or two layers at a time. These operations typically yield a new, intermediate, layer as output. Layers frequently consist of over 100,000 figures, and typical rulesets require hundreds of these operations. This results in the generation of great amounts of intermediate data during a design rule check, and a great amount of I/O. The number of data items is very large, and the amount of computation per item relatively small. Hence processing is I/O bound, and slow.

Several ideas for speeding up design rule checking have been proposed. One approach is to use special-purpose hardware. A number of hardware-assisted systems have been suggested. These systems employ pixel-based representations for the mask data; see Figure 1.3. An array of square pixels is laid over the design, and each pixel is marked with the mask layers occurring in it. In order to have sufficient resolution the pixel-array must be fine. A fine array over an entire VLSI design involves a very large amount of data. Thus, like the traditional region-operation approach, the hardware-assisted approaches involve a large amount of data, with relatively little processing per data item, and hence tend to be constrained by I/O bandwidth. Such special-purpose processing engines are also likely to be complex and expensive. No fully functional hardware-assisted design rule checker has yet been completed.

The elimination of redundant checking can be very effective in speeding up design rule checking. One such strategy, hierarchical processing, involves checking only one instance of repeated structures in a design. Since designs typically contain much repetition, such a strategy can reduce computation very significantly. For example, hierarchical checking of the RiscI microprocessor chip with Leo45 speeds up checking by almost a factor of 6. Another strategy, incremental checking, is to check only those portions of a design that have been modified since the last check. Again, this can greatly reduce the computation required for checking - particularly at the end of the design cycle, when minor modifications are typically

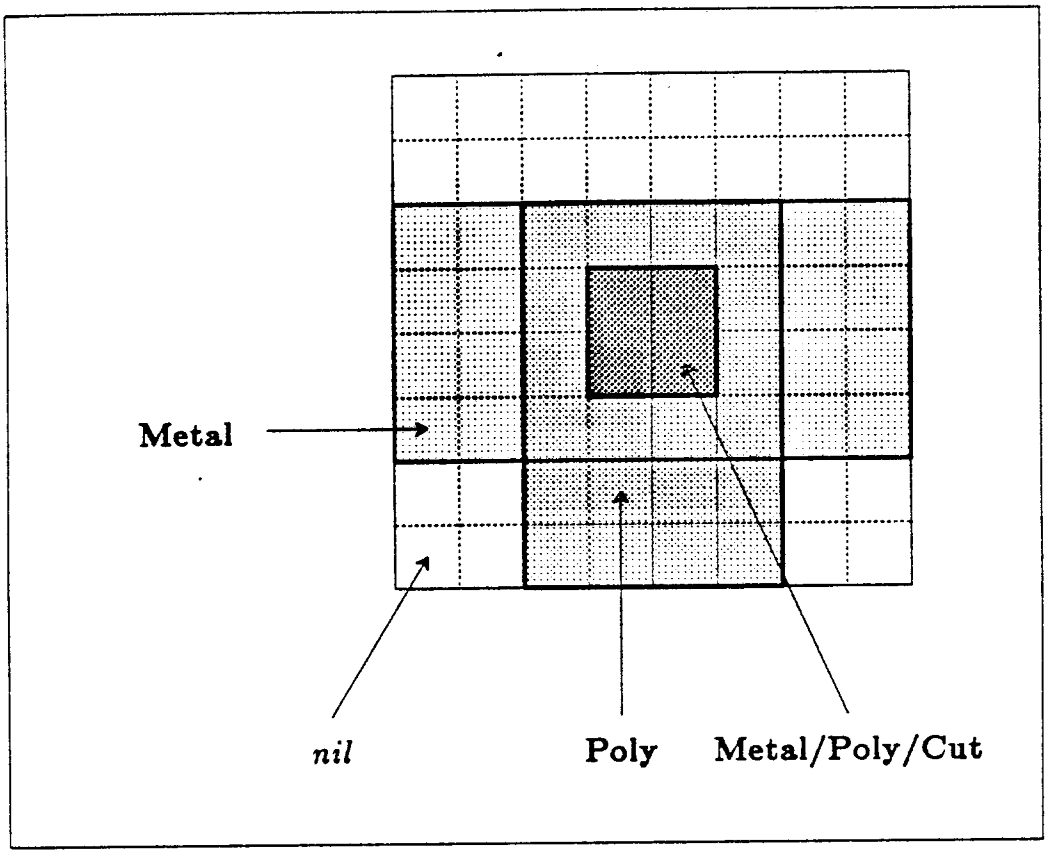


Figure 1.3. - Pixel-based Representation of Mask Artwork. In pixel-based systems, a square array of pixels is laid over the design, and each pixel is marked with the layers present within it. The pixel-array must be fine enough so that approximations at pixels that are only partially covered by mask layers don't result in false design rule violation reports or missed violations.

made tō fix bugs detected by design rule checking or simulation. An incremental check of RiscI, (again by Leo45) after a minor modification to the design takes less than 1% of the time for a full check. Lyra and Leo, corner-based systems developed in conjunction with this research, pioneered hierarchical and incremental checking (respectively).

1.2. Scope of My Research

My research has centered around an alternative approach to design rule checking: a *corner-based* approach where checking is done in terms of the corners in a design and their immediate environment. Corner-based checking is a fundamentally different approach to design rule checking that addresses the I/O bottleneck problem traditionally plaguing design rule checking. The input rule description is preprocessed with a *rule compiler* to generate an efficient, indexed, internal rule form and all rules are processed in parallel in one pass through the design artwork. Corner-based checking is both accurate and flexible. It lends itself to checking rules involving directional context, which are notoriously difficult to check in traditional region-operation systems.

My research also included the development of *Hierarchical* and *incremental* algorithms that reduce redundant checking and hence make design rule checks more efficient and interactive. Although these algorithms could be implemented on top of a region operation system, the corner-based approach, which associates violations with points rather than edges or areas, is more convenient.

Corner-based checking was first implemented in the Lyra system, in 1981. Lyra was written to test the the basic soundness of the corner-based approach and fill the need for an accurate design rule checker at Berkeley. It demonstrated that corner-based checking can be both accurate and flexible. It has been used on a number of large design projects, including the RISC microprocessors at Berkeley, spanning a number of nMOS and CMOS rulesets. On its first real use (for the RISC-I chip), Lyra found violations that had been missed by a previous DRC as well as by manual checking. Even though written in Lisp, Lyra was about 3 times as fast as the region-based NCA system, which was the industry standard at the time.

Lyra was the first hierarchical design rule checker. It demonstrated the feasibility and effectiveness of hierarchical checking. With interfaces to the Caesar and Kic geometric editors, Lyra also pioneered interactive design rule checking. The interface allowed designers to invoke Lyra on parts of the design currently being edited for "immediate" feedback on design rule violations. This feature proved quite useful to designers. Lyra is part of the Berkeley CAD tool distribution, and has been used at several hundred unviersity and industrial sites.

The viability of corner-based checking was further tested with Leo, a second, commercial, corner-based system. Leo was developed in conjunction with Metheus Corporation for use in their VLSI design workstation. Written in C with an eye toward efficiency, Leo is 3 times as fast as Lyra. In addition Leo is incremental, (it rechecks only the parts of a design that have been modified). Leo works quite well; its incremental and interactive checking capabilities have been selling points of the Metheus system.

The Lyra and Leo systems are useful only within a restricted context. They handle only manhattan designs, where all feature edges are lined up horizontally and vertically. A later version of Leo, Leo45, allows edges at 45 degree angles as well, but still excludes all other angles. The rulesets handled by these systems are composed of relatively simple, mainly unconditional rules. These restrictions are in the spirit of the simplified design philosophy popularized by Carver Mead and Lynn Conway. They are suitable for the Mead-Conway design style widely used in universities and a growing segment of the industrial community. They simplify the implementation of the systems, and improve performance.

Nevertheless, the question of the usefulness of corner-based checking in more general contexts must be addressed. To explore this issue, I have developed a general corner-based formalism and considered its implementation in detail. This formalism has provisions for processing all-angle mask data, specifying complex conditions on the interrelationship of features at corners (capable of capturing complex conditional rules), and provisions for the incorporation of nongeometric data, (again for checking conditional rules). This work shows that a general corner-based system is feasible, but that preprocessing would be required to generate any nongeometric contextual information used in the rules. The rule formalism is quite flexible. The proposed implementation uses standard compiler techniques as well as some fairly elaborate logical manipulations to transform the input rule specification into a simple regular internal form that can be efficiently checked. Though the performance of such a complex system can not be accurately predicted without implementation, it is encouraging that much of the complexity can be shifted to the rule compiler that transforms the input rule specification to an efficient internal form. The rule compiler is only run infrequently, so its efficiency is not a large concern. The actual rule checking is still relatively simple, though more complex and slower than in the more restricted systems that have been implemented. Many conditional rules rely on preprocessing to generate additional context information. Such preprocessing would presumably be done using traditional region-based operations. This suggests a hybrid system, involving region-based preprocessing to generate needed context information followed by the more efficient and flexible corner-based tolerance check.

1.3. Ideas In My Research

A number of key ideas on design rule checking have emerged out of my research. These ideas, more than particular systems or even particular algorithms, are my contribution to the field. Some of the ideas, such as pattern-directed, rule-based, processing and point/edge tolerance checks, were new. Others such as hierarchical, and incremental checking had previously been proposed, but were elaborated on (and validated) by my work. Still others involving more general corner-based systems, were just suggested by the research, and remain to be developed. The most important ideas are introduced below.

Corner-based checking uses *pattern-directed, rule-based* processing, a technique borrowed from Artificial Intelligence. The pattern-directed processing of corners works as follows. Each corner in a design is analyzed for certain mask patterns. The presence of certain predefined patterns triggers relevant rules. The rules in turn specify conditions to check at the corner. Care is given to index the triggering patterns so that the relevant rules at each corner can be identified quickly. Processing of a design involves one pass through the data (all rules, and layers are processed in parallel), and no intermediate layers are generated (the layers are checked "in-place"). This approach is radically different from the traditional region-operation approach, where rules are checked through sequences of operations, and each operation involves a separate traversal of the mask data. The *pattern-directed* processing concentrates the checking per data item scanned, and thus avoids the I/O bottle neck problem. Pattern-directed processing, looked at from another perspective, is rule-based. The rule-based nature of corner-based checking provides the characteristic flexibility of rule-based systems. Variants of design rules, that would require the coding of a new operation in region-based systems, can be accommodated in a corner-based system by the much simpler process of modifying a rule.

Another innovation of the corner-based approach, is the use of point/edge based tolerance checking, that is tolerance-checking involving the measuring of distances between corner-points and edges, rather than between pairs of edges; see Figure 1.4. This technique localizes checking to points in the design rather than edges or regions. Violations are associated with corner-points, and the independent checking of a piece of a design can be precisely defined as checking all the corner-points of that piece. The consequent simplicity of piecewise checking facilitates the implementation of hierarchical and incremental strategies.

Though hierarchical processing has been widely heralded as the solution to the excessive times required for design rule checking, hierarchical systems have been slow in coming. Lyra appears to have been the first fully-functional hierarchical design rule checker. Lyra demonstrated that hierarchical checking of structured designs can be effective, and that special restrictions on cell overlap are not required: as long as cell overlap remains relatively small, hierarchical checking in Lyra is effective. In addition Lyra's special handling of arrays proved very effective. Most of the regularity in VLSI designs is in the form of arrays. Special handling of arrays alone will give most of the advantages of hierarchical checking. There was also a negative result: checking of poorly-structured designs was as much as several times slower hierarchically than flat! This was because mask features involved in cell interactions ended up being checked at several distinct levels in the hierarchy.

Once hierarchical checking was in place, incremental checking proved easy to implement and very useful. User response to incremental checking in Leo was extremely favorable. Leo users frequently run design rule checks each day or so. Violations no longer go unnoticed

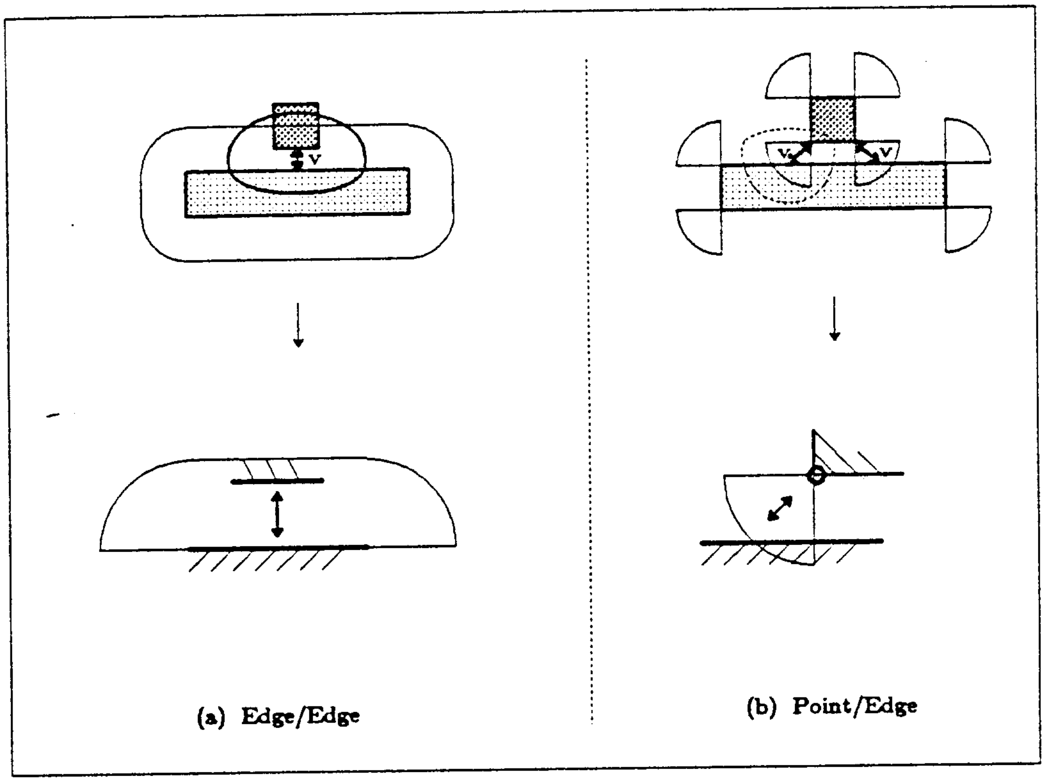


Figure 1.4. - Alternative Tolerance Check Methods. Traditionally, tolerance checks on mask regions have been done by checking the distance between region *edges* (a). In corner-based checking, tolerances are measured from corner *points* to region boundaries (b). Such point/edge checking has the advantage of very naturally splitting up into piecewise checks: checking a piece of a design, corresponds to checking tolerances from corner-points within that piece.

until after the design is complete and they are hard to fix.

Several new design rule check systems have incorporated some of the above ideas. The Mart design rule checker, developed by Bruce Nelson and Mark Shand at CSIRO is based directly on corner-based checking as in Lyra. The new internal Intel design rule checker uses point/edge tolerance checking to facilitate piecewise processing. The Magic design rule checker, recently developed by George Taylor and John Ousterhout at Berkeley, is strongly influenced by Lyra. Though edge-based rather than corner-based, the Magic system uses pattern-directed rule-based processing. Its hierarchical algorithm is similar to Lyra's, and its handling of arrays is identical. All these systems are discussed in Chapter 7, and references are given at the end of that chapter.

1.4. Outline

About half of the material in the following chapters provides background. It discusses the origin and nature of design rules, and presents the various approaches to design rule checking. This material provides an introduction to the design rule checking problem, and the context for the discussion of the corner-based approach, and the hierarchical and incremental algorithms of Lyra and Leo.

The remaining chapters are organized as follows. Chapter 2 considers where design rules come from, what they look like, and their role in the various design methodologies. It develops the *topo-tolerance* model for design rules that is used through out the thesis. Chapter 3 presents the traditional region-operation based method of design rule checking. Chapter 4 surveys existing design rule checkers, giving examples of traditional systems as well as other approaches. Chapter 5 introduces corner-based checking in fully general form. It develops a corner-based formalism and explores the scope of rules that can be handled by it with a number of examples. Chapter 6 discusses how a general corner-based system might be efficiently implemented. Chapter 7 surveys actual corner-based systems, focusing on the restrictions of each: how they simplify the implementation and how they limit rule checking capabilities. Chapter 8 discusses hierarchical and incremental checking. It presents both the approaches I used in Lyra and Leo, and the approaches used in other systems. Numerical measurements are presented in Chapter 9. Chapter 10 is the the conclusion.

CHAPTER 2

The Nature of Design Rules

2.1. Introduction

Design rules specify constraints on the minimum size of circuit components, and the maximum component density for integrated circuits. These constraints are given as minimum tolerances on various spacings, widths, enclosures and extensions in the mask artwork for the circuit. See Figure 2.1 for examples.

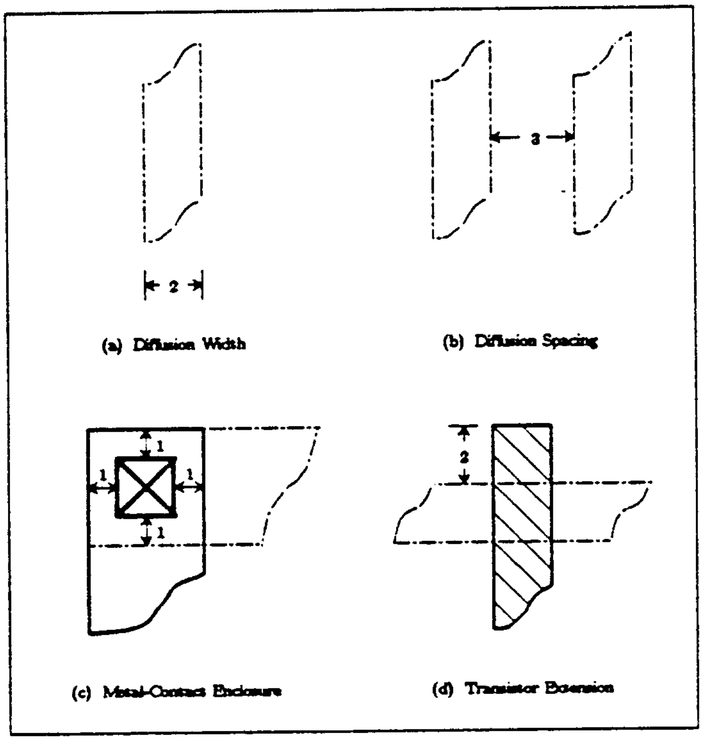


Figure 2.1. - Typical Design Rules. These examples are taken from the Mead-Conway rules for nMOS. Parts (a) and (b) give minimum widths and spacings (respectively) for lines on a particular layer. Parts (c) and (d) specify dimensional constraints on the formation of contacts and transistors respectively. A design rule set contains anywhere from a couple dozen to over two hundred such rules. Some rules are more complicated; examples will be given later in this chapter.

There are a number of reasons for making devices as small as possible and component density as high as possible. Most importantly, the probability of any given chip being fabricated correctly (the yield) decreases dramatically as overall circuit size grows. This is because fatal defects, caused by impurities in the silicon crystal or dust contamination during processing, occur with approximately fixed and independent probability in each unit-area of the circuit. Thus the probability of at least one (fatal) defect occurring in a circuit grows exponentially with the area of the circuit. Hence smaller circuits have much better yield and are more economical to produce. In addition, there is a practical limit on the maximum circuit size that can be fabricated: beyond some size, yield will be so astronomically small that no fabricated chip is likely to work. Thus smaller devices permit more complex chips. Even disregarding yield, smaller circuits are more economical simply because there are more chips per wafer fabricated; see Figure 2.2. Still another reason for minimizing component sizes is that circuits composed of smaller components are faster and consume less power.

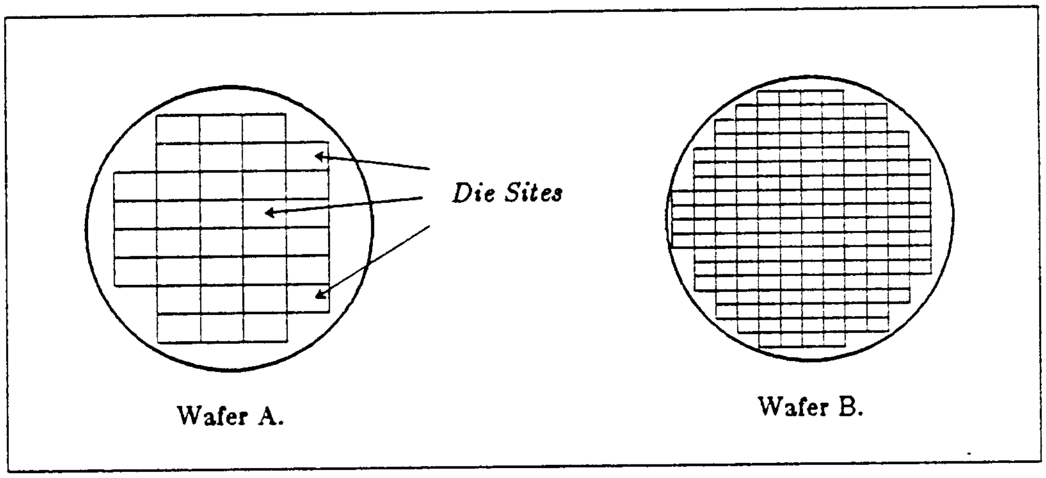


Figure 2.2. - Wafers and Die Sites. Multiple copies of an integrated circuit are fabricated simultaneously on circular silicon wafers. After fabrication is completed, a wafer is fractured (or diced) into rectangular chips (or dice). Each chip contains an individual copy of the circuit. The smaller the area of a chip, the more copies can be fabricated per wafer, and the more economical production becomes.

Limitations on the minimum size of circuit components, and hence the design rules, arise primarily from imperfections in the mask preparation and fabrication process. These imperfections result in distortions of the artwork during the transfer from the original digital specification to the actual integrated circuit layers. They are numerous and varied, arising at every step in the fabrication process. Examples are imperfect optical resolution during certain processing steps, and imperfect alignment between masks or layers.

In addition design rules result from the physical properties of the fabricated circuit. For instance, to prevent shearing of brittle metal lines, some processes have design rules that prohibit metal from crossing over features that rise and fall abruptly. Similarly metal migration effects, involving the erosion of metal atoms under the influence of a strong electric current, necessitate wider metal lines in some cases. The gradual nature of diffusion boundaries also leads to width and spacing restrictions.

Design rules are an abstraction of the physical limitations of the fabrication process that permits the decoupling of process engineering and circuit design. A circuit designer need not be concerned with the complex details of the fabrication process: he need only conform to the design rules. Similarly, process engineers have flexibility in the development or modification of the fabrication process, as long as the final process meets the design rules.

The following sections explore the origin, form, and uses of design rules in more detail. First, the next section takes a closer look at the process engineering side of design rules: the physical structure of integrated circuits is described; the principle artwork transfer steps in the fabrication process are considered; and it is shown how limitations in the fidelity of these transfers, and physical limitations on the fabricated circuit lead to design rules. The following section discusses the form of design rules. A model for design rule form is presented and illustrated with a representative sample of design rules. The final section considers the role of design rules in the various design methodologies, and argues that a basic automatic design rule checking capability is important regardless of design methodology.

2.2. Integrated Circuit Fabrication and the Origin of Design Rules

Integrated circuits consist of layers of patterned conducting material stacked vertically on the surface of a silicon substrate and separated by insulating material. Electrical devices such as transistors and capacitors are formed through the interaction of certain layers across thin insulation. Electrical contacts between layers are formed by cutting through the insulating material between. Regions of the circuit are implanted with various impurities to selectively change the electrical characteristics of the layers and the devices formed by their interactions. A circuit is defined by geometric patterns giving the regions where each of these layers (conducting, insulating or implant) is present.

The physical design of a circuit is originally in the form of digital design files specifying the geometric pattern for each layer. To realize the circuit, the pattern for each layer must be transferred to a physical layer in the circuit. This transfer is generally effected in at least three stages, as illustrated in Figure 2.3. Prior to fabrication a photographic mask patterned with transparent regions on an opaque background (or vice versa) is created for each layer. The photomask is used to pattern a photosensitive resist, deposited uniformly over the surface of the circuit. The actual layer is then patterned from the resist. Typically the layer has been deposited immediately below the resist, and is patterned by an etchant that dissolves the layer everywhere it is not covered by the resist.

Each transfer of the patterning information introduces distortions. The type and degree of distortions is dependent on the procedure and equipment used. Mask generation is often done by a block flash technique. A photographic emulsion is exposed to flashes of light directed through a rectangular aperture, whose position, size and angle of rotation are digitally controlled by a pattern generation tape. The pattern generation tape is derived directly from the physical design files for the circuit. There are several sources of distortion during flashing. Optical diffraction effects at the edge of flashes cause a loss of edge acuity on the emulsion; over- or under-exposure due to imperfect control of the timing and intensity of

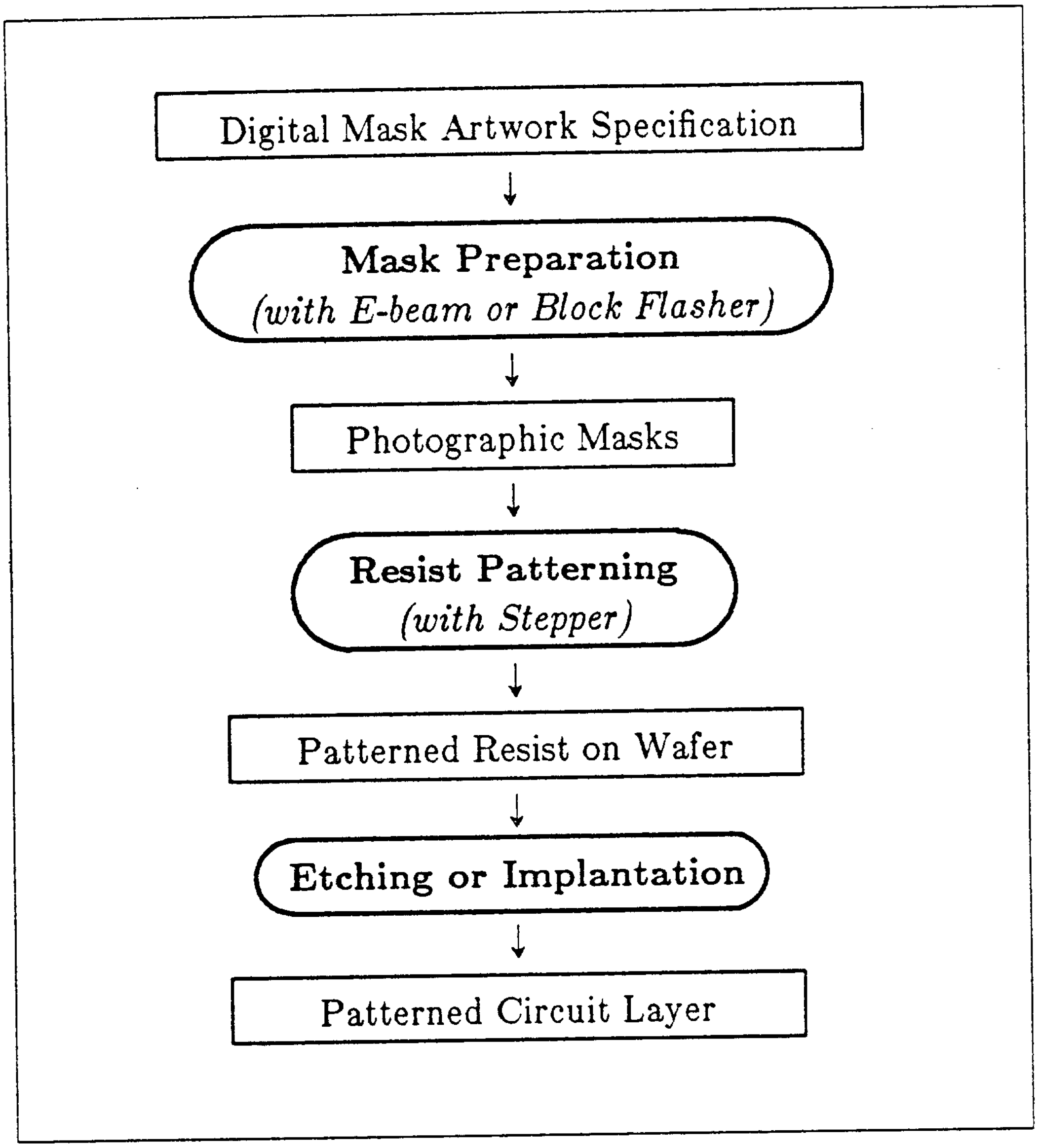


Figure 2.3. - Pattern Transfers in Integrated Circuit Fabrication. The fabrication of integrated circuits generally involves at least three transfers of the geometric pattern information: from the original digital specification to the photographic mask, from the mask to a photosensitive resist on the surface of the wafer, and finally from the resist to the actual circuit layer.

flashes can lead to slight edge motion; and mechanical imprecision in the flasher can result in inaccurate positioning of flashes. Also, decomposition of the original design into a sequence of rectangular flashes may require approximations, since the size, position, and angle of flashes can only be varied in discrete increments. Non-polygonal features, such as circles, require approximation too.

Patterning of the resist typically involves a *stepper* that focuses an image of the photomask successively onto each chip site on the resist-coated wafer. After exposure, the resist pattern is developed using a solvent that dissolves the exposed (negative resist), or unexposed (positive resist) sections. Distortions introduced in the patterning of the resist are due mainly to the limited optical resolution of the imaging system, and diffraction effects at region edges. Distortions also result from other factors such as nonuniform resist thickness. Finally, imperfect alignment between masks, both translational and rotational, leads to uncertainty about the relative position of shapes on different layers.

The final transfer of the artwork pattern to the circuit layer is through an etching or implantation step. The fidelity of this transfer is limited by the diffusion of etchant or implant laterally underneath the resist boundaries. Imperfect control of the reactivity of the active species (e.g. etchant strength), or of timing, results in uncertainty about the size of the regions created.

Some distortion effects are pattern dependent, that is, they vary with the circuit artwork. For example light reflected by metal-coated polysilicon during resist exposure for the metal layer, can cause edges on the metal layer to be displaced; see Figure 2. This reflection effect only occurs when a polysilicon edge runs near a metal edge. Another pattern dependent effect involves etching in open versus confined regions. The rate of etching may be less in more confined spaces. Thus, for instance, the outside edges of a set of parallel metal lines may be etched more than the internal edges, as in Figure 2.5.

The above description of circuit fabrication, while correct in outline is greatly simplified. Processing often involves additional transfers of the artwork pattern. For example working masks may be produced from master photomasks, or reduced (and repeated) masks called reticles may be used. A typical integrated circuit fabrication process involves over 100 distinct processing steps [Sze, 1983]. Imperfections in each step contribute to overall

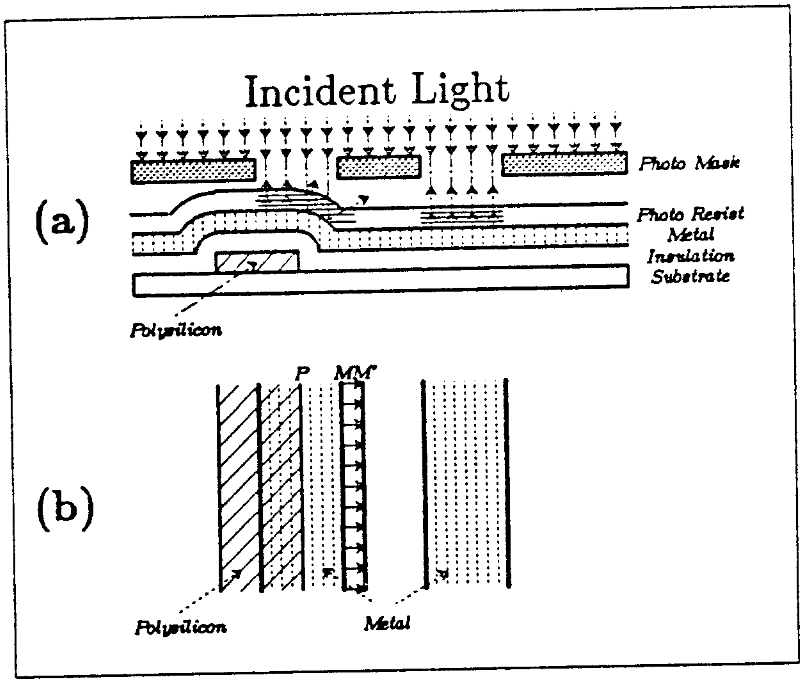


Figure 2.4. - Reflection: A Pattern Dependent Effect. During resist patterning of an aluminum layer, (a) above, the shiny aluminum-coated wafer reflects as much as 90% of the incident light back through the resist. Where the wafer is flat the light is reflected harmlessly straight back up. However, protrusions caused by underlying polysilicon edges scatter the light resulting in undesired resist exposure. This results in the displacement of metal edges where polysilcion edges run nearby, as shown in (b). In this case metal edge M is displaced to M' by light scattered from the nearby polysilicon edge P.

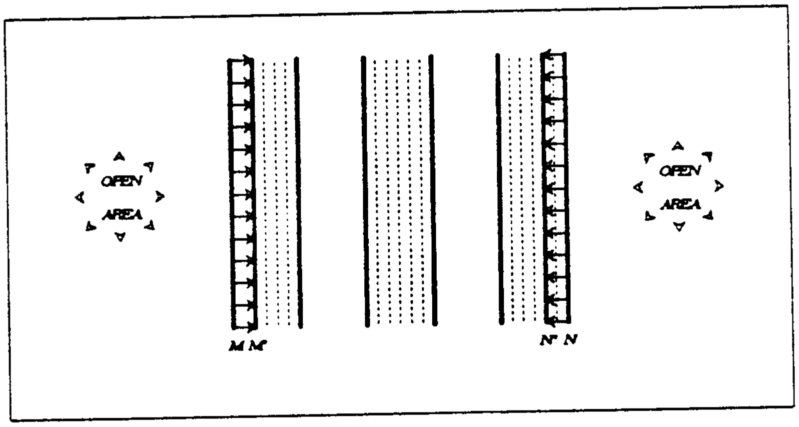


Figure 2.5. - Variable Etch: Another Pattern Dependent Effect. Etching can be more vigorous in open areas, such as outside a set of closely spaced parallel lines, than in confined areas, such as between closely spaced lines. This results in the narrowing of lines adjoining open areas.

distortion in the final circuit.

The sources of artwork distortion during fabrication are numerous and complex. In general the distortions lead to uncertainty about the exact position of region boundaries in the

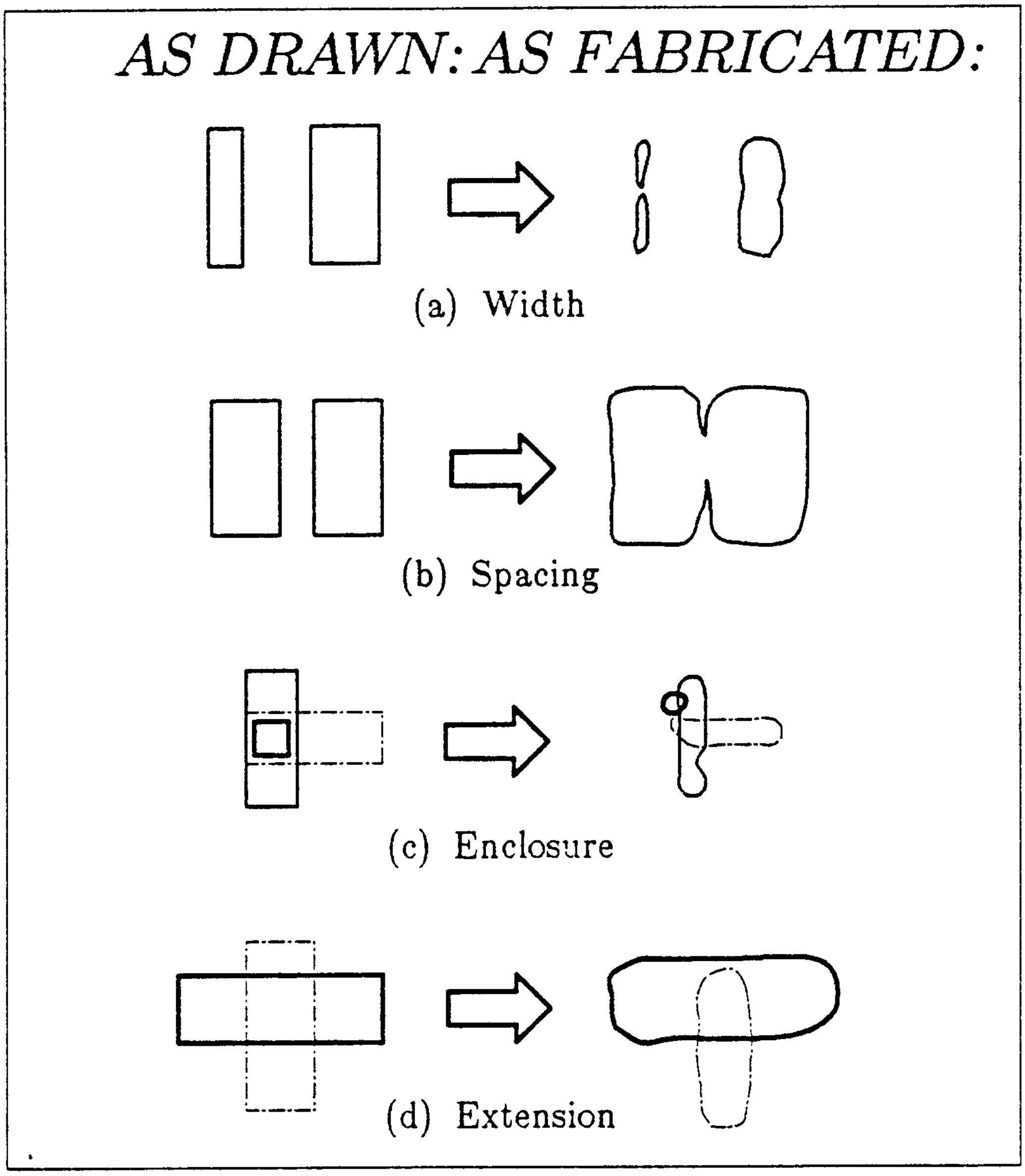


Figure 2.8. - How Distortion Causes Circuit Failure. This figure shows examples of how distortion can lead to circuit failure if minimum tolerances are not observed. Part (a) illustrates how a narrow line can be split. Part (b) shows the merging of features that are drawn too closely together. Part (c) shows how the desired connection between layers can be lost if a contact is drawn with insufficient enclosure. And part (d) shows how transistors can fail, if they are drawn with insufficient extension.

fabricated circuit. Shapes may be slightly larger or smaller than intended, and their relative position, particularly between layers, will be inexact. This is illustrated in Figure 2.6. Narrow lines may not be resolved at all (i.e. may not appear in the fabricated circuit), while somewhat wider lines may be narrowed to the point where they are split into pieces, as in Figure 2.6(a). Similarly shapes that are too closely spaced may be merged during fabrication, as in Figure 2.6(b). Thus minimum width and spacing tolerances are needed to ensure that electrical nodes are neither split nor merged together during fabrication.

The formation of contacts and devices such as transistors and capacitors involves overlaps and extensions between layers. For example, in a typical MOS process a contact from the metal layer to the diffusion layer is formed as shown in Figure 2.1(c). In order for contact to be made it is necessary that all three layers, metal, diffusion and the cut in the insulation between them, overlap. Thus minimum overlap tolerances are needed to ensure all three layers will overlap sufficiently in the fabricated contact despite misalignments between layers, and shrinks on the individual layers; see Figure 2.6(c).

The formation of MOS transistors involves the extension of a polysilicon line over a diffusion line, as in Figure 2.1(d). The transistor will not function if the polysilicon fails to extend all the way across the diffusion line in the fabricated circuit, (see Figure 2.6(d)), so a minimum tolerance on the extension of the polysilicon *beyond* the diffusion is required.

In addition to causing circuit failure through broken or shorted nodes and inoperative contacts or devices, distortions can result in degradation of circuit performance through the formation of parasitic devices, e.g., capacitors formed by unintended overlap between layers, or voltage drops in lines that end up too narrow. Cumulatively such effects can lead to complete functional failure of the circuit, or simply degrade performance so the circuit will not meet design specifications. In a few cases, distortions can result in long-term failure of the circuit. For example, narrow metal lines carrying high current are subject to eventual failure due to metal migration: the metal atoms actually erode away under the influence of the electric current.

Many of the factors contributing to artwork distortion during fabrication are *random*. They vary from wafer to wafer, and often from region to region on an individual chip, according to some probability distribution. The distortion of individual shapes and relationships between shapes results from the combined effects of these many random factors, and thus is best characterized in statistical terms. The maximum distortions occur when all the individual factors are by bad luck near their respective maxima and all work together in the same direction. The greatest distortion on an average chip will be much less than such a worst case since the many factors contributing to distortion will tend to average out and cancel each other.

Specifying design tolerances based on worst case distortions would be overconservative. Such a choice would lead to unnecessarily large devices and circuit areas, degrading circuit performance, increasing power consumption and quite possibly *reducing* the yield of working chips because of fatal random defects whose probability increases exponentially with chip area. Thus the specification of design rule tolerances is a compromise that seeks to make minimum tolerances small while, at the same time, keeping losses from pattern distortion low.

2.3. The Form of Design Rules

The last section showed how numerous factors cause distortions of the design artwork during fabrication. These distortions result in uncertainity about the eventual size and relative position of artwork shapes in the fabricated circuit. Coupled with physical requirements of the circuit, such as the need to maintain the integrity of nodes and minimize parasitic devices, these effects lead to the design rules for the process.

Design rules take the form of minimum tolerances on spacings, widths, enclosures and extensions on the artwork as designed. These tolerances are intended to be sufficiently large so that the corresponding relationships in the fabricated design will be maintained (and of sufficient dimension) despite distortions. Because these rules give tolerances on topological relationships, I refer to them as *topo-tolerance* rules.

The simplest and most common design rules specify minimum width and spacing for the nodes of a single layer. See, for example, Figures 2.1(a) and (b). Here diffusion lines are required to be at least 2 units wide and distinct nodes are required to be spaced at least 3 units apart.

Enclosure and extension rules generally involve interlayer contact or device constructs. For example Figure 2.1(c) shows a typical rule for contacts between diffusion and metal in an nMOS process. Here both the diffusion and the metal must enclose the contact cut for a distance of at least one unit. Figure 2.1(d) shows an nMOS transistor rule that requires polysilicon to extend past the diffusion for at least two units.

In addition to specifying tolerances on topological relationships for single layers and between pairs of layers, topo-tolerance rules can refer to regions defined by a combination of layers. For instance in nMOS processes, the operating characteristics of a transistor can be changed by implanting the transistor gate region: unimplanted transistors are enhancement mode, while implanted transistors are depletion mode. To ensure that the entire gate region of depletion mode transistors actually gets implanted, the implant layer is required to enclose depletion mode gate regions, by some tolerance. Similarly, to avoid accidental implantation of enhancement mode transistors, there is a spacing rule between enhancement mode gate regions and the implant mask. These rules are illustrated in Figure 2.7. Note here that depletion mode gate regions are defined as

Polysilicon **AND** *Diffusion* **AND** *Implant*,

and enhancement mode gate regions are defined as

Polysilicon **AND** *Diffusion* **AND** (**NOT** *Implant*).

In general topo-tolerance rules can refer to regions on *composite* layers defined as arbitrary combinations of the mask layers. Such combinations are specified by boolean expressions on the mask layers, e.g. by using **AND**, **OR** and **NOT** operations.

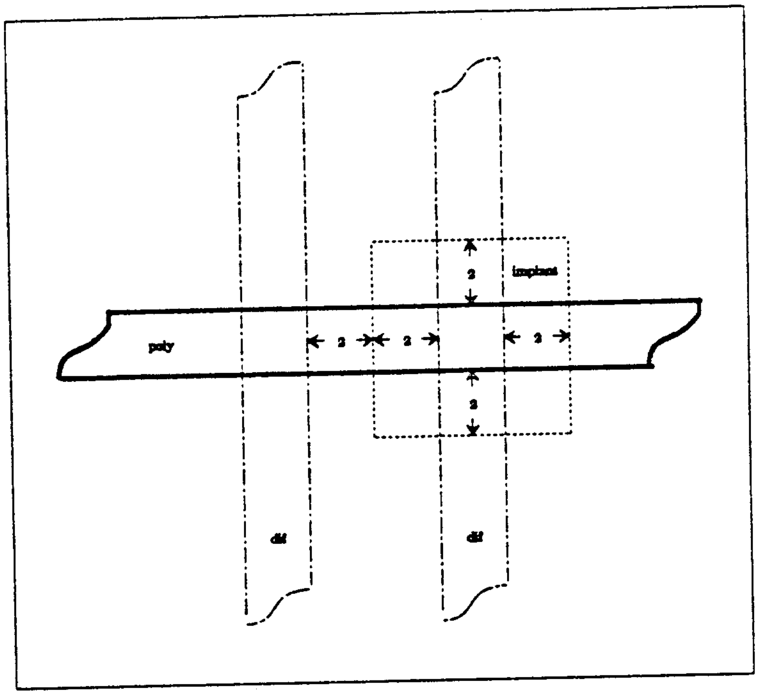


Figure 2.7. - Implant Tolerances Involving Combinations of Layers. Typical implant rules require implanted transistors to be enclosed by implant for a minimum of 2 units and nonimplanted transistors to be spaced at least 2 units from implant regions. These rules do not involve spacings between mask layers; rather they each involve a spacing between a mask layer and a (boolean) combination of mask layers.

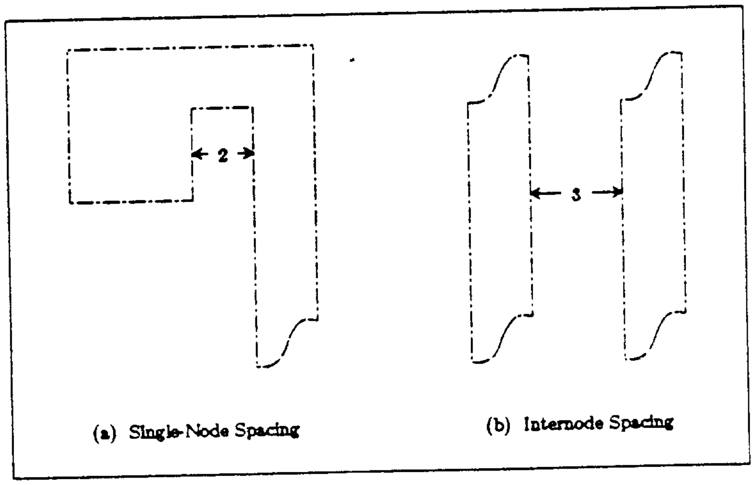


Figure 2.8. - Internode Spacing; a Conditional Rule. Frequently the spacing rule for distinct nodes specifies a greater minimum spacing than is required between segments of a single node. Such a spacing rule is *conditional* because it only applies in limited contexts, i.e., between boundary edges of distinct nodes.

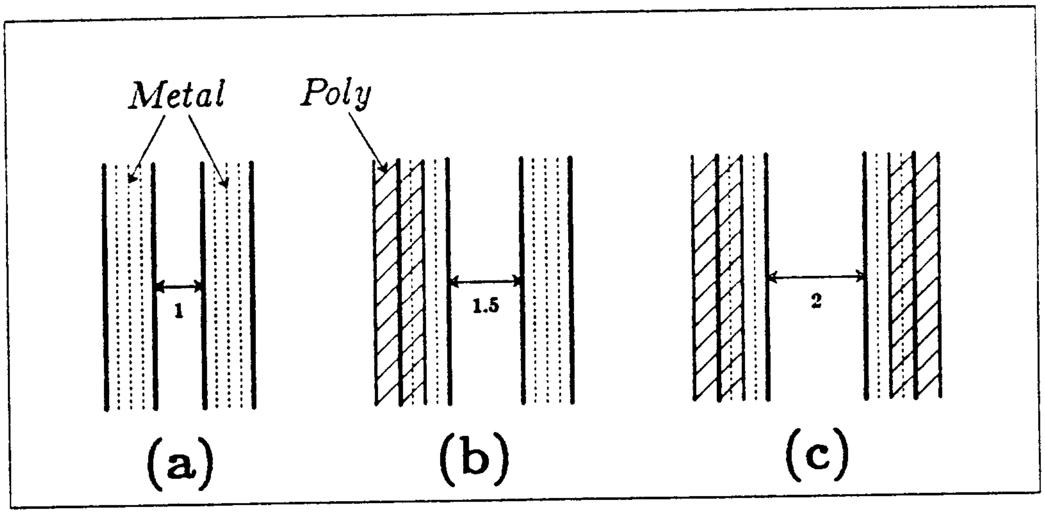


Figure 2.9. - Reflection Rule. The spacing between metal lines may be greater when polysilicon edge(s) are nearby. This is because reflections from the metal coated polysilicon edges during patterning of the metal layer can cause the metal edges to move outward.

The above rules, with the exception of internode spacing, are unconditional: the specified spacings, widths, or enclosures apply unconditionally to the mask layers throughout the circuit. However design rules are often conditional, that is, the specified topo-tolerance is only relevant in certain contexts. The conditions on design rules can take many forms. The most common example of a conditional rule is internode spacing on a conducting layer. Such a spacing rule is conditional because it only applies to mask regions that belong to distinct nodes; see Figure 2.8. Another example of a conditional design rule involves the reflection phenomenon introduced in the previous section. To take reflection into account a rule might require that metal lines be spaced 1 unit apart when there are no polysilicon edges paralleling the facing metal edges, 1.5 units apart when a polysilicon edge runs near one of the two metal edges, and 2.0 units apart when polysilicon lines run near both the metal edges. Such a rule is illustrated in Figure 2.9. Some design rules require that long parallel lines be spaced more conservatively than short ones, to avoid capacitive coupling. Figure 2.10 gives an example. A tendency for etching to be more vigorous in open areas can lead to conditional width rules. For example minimum metal width might be 8 units if no other metal is present nearby, 7 units if metal is nearby on one side, and 6 units if metal is present on both sides of a line. Such a rule is illustrated in Figure 2.11. To avoid metal migration effects, metal width is sometimes dependent on the current density a line will carry; for instance, power and ground lines are often required to be wider than lines carrying other signals.

Conditional rules can get very intricate. As a final example, consider the following: To avoid metal shearing due to rough underlying terrain, greater spacing between polysilicon and diffusion edges may be required when a metal line runs perpendicularly across these edges; see Figure 2.12. Since shearing in the direction of current does not pose a problem, such a rule is actually dependent on the direction of current in the overlaying metal.

The above examples illustrate conditional rules that depend on the presence of nearby regions or edges on the same as well as different mask layers, rules that depend on the length, expected current density or function of lines, and even a rule that depends on current direction. In general design rule conditions may be very complex and may involve geometric information about circuit artwork, topological information (such as node connectivity), electrical information, and functional information about the circuit.

With the exception of an occasional rule concerning areas, perimeters, or exact (not minimum) dimensions, the examples of this section illustrate the nature and range of design rules for integrated circuits. Simple rulesets are comprised of a relatively small number of conservative primarily unconditional rules giving minimum tolerances on widths, spacings, enclosures and extensions for mask layers and layer combinations. More complex rulesets involve a greater number of conditional rules that give more precise tolerances by specializing the context in which each rule applies.

2.4. Automatic Design Rule Checking and Design Methodology

The topo-tolerance rules defined in the last section form the basis for the interface between process engineers and circuit designers. However, the nature of the design rules actually seen by circuit designers varies considerably with design methodology. This section

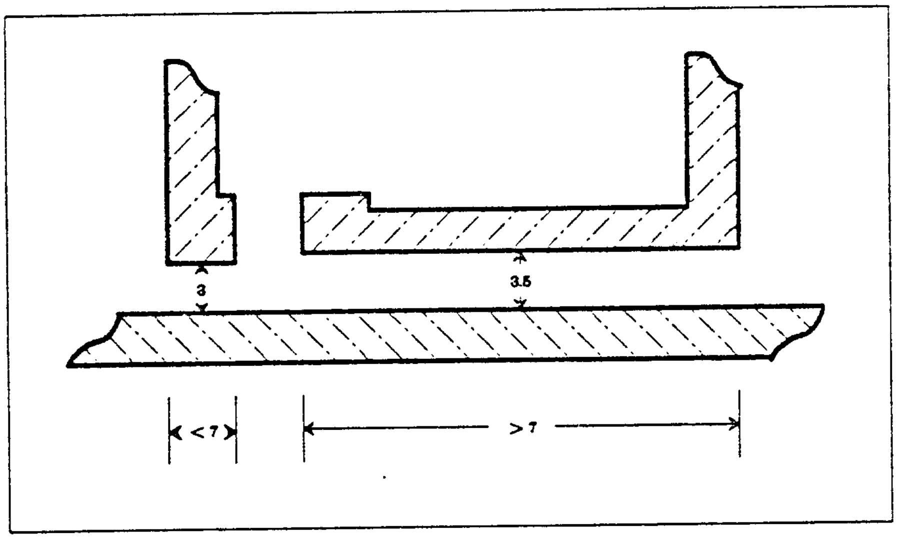


Figure 2.10. - Length-Dependent Spacing. Sometimes spacing rules are conditional on the length over which lines run parallel. For example, lines might be required to be separated by 3 units if they run parallel for less than 7 units, but be separated by at least 3.5 units if they run parallel for lengths of 7 units or more.

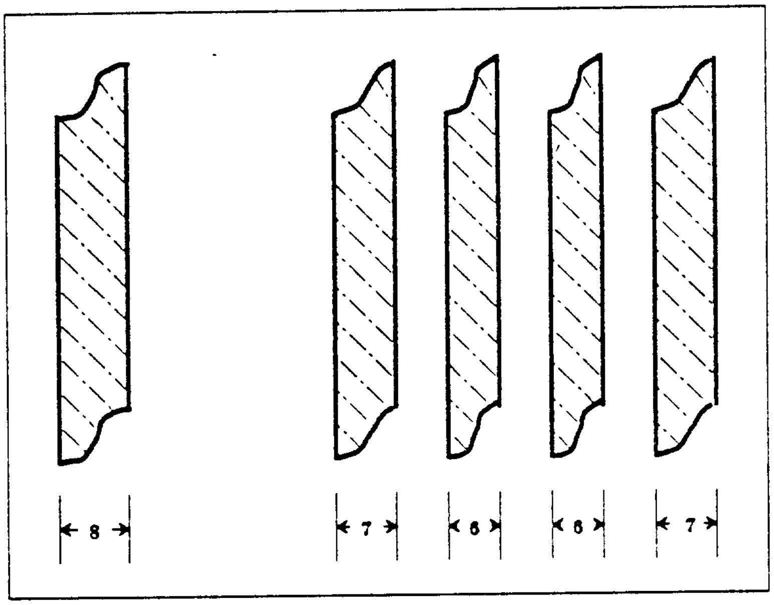


Figure 2.11. - Density Dependent Width. Because etching is more vigorous in open areas, width rules are sometimes conditional on the presence of nearby lines on the same layer. For example minimum width may be 6 units for interior minimum-spaced lines, 7 units for lines at the edge of a minimum-spaced set, and 8 units otherwise.

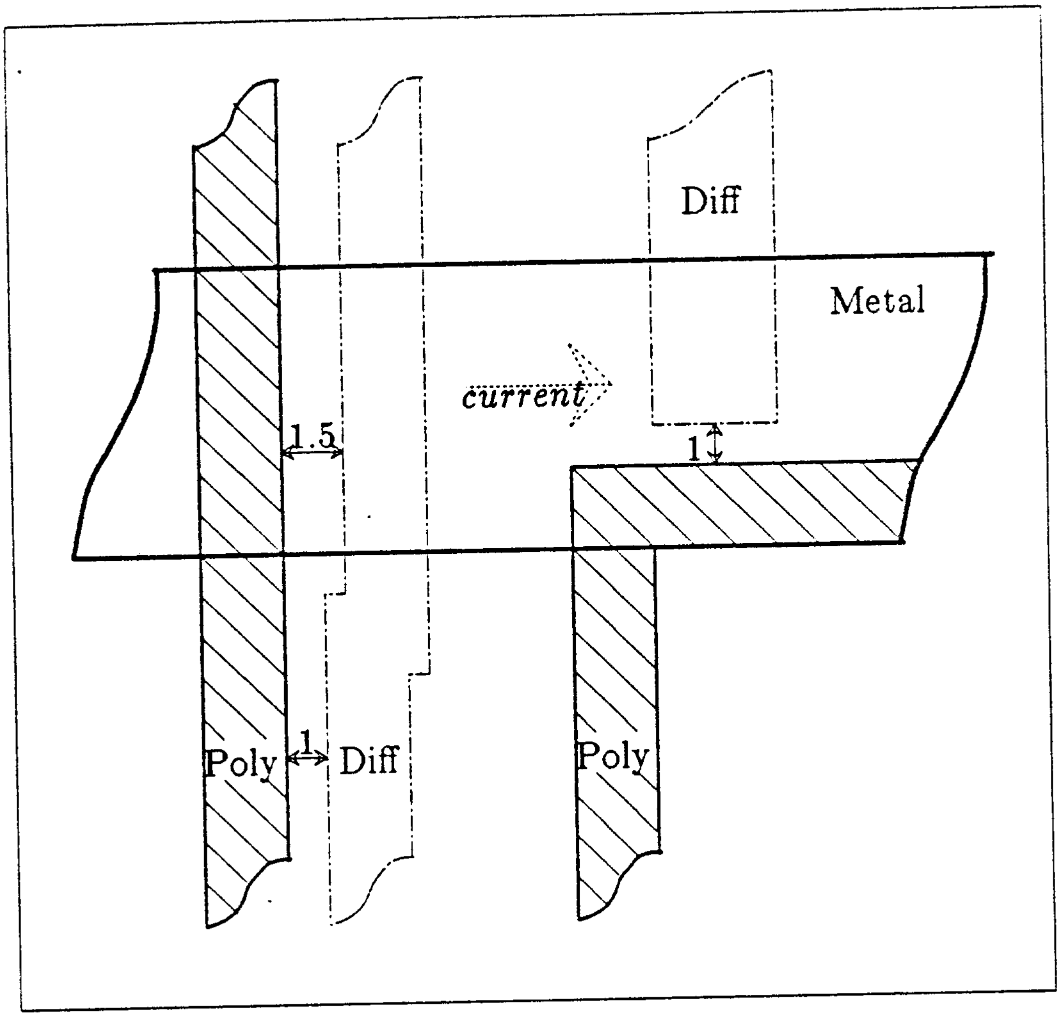


Figure 2.12. - Spacing Conditional on Current Direction in Overlying Metal. Sometimes minimum spacing between poly and diffusion is conditional on the presence of overlying metal. This is because nearly-coincident poly and diffusion edges result in an abrupt change in the vertical dimension that can cause overlying metal to shear. Since shearing in the direction of current flow does not cause problems, the more conservative spacing need only apply to polysilicon and diffusion edges running across the direction of current in the overlying metal.

explains why a spectrum of design methodologies is in use, and briefly describes the major methodologies and the nature of the design rules each presents to the designer. Then it argues that automatic topo-tolerance checking is important regardless of the methodology employed.

2.4.1. The Spectrum of Methodologies

Design methodology and design automation are receiving much attention. There is a large spectrum of design styles in use, ranging from highly constrained, highly automated, low density, gate-array designs to full-custom designs finely tuned to a specific process to achieve maximum density and performance. These methodologies differ in the tradeoff they make between simplifying design constraints on the one hand and ultimate circuit size and performance on the other; see Figure 2.13. The nature of design rules and other constraints employed by the methodologies will be outlined below.

In general, properly chosen constraints can simplify the design process. Such simplifications improve designer productivity directly, since design decisions can be made

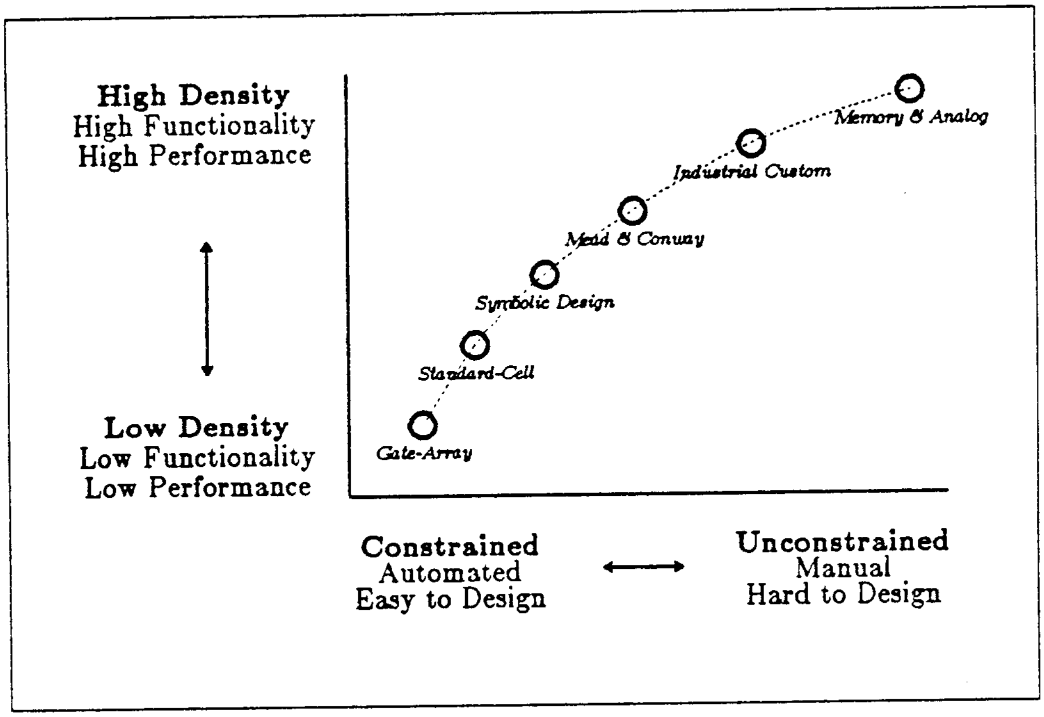


Figure 2.13. - Design Cost/Density Tradeoff. There is a basic tradeoff between the degree to which a design method is constrained, and hence automated, quick, and painless on the one hand, and the penalty in circuit density, performance, and functionality engendered by those constraints on the other. Different positions with respect to this tradeoff are appropriate to different projects. A wide spectrum of design methods ranging from fully automated gate-array to hand-tailored analog design are in use.

more quickly and accurately, and indirectly since they facilitate the automation of the design process. On the other hand, more constrained design styles generally lead to less dense and less efficient circuits, resulting in more stringent limits on the maximum functionality per chip, lower performance, and increased production cost. The best methodology to use for a particular product depends on the functionality and performance required, the volume of chips that will be produced, and the particular mix of resources available for design and fabrication. However, it is apparent that over time, as fabrication technology continues to improve, more constrained methodologies will be increasingly favored.

2.4.2. Design Rules and Other Constraints Employed by the Methodologies

Table 2.1 summarizes the characteristics of a number of design methodologies. The methods are given in order of increasing design rule complexity. In fully automatic gate array designs [Soukup 1981] the designer does not deal with topo-tolerance rules at all; he works at the netlist level, specifying the gate interconnections required to implement the circuit. A designer using the standard cell approach [Soukup 1981] need not deal with a full set of topo-tolerance rules either, since all devices and hence device rules are encapsulated in the predefined cells. Designers using symbolic design systems [Bales 1979] [Hsueh 1979] work with a more abstract representation than mask data: transistors, contacts, and their interconnections are represented explicitly. Layout is generated automatically and is usually guaranteed to be design-rule correct. The remaining methodologies, i.e., Mead-Conway [Mead & Conway 1980], traditional custom, and memory & analog design, all work directly with the mask layer specification, and all require a full set of topo-tolerance rules. They differ mainly in the number and complexity of the rules employed.

2.4.3. The Need for an Automatic Topo-Tolerance Checking Capability.

Regardless of the design methodology used, there must be some automatic method for guaranteeing that the final mask artwork for the design satisifies the topo-tolerance rules for

Design Methodologies		
Method	Description & Constraints	Design Rules
Gate-Array	Predefined, fixed, regular, array of gates with horizontal and vertical wiring channels. Only netlist is specified by designer. Gate assignment and routing of netlist is done automatically.	None.
Standard-Cell	Predefined library of fixed-height cells. Power and ground routed horizontally through cells at standard locations. Designer selects cells of desired functionality, places cells in rows and routes the signals. External pads must also be placed and routed, and power and ground connections made. No transistors are permitted outside cells. Routing is usually semi-automatic.	Simple width and spacing rules for wiring, with stylized contacts. No transistor rules, since all devices encapsulated in predefined cells.
Intelligent Layout Systems.	Designer places and routes devices freely using abstract representations for devices and interconnect. Abstract representation permits stretch/compact operations that preserve integrity of design. Upon completion, abstract representation automatically converted to mask layers.	Simple width and spacing of interconnect and devices must be considered during automatic generation of mask data. More complex device form rules are not relevant since correct, stylized, devices are automatically generated. Simple rules make automatic stretch/compact feasible.
Mead-Conway.	Designers strive for regular placement and interconnection schemes that take full advantage of the topological properties of the implementation medium. Simple conservative design rules are used to free designer from messy low-level details.	Full set of topo-tolerance rules. Rulesets are simple, i.e., a small number of mostly unconditional rules.
Traditional Full Custom.	Devices are placed and interconnected with emphasis on high density and performance. A set of stylized layouts for gates, memory cells, etc., is developed and used whenever practical.	Elaborate full topo-tolerance ruleset. Conditional rules allow more precise tolerances to be used, permitting denser design.
Memory and Analog Designs.	Designers work closely with a particular process to achieve maximum performance and density.	Large complex rulesets with many conditional rules. Rules often depend on anticipated power and signal strength in particular regions of a circuit.

Table 2.1

the particular process being used: manual checking is unacceptable. The size and complexity of VLSI designs makes manual checking an extremely tedious and error prone process. Even experienced people concentrating on a single rule with diligence miss rule violations [Fitzpatrick et. al. 1981]. Trial fabrication runs on circuits are time consuming (turn around is typically several weeks to several months) and expensive. In addition it is difficult to trouble-shoot finished circuits, even for fatal DRC violations that render the circuit completely nonfunctional. Nonfatal violations contributing to reduced yield, reliability, and performance of the circuit are likely to go completely undetected during circuit testing. Thus the mask artwork must be automatically generated, in a design rule correct way, or accurate automatic design rule checking must be used to eliminate all design rule violations prior to fabrication.

In fact an automatic topo-tolerance checking capability is important regardless of the methodology employed. Although the more constrained methodologies shield the user from much of the detail of topo-tolerance design rules, a topo-tolerance checking capability is still important. Topo-tolerance checking is used for the development and maintenance of systems employing constrained methodologies. For instance the library cells in standard cell systems and the templates for gate-arrays must be verified with a full topo-tolerance DRC. Further, automatic layout generation systems are complex and hence prone to error. Topo-tolerance checking is required to verify the correctness of automatically generated layout.

Topo-tolerance checking is also used to verify that no errors have been introduced during the composition of independently-generated pieces of a circuit into a complete design. Designs often combine elements developed with a variety of tools and within the framework of a variety of systems. Integration and composition of these components is an error-prone process involving multiple format conversions and often manual intervention to complete final placement and route where automatic tools are not available or not quite adequate. Topo-tolerance checking on the mask data has the advantage that it is done on the final representation of the design, and thus can catch errors introduced during the final conversion and integration steps: it provides a check on the correctness of *all* the steps leading to the final design.

This thesis focuses on the automatic checking of full topo-tolerance rules on the mask data for designs.

2.5. Summary

Dense designs are more economical, have better performance characteristics, and allow greater functionality per chip. Design rules codify limitations on the minimum size of circuit components, and hence on the maximum density achievable. These limitations originate from numerous distortions introduced during circuit fabrication and to some extent from the physical characteristics of the fabricated circuits.

Design rules provide an interface between the process engineers and the circuit designers. The process engineers need not concern themselves with the details of the circuits being fabricated as long as they can meet the specified design rules, and the circuit designers need not concern themselves with the details of the fabrication process, as long as they obey the design rules.

Design rules take a topo-tolerance form, that is, they specify tolerances on certain spacing, width, overlap, and extension relationships in the circuit artwork. The intention is that these tolerances are sufficient to maintain the relationships to some minimum acceptable dimension despite process distortions. In general topo-tolerance rules involve relationships between regions defined by combinations of mask layers. Many rules are unconditional: the specified tolerances apply to the artwork relationships wherever they occur throughout the designs. However rules can be conditional on nearby artwork on the same or different layers, on topological relationships (such as node connectivity), on electrical properties, and even on information about the function of the circuit.

Though topo-tolerance rules are the basis for the designer/process interface, the nature of the rules a designer sees directly varies with design methodology. Design methodologies range from highly constrained, automated, low density, methods such as gate-array, to hand-tailored, high-density, high-performance custom designs. The more constrained methodologies shield the designer from low-level details such as a complete set of topo-tolerance rules. Designers using these systems work with simpler, more abstract design rules.

Full topo-tolerance checking of mask artwork data is important regardless of the design method. This is true for several reasons: complex automated systems can make mistakes, manual intervention into automated systems can introduce errors, and the final composition of a design from independently generated pieces is often ad hoc and error prone. This thesis focuses on full topo-tolerance checking of mask data.

2.6. References

For a general introduction to VLSI design, see [Mead & Conway 1980]. This text also presents the famous Mead-Conway method of design, and gives a complete set of design rules for nMOS. For more details on processing technology, see [Sze 1983]. The Lyon paper [Lyon 1981] presents an interpretation of design rules in terms of edge motion. Three examples of symbolic layout systems are given by [Bales 1979], [Hsueh 1979], and [Ousterhout 1984]. For an introduction to gate-array and standard cell techniques, see [Soukup 1981].

[Bales 1979]
M.W. Bales, *Layout Rule Spacing of Symbolic Integrated Circuit Artwork*, MS Thesis, UCB/ERL M82/72, University of California, Berkeley, CA, May 1982.

[Fitzpatrick, et. al. 1981]
D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Sequin, R.W. Sherburne and K.S. Van_Dyke, "A RISCy Approach to VLSI," *VLSI Design*, Vol. 2, No. 4, Fourth Quarter 1981, pp. 14-20.

[Hsueh 1979]
M.Y. Hsueh, *Symbolic Layout and Compaction of Integrated Circuits*, PhD Thesis, UCB/ERL M79/80, University of California, Berkeley, CA, December 1979.

[Lyon 1981]

R. Lyon, "Simplified Design Rules for VLSI Layouts," *VLSI Design*, Vol. 2, No. 1, First Quarter 1981, pp. 54-59.

[Mead & Conway 1980]

C. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, 1980.

[Soukup 1981]

J. Soukup, "Circuit Layout," *Proc. of the IEEE*, Vol. 69, No. 10, October, 1981, pp. 1281-1304.

[Sze 1983]

S.M. Sze, *VLSI Technology*, Mcgraw Hill, New Jersey, 1983.

CHAPTER 3

The Region-Operation Approach

3.1. Introduction

This chapter presents the region-operation approach to design rule checking which, with variations, is used throughout the industry. A region-operation based system consists of a collection of primitive operations, each of which takes one or two layers as input and generates an output layer. There are several types of operations. *Tolerance operations* check topo-tolerances between layers and output the portions of regions that are in violation. Tolerance operations can be preceded by *boolean operations* to select layer combinations for checking, and by *topological, sizing, connectivity* and other operations to select regions for conditional checks. Since each operation is independent and yet can be combined with any other, a region-operation system is extremely flexible: operations can be sequenced together as desired, and new operations can be added whenever needed without disturbing the integrity of the system. Region-operation systems provide an integrated solution to mask artwork processing: a comprehensive set of operations is used that can handle circuit extraction (i.e. the extraction of transistor connection networks, capacitances and other electrical information), and other mask artwork functions as well as design rule checking. The operations intended primarily for non-DRC functions still serve to enrich the DRC, enabling a variety of conditional checks.

The main drawback of the region-operation approach is that each operation independently requires a pass through two input layers and the generation of an intermediate output layer. Since mask layers for large circuits contain hundreds of thousands of figures, and since hundreds of operations are required to implement a complete design rule check, design rule checking with region operations requires a deal of I/O and hence is slow. Another problem with the region-operation approach is that direction-sensitive rules, often involved in transistor or contact form, are notoriously difficult to check. These problems are considered in detail later in the chapter, after the form, function, and implementation of region-based systems are discussed.

The next chapter, which surveys DRC systems, presents a number of examples of region-operation systems.

3.2. Mask Artwork and Mask Artwork Functions

In the region-operation approach, design rule checking is done in the context of a more general mask artwork processing system. Such mask artwork systems are very similar, consisting of similar sets of operations implemented in similar ways. This similarity in form is due in part to similarity in function: they all process mask artwork, and all provide the same basic functions. This section introduces this common ground: it presents the form of mask artwork, and the functions typically performed on it.

Mask artwork files specify the two-dimensional geometric pattern of regions for each layer in the circuit. Artwork file formats have the following characteristics:

i. Regions are defined in terms of primitive closed *figures* such as rectangles, trapezoids, polygons and round flashes. Figures are identified with individual mask layers. Often there is one file per layer.

ii. Figures on a layer are in general allowed to abut and overlap. However some formats do not allow overlap between the figures composing a single layer.

iii. There is usually a *symbol/instance* mechanism, which allows a symbol consisting of a collection of primitive figures (and possibly *instances* of other symbols) to be defined. A design may contain multiple instances of any given symbol. Each instance has an associated translation and rotation, which specifies where the symbol instance is to be placed. Symbol instances may be nested to structure a design hierarchically. Hierarchy simplifies the design process by allowing modular structuring of designs, allows far more space-efficient representation of repetitive designs, and can be exploited to expedite some processing of designs, such as design rule checking (see Chapter 8).

In addition to design rule checking, region-operation systems provide for *circuit extraction*, and *compensation* functions. Circuit extraction involves recognizing transistors in the mask artwork, establishing their interconnection network, and determining electrical parameters such as capacitances. (Extracted circuits are compared with original schematics, checked for electrical rule violations, and used as input to circuit simulators.) Compensation inviolves growing and shrinking of regions on certain mask layers to accomodate peculiarities of a particular process. An advantage of this integrated approach to mask artwork functions is that operations intended primarily for circuit extraction or compensation functions are never-the-less available to the DRC, enabling a variety of conditional checks.

3.3. Region Operations

Most mask artwork systems use the region-operation approach to implement design rule checking, circuit extraction, and compensation functions in one integrated system. Each function is achieved through an appropriate sequence of region-operations. The following types of region-operations are usually provided:

i. Tolerance

ii. Boolean

iii. Topological

iv. Sizing

v. Connectivity and Tagging

vi. Area, Length and Perimeter

Each operation takes one or two layers as input and generates an (intermediate) output layer, typically consisting of selected or modified portions of the input. In the case of tolerance operations the output consists of the parts of input regions involved in design rule violations. Some of the operations involved in extraction, e.g., connectivity and area operations, generate numerical output such as node numbers and node areas, or tag the mask data with such information. This will be considered further when the individual operations are discussed.

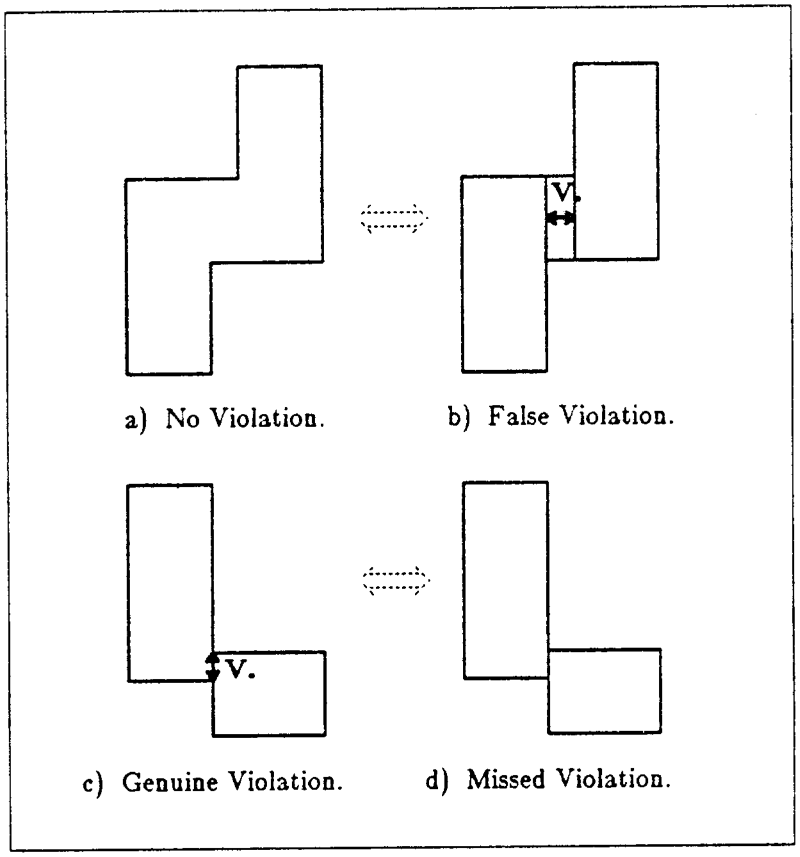


Figure 3.1. - Region-Based vs. Figure-Based Operations. A width check based on figures (b) rather than the regions formed by the figures (a) can lead to spurious violation reports. Conversely, parts (c) and (d) show how a figure-based width check can miss violations. In general it is important that operations work on regions as a whole, rather than the figures composing the regions.

Operations are *region* not *figure* based, e.g. a width check verifies the width of entire connected regions, not of figures in the mask artwork description. Figure 3.1 illustrates why this distinction is important: region-based operations avoid pathological dependencies on the decomposition of mask regions into figures. Region operations are implemented using edge-based algorithms that operate on the region boundary edges. The input mask data is converted to such *edge-files* at the beginning of processing, and the output of region-operations is in boundary-edge form.

The edge-files used by the region-operations differ from the mask artwork representation in another important respect: mask artwork files are usually hierarchical, containing symbol definitions and instances, while edge-files are generally flat. During conversion to boundary-edge form, symbol instances in the original mask artwork are replaced by their definitions.

Since most region-operations read and write boundary-edge files, they can be freely combined. All of the types of operations listed above, regardless of their primary function, contribute to the design rule checker. For example, sizing operations, though needed primarily to implement grows and shrinks for compensation, are useful for establishing certain contexts in conditional design rule checking. Similarly, connectivity operations, needed primarily for circuit extraction, also allow for the checking of conditional design rules depending on connectivity. Each type of operation is discussed below, with emphasis on the role it plays in design rule checking.

3.3.1. Tolerance Operations

Tolerance operations check *width*, *spacing*, *enclosure* and *extension*. Tolerance operations take one or two layers and a tolerance as input, and output boundary-edges that are too close together, as shown in Figure 3.2. Violating edges are often thickened and output as regions, to facilitate plotting or further processing.

A spacing check between layers A and B for distance n can be visualized as follows: An n wide halo is drawn around each region on layer A and checked for intrusion by layer B, see Figure 3.3. Such a halo can be constructed piecemeal, creating an n-deep outward-facing box adjacent to each boundary-edge and joining together these boxes with circular sectors at each convex corner, see Figure 3.4. When an edge intrudes into a halo, both the intruding edge, and the edge giving rise to the particular halo piece are output. Checking halos for intruding edges can miss situations where a region on one layer completely encloses a region on the other, thus a halo based spacing operation must also check for, and flag, overlaps between the layers.

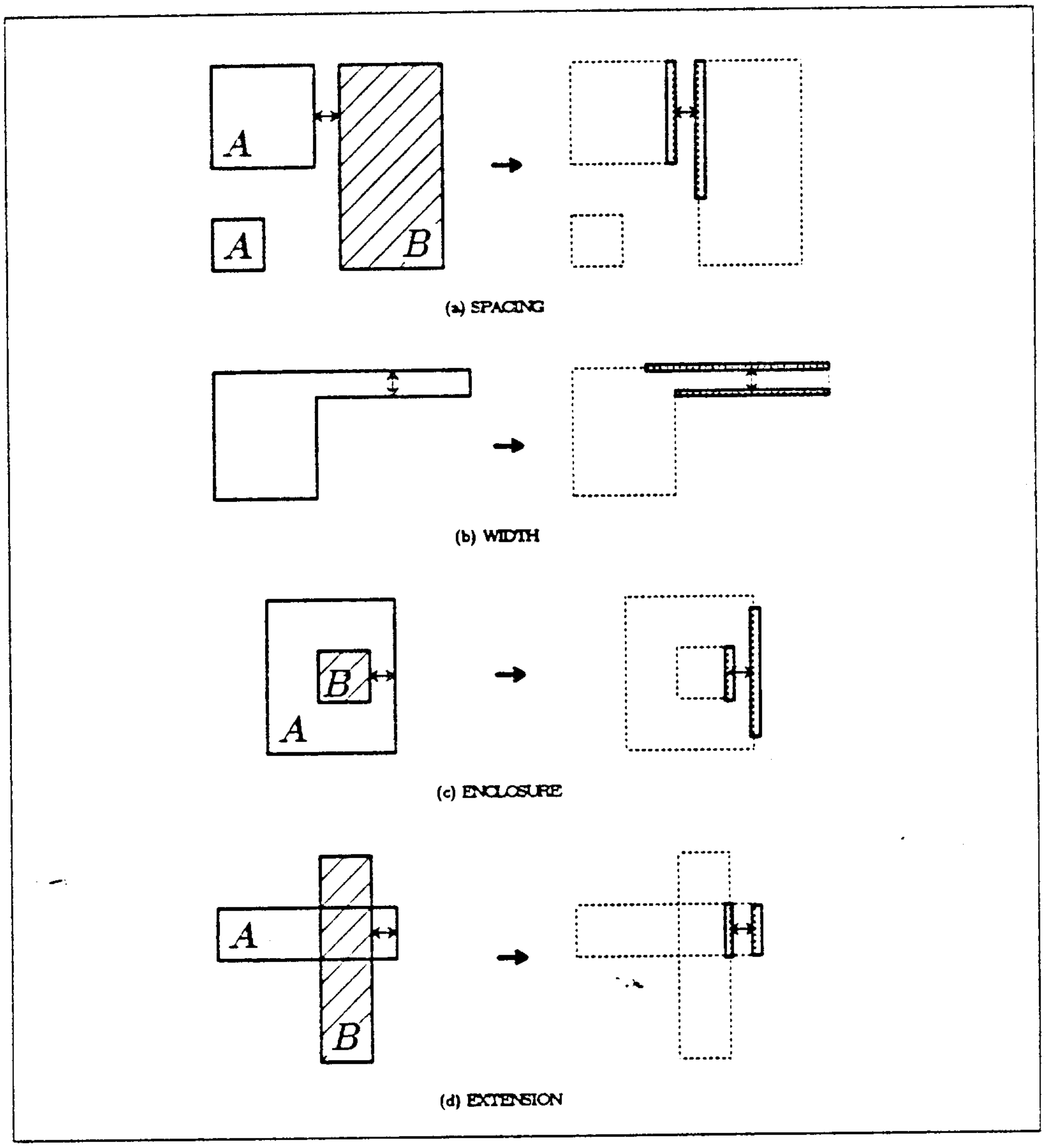


Figure 3.2 - Tolerance Operations. Tolerance operations output (portions) of edges that are too close together. Often edges are "thickened" into regions to permit plotting or further region-operations. The dotted versions of the input regions regions are for reference: they are not actually part of the output.

In some cases it is desirable to check spacings perpendicularly outward from region boundaries, but not diagonally out from corners. Such checks can be done using halos without corner sectors, as in Figure 3.4(a). This sort of check is appropriate, for example, in facing-edge rules; see Figure 3.5. Such rules guard against the formation of long narrow slivers of resist during processing, which could physically break off and deposit themselves elsewhere.

In single-layer spacing checks it is often desirable to avoid reporting situations where a region intrudes into its own halo; see Figure 3.6. A variant of the **SPACING** operation is usually provided for this purpose. Such operations require prior tagging of edges with node numbers, via the connectivity operations discussed below.

Spacing and other tolerance operations usually have provisions for user-supplied filter routines. Such routines can be used to implement less common or more complicated conditional checks, depending on node numbers or other information tagged to edges. Tolerance operations pass the filter routines each pair of edges that violate the specified tolerance, along with all the information tagged to them. Based on this information the filter routines determine whether the edge pairs should be output or not. This facility is very powerful. It allows tolerance checks to be conditional on connectivity, edge length, region

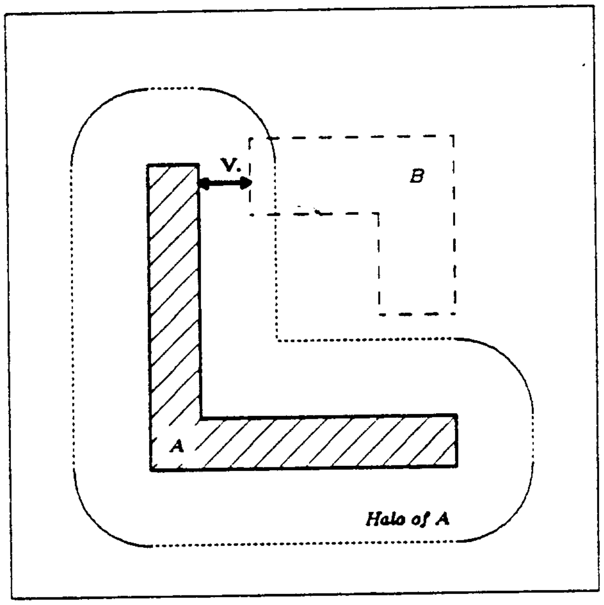


Figure 3.3. - Tolerance Checking with Halos. Interlayer spacing between layers A and B is verified by checking for the presence of layer B within a halo around A. In general tolerance checks can be done by checking for the presence of the appropriate layers in the appropriate outward- or inward- facing halo regions. Actual overlap between regions must also be checked for, since one region may completely enclose another.

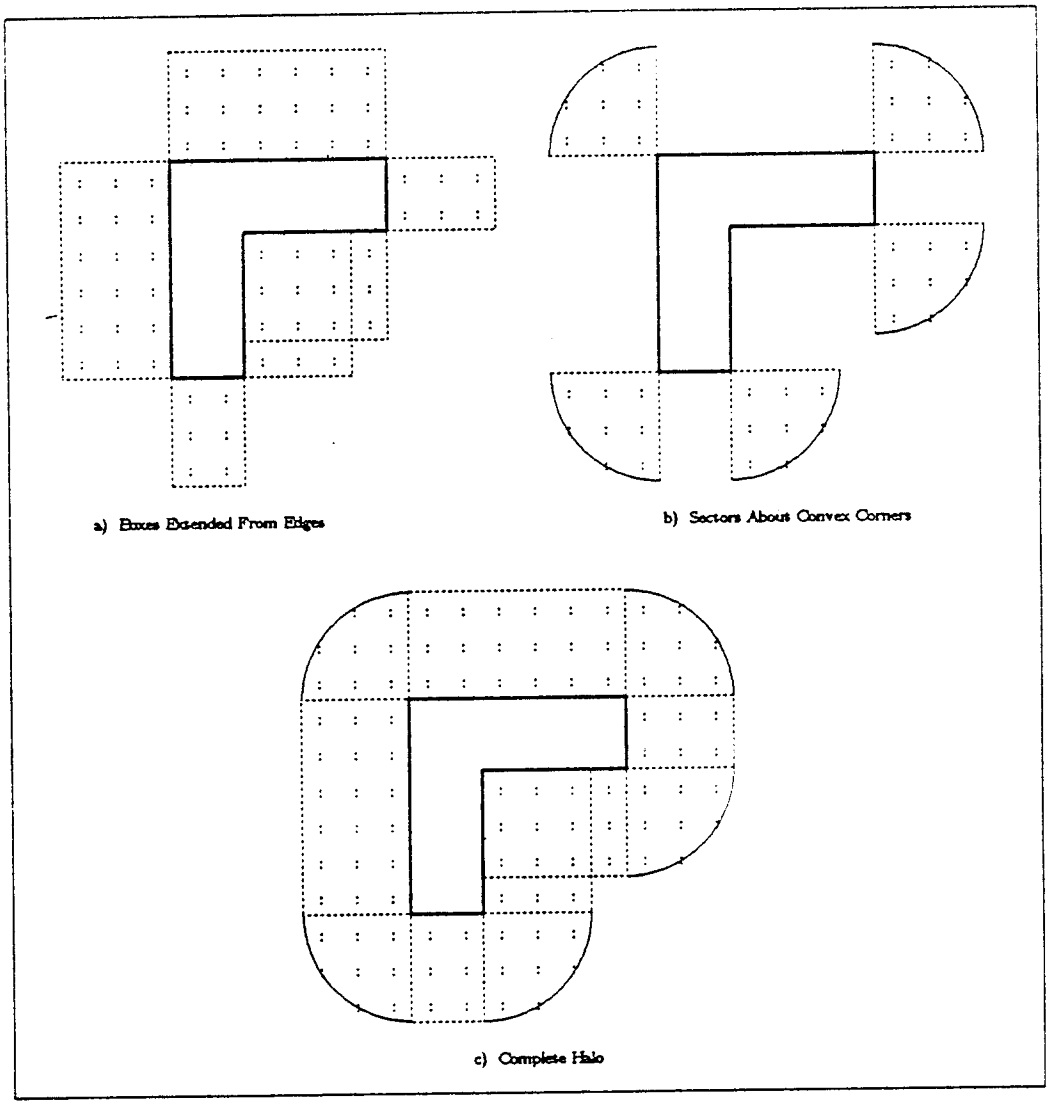


Figure 3.4. - Halo Construction. An outward facing halo can be constructed from two components: boxes extending perpendicularly outward from region edges (a), and circular sectors around convex corners (b). This construction lends itself to the boundary-edge processing employed by the region-operations. Some checks involve only perpendicular tolerances. Such checks can be implemented by leaving the corner sectors out of halos, as in (a) alone.

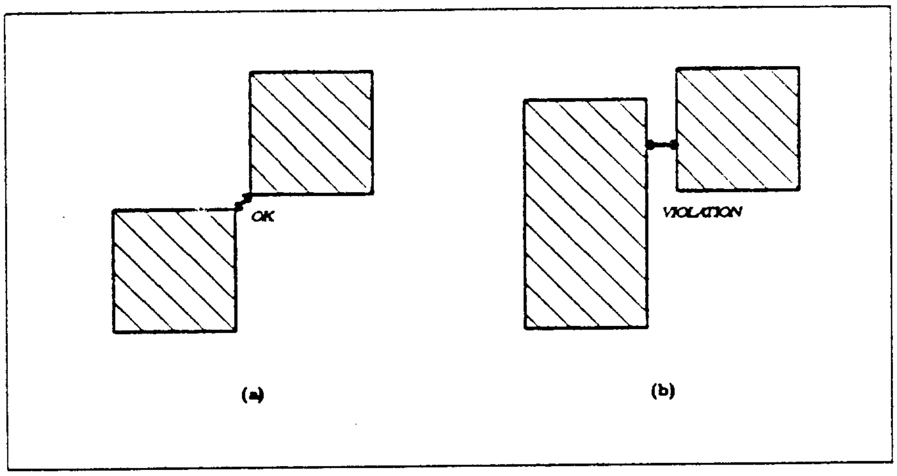


Figure 3.5. - Facing-Edge Rules. Facing-edge rules specify a minimum spacing between facing edges (b), that does not apply diagonally (a). They guard against the formation of narrow slivers of resist during processing, which could break off and float to other parts of a circuit. Such rules are checked with exclusion halos having no corner sectors. These rules are checked by leaving the corners out of exclusion halos.

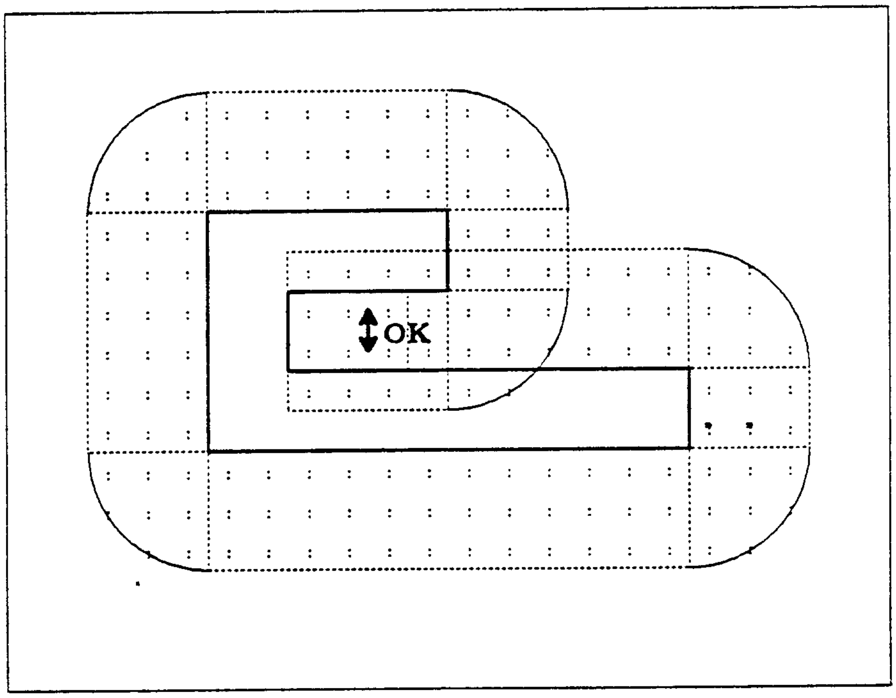


Figure 3.6. - Notches and Single Layer Spacing. This figure shows how a notched region can intrude into its own halo. Single layer spacing rules often permit such notches, being concerned only with internode spacing. Such rules can be implemented by using a variant on the spacing tolerance check that compares the node numbers of intruding edges with the node number of the region to which the halo belongs.

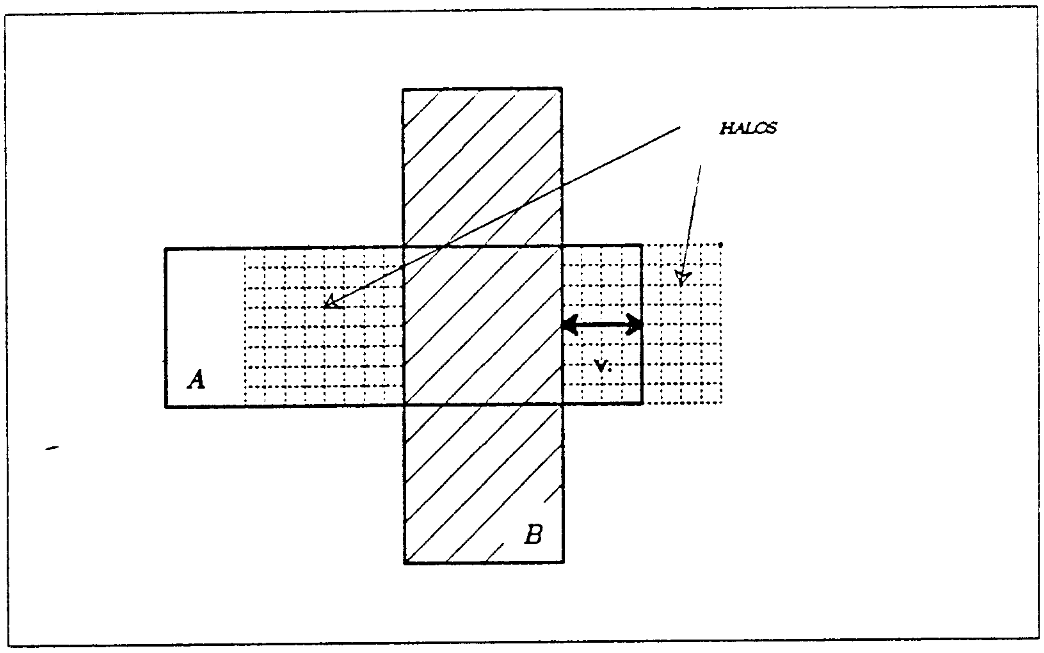


Figure 3.7. - Extension checks. This figure illustrates how the extension of a layer A beyond a layer B is checked. The check is unusual in that halo-boxes extend outward only from those *parts* of edges on the second layer (B), covered by the first layer (A). No corner-sectors are involved.

area or perimeter, electrical characteristics of a node, and even on the function of the intended function of a node. In addition, since filter routines are written in a general purpose programming language, complex and unanticipated conditions are readily handled.

Width and enclosure checks are closely related to spacing. In fact they can be expressed in terms of the **SPACING** operation as follows:

WIDTH[A,n] <=> **SPACING1**[(**NOT** A),n];
ENCLOSURE[A,B,n] <=> **SPACING**[A,(**NOT** B),n];

Here **SPACING1** checks spacings between regions on a single layer, and **SPACING** checks spacings between regions on two distinct layers. The **NOT** operation takes the complement of a layer, i.e. all the regions where the layer is *not* present. Alternately, width and enclosure can be visualized in terms of inward facing halos.

Extension checks are less closely related to the other tolerance checks; see Figure 3.7. Note that extension checks involve a perpendicular tolerance only - diagonal tolerances are not involved. Halos are not drawn around the entire boundary, but only from portions of edges of the second layer that are covered by the first layer.

3.3.2. Boolean Operations

Boolean Operations, typically **AND**, **OR**, and **AND_NOT**, take two layers as input and create a composite layer as output. These operations are illustrated in Figure 3.8. Note that the **AND_NOT** operation also serves as a *not* or complement operation when performed on a single layer.

Boolean operations permit layer combinations to be selected for tolerance checks. For example, the implant/nonimplanted-gate spacing rule illustrated in Figure 2.7 can be checked by the following sequence of operations:

> *Gate* = *Polysilicon* **AND** *Diffusion*;
> *NIGate* = *Gate* **AND_NOT** *Implant*;
> *Violations* = **SPACING**[*NiGate*,*Implant*,2];

The first operation generates a layer, *Gate*, layer consisting of all regions where both polysilicon and diffusion are present. The second operation generates a layer, *NIGate*, consisting of all gate regions where implant is not present. The final operation generates a layer, *Violations*, of all cases where implant is closer than 2 units to a nonimplanted gate region.

Boolean operations are also useful during circuit extraction, for instance to determine contacts and gate regions.

3.3.3. Topological Operations

Topological operations, such as **TOUCHING**, **OVERLAPPING**, and **ENCLOSING**, select regions based on topological relationships; see Figure 3.9. They differ from boolean operations in that they select an *entire* input region when part is involved in the

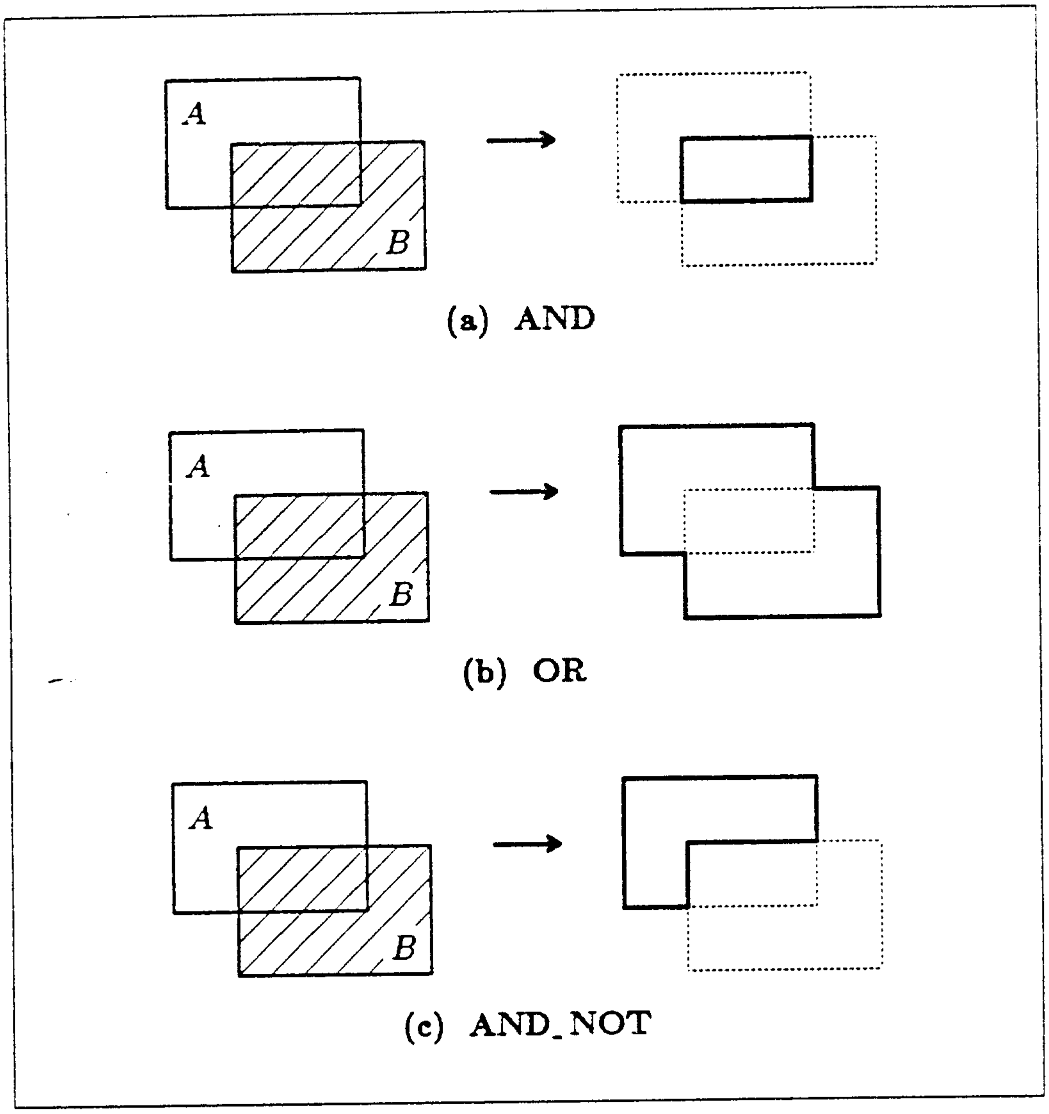


Figure 3.8. - Boolean Operations. Boolean operations define regions in terms of layer combinations. The **AND** operation outputs the regions where *both* input layers are present, **OR** outputs regions where at least *one* of the input layers is present, and **AND_NOT** outputs regions where *only the first* input layer is present. The dotted regions are only for reference, they are not part of the output.

specified interaction, while boolean operations only select the interacting portion.

Topological operations are useful for determining the roles of regions to permit conditional checks. For instance polysilicon/diffusion overlap usually implies a transistor region for which a certain minimum width as well as polysilicon and diffusion extensions are

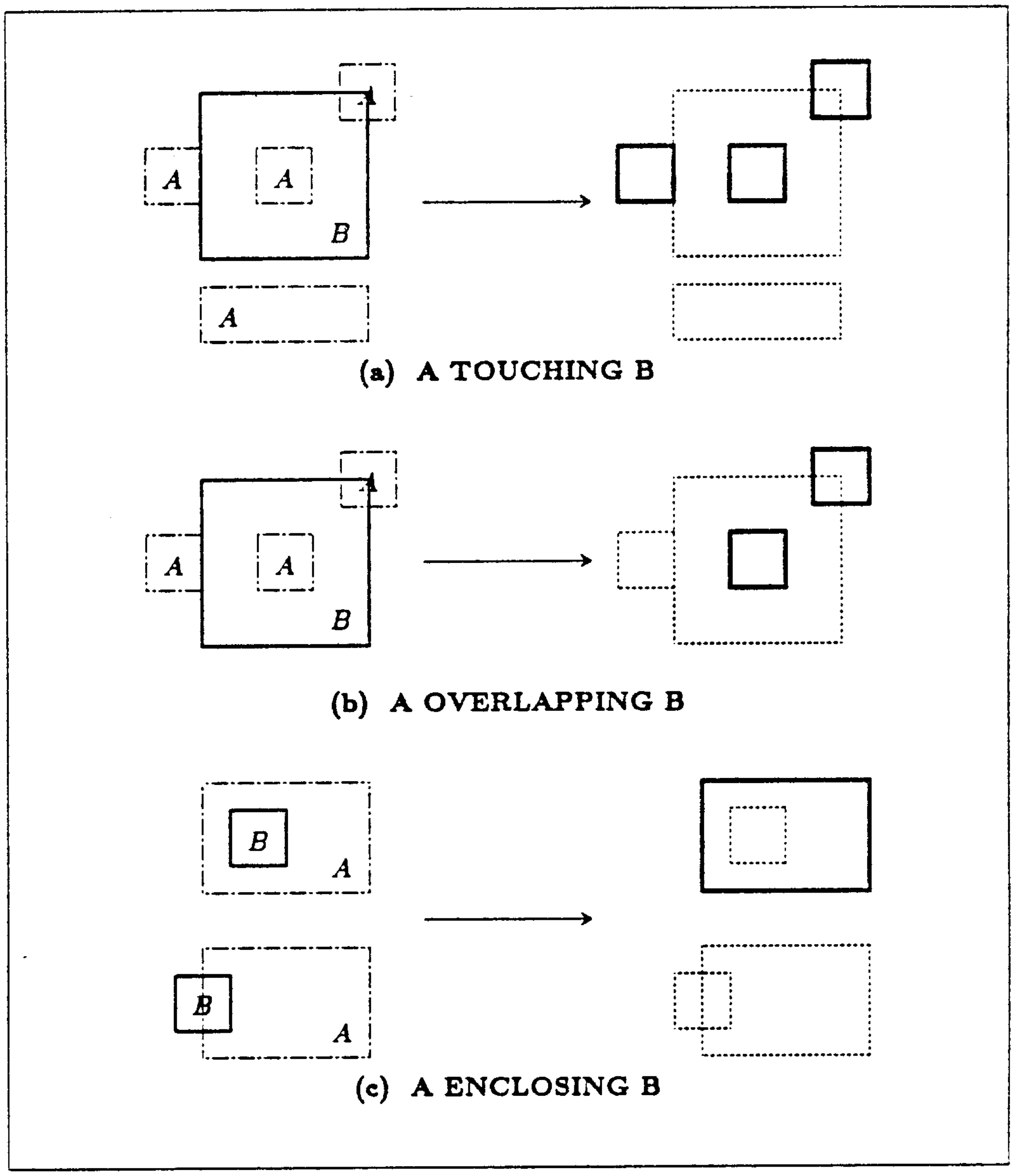


Figure 3.9. - Topological Operations. Topological operations select regions based on their relationship to other regions. They differ from boolean operations in that they select *entire* regions of the input when an interaction with another layer occurs, while boolean operations only select the interacting *parts* of regions. **TOUCHING** selects regions on the first layer that touch or overlap regions on the second layer. **OVERLAPPING** selects regions on the first layer that actually overlap (not just touch) regions on the second layer. **ENCLOSING** selects regions on the first layer that entirely enclose a region on the second layer.

required. However polysilicon and diffusion also overlap in butting contacts where these tolerances are not relevant. See Figure 3.10. These cases can be distinguished between, both for DRC and circuit extraction purposes, by the following sequence of operations:

PD = *Polysilicon* **AND** *Diffusion*
PDBC = *PD* **OVERLAPPING** *Cut*
Gate = *PD* **AND_NOT** *PDBC*

The first operation creates a layer, *PD*, containing all polysilicon/diffusion overlap. The second operation, selects the regions of *PD* involved in contacts, and the third operation selects all the *other* regions in *PD*.

3.3.4. Sizing Operations

Sizing operations are *grows* and *shrinks* on layers; see Figure 3.11. A grow operation generates widened versions of the input regions, and a shrink generates narrower versions. A true **GROW** or **SHRINK** operation involves rounding of region corners, as in the top part of Figure 3.11. Since many mask artwork representations do not include arcs, grows (shrinks) are usually done by moving all edges outward (inward) and extending them so they continue to meet, as in middle part of Figure 3.11. More elaborate polygonal approximations of the true **GROW** and **SHRINK** operations are also possible, e.g., as in the bottom part of Figure 3.11. Sizing operations are used to adjust the width of regions to compensate for peculiarities of a particular process, e.g., a tendency to print lines on certain layers either a bit too narrowly or a bit too widely.

Sizing operations are also used to implement conditional design rules, where a tolerance is dependent on the proximity of another layer. For example, to take the reflection phenomenon into account (see Figure 2.4), the metal width check in Figure 2.9 treats metal edges flanked by polysilicon edges differently. This rule treats metal edges affected by reflection from polysilicon edges as if they were moved outwards 0.5 units. The following sequence of operations implements this rule:

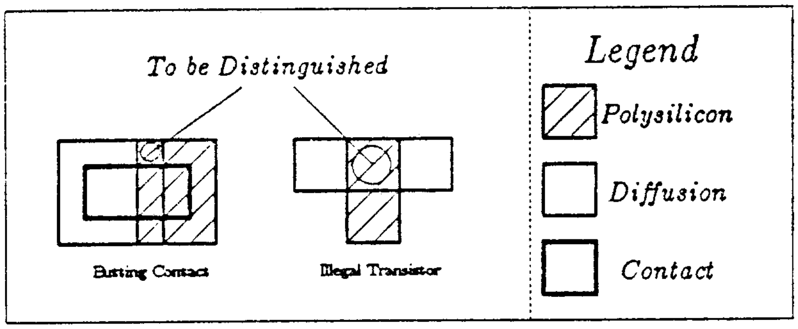


Figure 3.10. - Transistor vs. Butting Contact in Poly/Dif Overlap. Polysilicon/Diffusion overlap occurs in transistors and butting contact (circled regions). Extensions and minimum width are required for such regions in transistors but not when they occur in contacts. These cases can be distinguished between with an OVERLAPPING operation that checks whether each polysilicon/diffusion region overlaps a contact-cut.

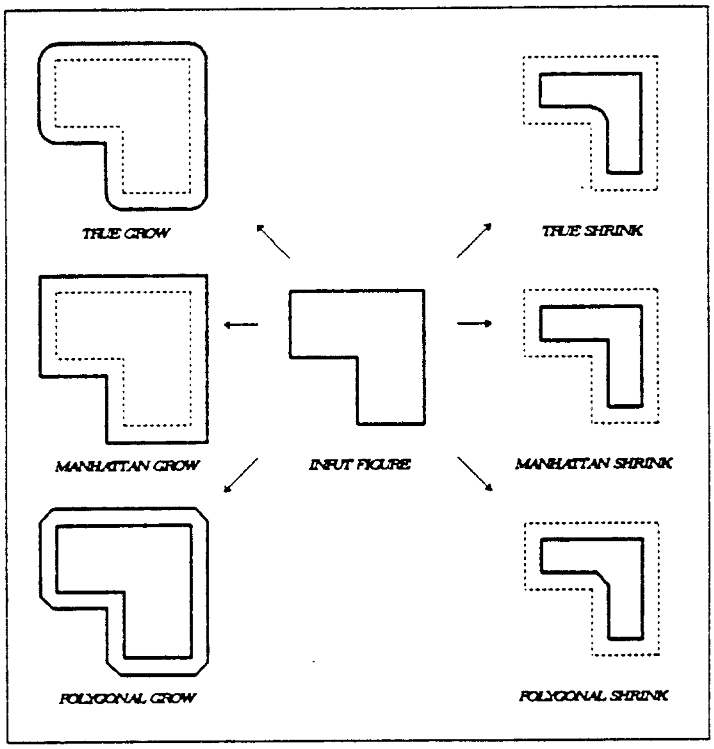


Figure 3.11. - Sizing Operations. A true grow or shrink (by a certain radius) involves rounded corners (top). Since region edges are limited to straight line segments in the region-operation approach, a polygonal approximation must be used. The simplest approximation is obtained by moving all boundary edges out (in) by the radius of the grow (shrink), and then extending or trimming the edges so they just meet again (middle). More elaborate approximations are also possible (bottom) and yield more accurate results.

PGrow = **Grow**(*Polysilicon*,1)
PHalo = *PGrow* **AND_NOT** *Polysilicon*
MShrink = **SHRINK**(*Metal*,.01)
MEdges = *Metal* **AND_NOT** *MShrink*
MRefEdges = *PHalo* **AND** *MEdges*
MAdd = **GROW**(*MRefEdges*,.5)
MetalNew = *MAdd* **OR** *Metal*
Violations = **SPACING1**(*MetalNew*,1)

Figure 3.12 illustrates how this works. The first two operations, a **GROW** followed by an **AND_NOT**, generate the *PHalo* layer identifying areas near (but to the outside of) polysilicon edges. The next two operations, a **SHRINK** and an **AND_NOT**, create the layer *MEdges* consisting of very narrow regions along metal edges. Then the intersection of *PHalo* and *MEdges* is formed, (via **AND**) to create the *MRefEdges*. This layer consists of narrow regions along those portions of Metal edges effected by reflection. Next *MRefEdges* is grown by .5 units and combined with the original metal layer (via **OR**) to create the *MetalNew* layer. *MetalNew* is the original metal layer, with all edges effected by reflection moved out .5 units. The actual spacing check is done on this layer.

3.3.5. Connectivity and Tag Operations

Connectivity operations are used to identify connected regions on a layer with a unique *node* number, and to determine connections between regions on different layers. The output of connectivity operations is originally in the form of a tag-file giving pairs of edges or nodes that are connected. Tag-files can be processed to generate nongeometric data, such as transistor connection lists, or can be used as input to a **TAG** operation that actually tags edges in an edge-file with node numbers or other information.

The primary use of connectivity operations is to determine circuit connection networks during circuit extraction. For details on how this is accomplished see [Szymanski & Van Wyk 1983]. What is important for design rule checking is that operations exist for tagging mask data with node numbers. This permits connectivity-dependent design rules to be handled, via built in primitives such as single-layer internode spacing, or in more complicated or unusual cases via user-written filter routines that discriminate on the basis of the attached

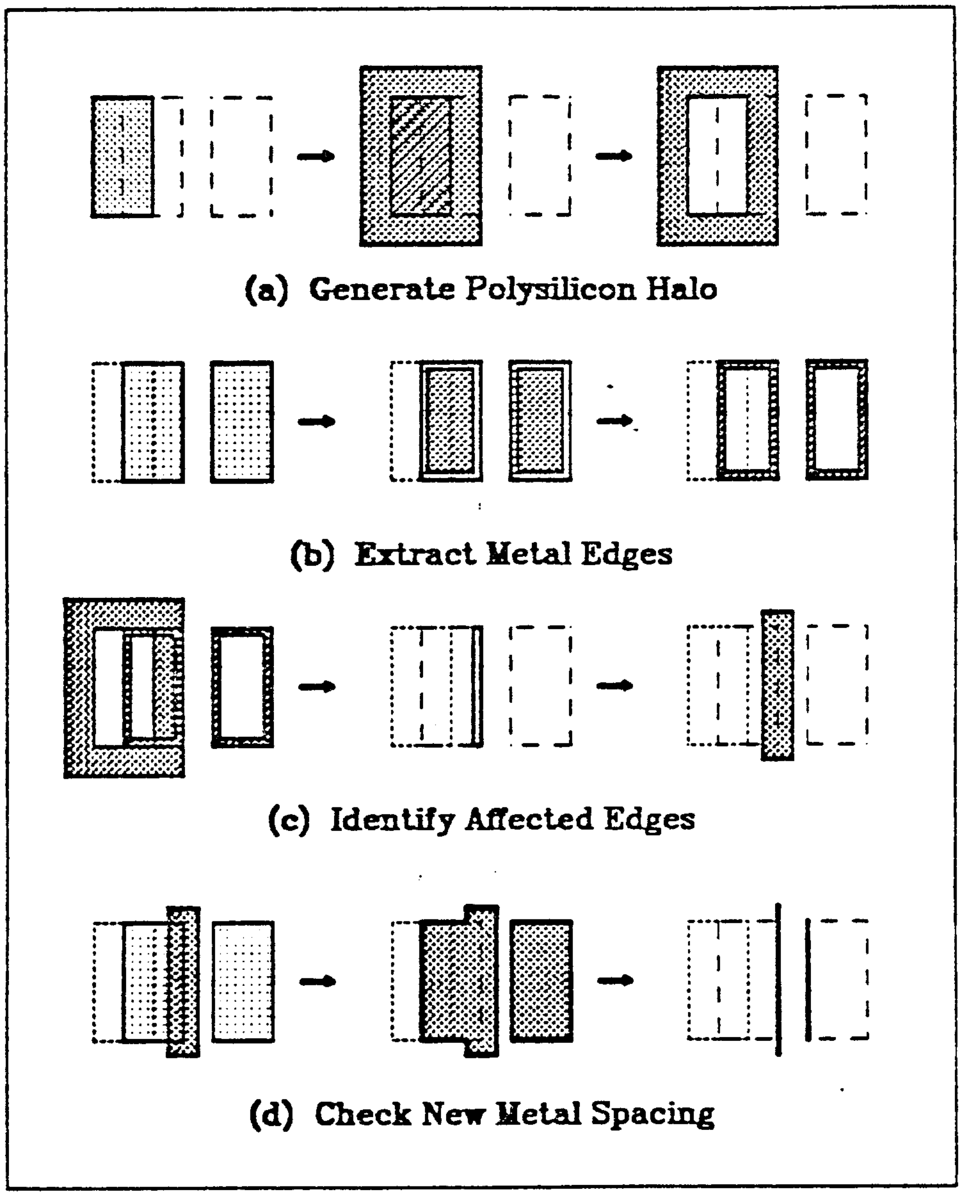

(a) Generate Polysilicon Halo
(b) Extract Metal Edges
(c) Identify Affected Edges
(d) Check New Metal Spacing

Figure 3.12. - Implementation of a Reflection Rule using Grows and Shrinks. A reflection rule requiring greater spacing between metal edges affected by nearby polysilicon can be checked as follows. Use a **GROW** and an **AND_NOT** operation to generate halos around polysilicon regions, (a). Use a **SHRINK** and an **AND_NOT** to mark metal edges with thin slivers, (b). Then **AND** together the polysilicon halos with the metal edge slivers to identify metal edges affected by reflection, and **GROW** out the resulting layer, (c). Finally **AND** together the *grown* metal edges with the original metal layer to obtain a new metal layer that is widened at affected edges, and do a spacing check on the new layer, (d).

connectivity information.

The **TAG** operation can be used to associate arbitrary information with the edges in an edge-file. This allows design rules to be conditional on any information, as long as there is some way to generate a tag-file associating the information with edges or nodes in the mask data. The area and perimeter operations of the next section, for example, generate tag files that can be used by **TAG** to associate area and perimeter information with the mask regions.

3.3.6. Area, Length and Perimeter Checks

Region areas and perimeters, and edge lengths, are important for calculating electrical properties of circuits such as capacitance and transistor sizes. In addition design rules occasionally depend on these parameters, e.g., the length-dependent spacing rule illustrated in Figure 2.10. **AREA**, **LENGTH** and **PERIMETER** operations generally generate output in the form of tag files for regions or edges giving the numerical values of their area, length or perimeter (respectively). These tag-files consist of region-number/parameter-value pairs. If desired, a second pass through the layer can be made to incorporate the tag data directly into the edge-file.

Area, length, and perimeter tags can be accessed by user-supplied selection routines to implement conditional tolerance checks depending on these parameters. In addition, variants of **AREA**, **LENGTH**, and **PERIMETER** operations that output, those regions where the relevant parameter falls in a specified range, are usually provided for design rule checking; see Figure 3.13. For example such an **AREA** operation can be used to check contact rules requiring contacts of fixed area.

3.4. Scanline Implementation of Region Operations

Recall that region-operations process region boundary-edges, not the abutting and overlapping figures of the input mask artwork description. One edge-file is kept for each layer in the input, and for each intermediate layer generated by the region-operations. Edge-files

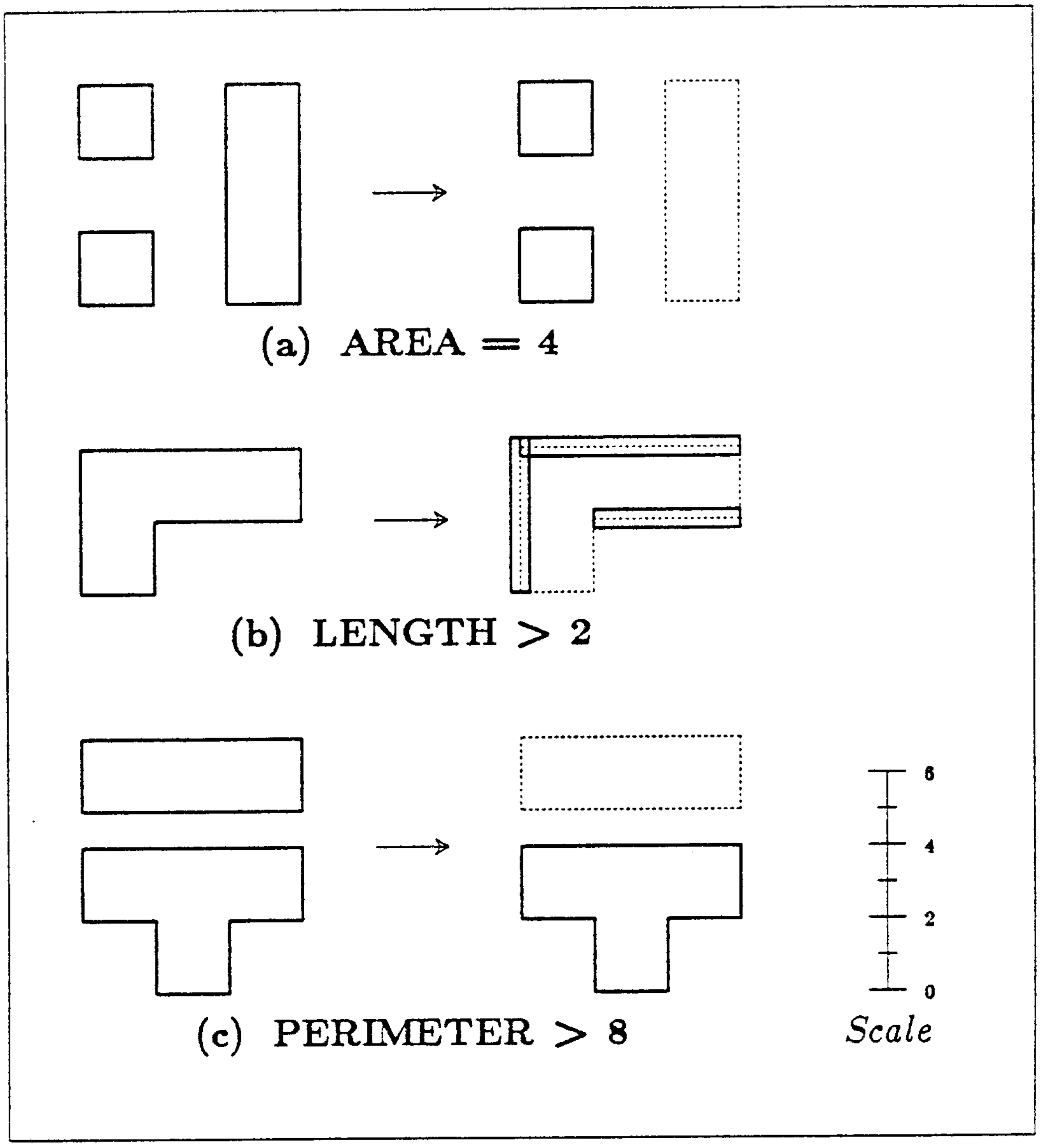


Figure 3.13 - Area, Length, and Perimeter Operations. Area, length and perimeter operations can be used to tag edges with these parameters, for use in circuit extraction or design rule checking. Variants of these operations (shown above) allow selection of regions (or edges in the case of **LENGTH**) where these parameters fall within a specified range.

are too large to comfortably fit into main memory: a single edge-file will typically contain several hundred thousand edges.

Traditionally, edge-files are processed using scanline techniques [Szymanski & Van Wyk 1983]. Scanline processing allows a sequential pass through the input edge-files; it does not require more than a fraction of the edges to be kept in main memory at any one time; and it permits ready access to local geometric context, such as the layers present at a point, and the edges that are nearby.

Scanline processing proceeds horizontal *scanline* by horizontal scanline, left to right, and bottom to top, as shown in Figure 3.14(a). Prior to scanline processing, the input edge-files are sorted into *scan order,* the order in which they will be encountered during the scanline processing. During the scanline processing an *active-list* of edges crossing the current scanline is maintained in main memory; see Figure 3.14(b). Note that edges are of two types, *beginning* and *ending.* Beginning edges lie to the left of regions, (precede them in scan order), and ending edges lie to the right of regions, (succeed them in scan order). Horizontal edges are not explicitly represented. Their presence is readily deduced during processing. As a scanline is processed from left to right, new edges beginning on the current scanline are merged into the active list, and edges ending at the current scanline are deleted from the active list. The maintenance of the active list together with a running nesting count for each layer, i.e. beginning edges less ending edges encountered in the current scanline, provides complete information about the layers and boundaries present at the current point in the scan. This allows a wide range of processing to be integrated into the scan algorithm.

Consider for example, the single-layer **OR** operation. This operation is usually performed after initial conversion of figure data to edge form. The initial edge file is not really in boundary-edge form, as it includes all (nonhorizontal) edges from the original figures, including interior edges of abutting or overlapping figures. The single-layer **OR** removes such non-boundary edges, creating a true boundary-edge file for use with tolerance and other region-operations; see Figure 3.15. The single-layer **OR** can be implemented by marking edges for output whenever a transition from or to 0 occurs in the nesting count. If the 0 transition occurs only below or above the current scanline, the edge is split. When the tops of marked edges are reached they are written to the output. Since the output need not be

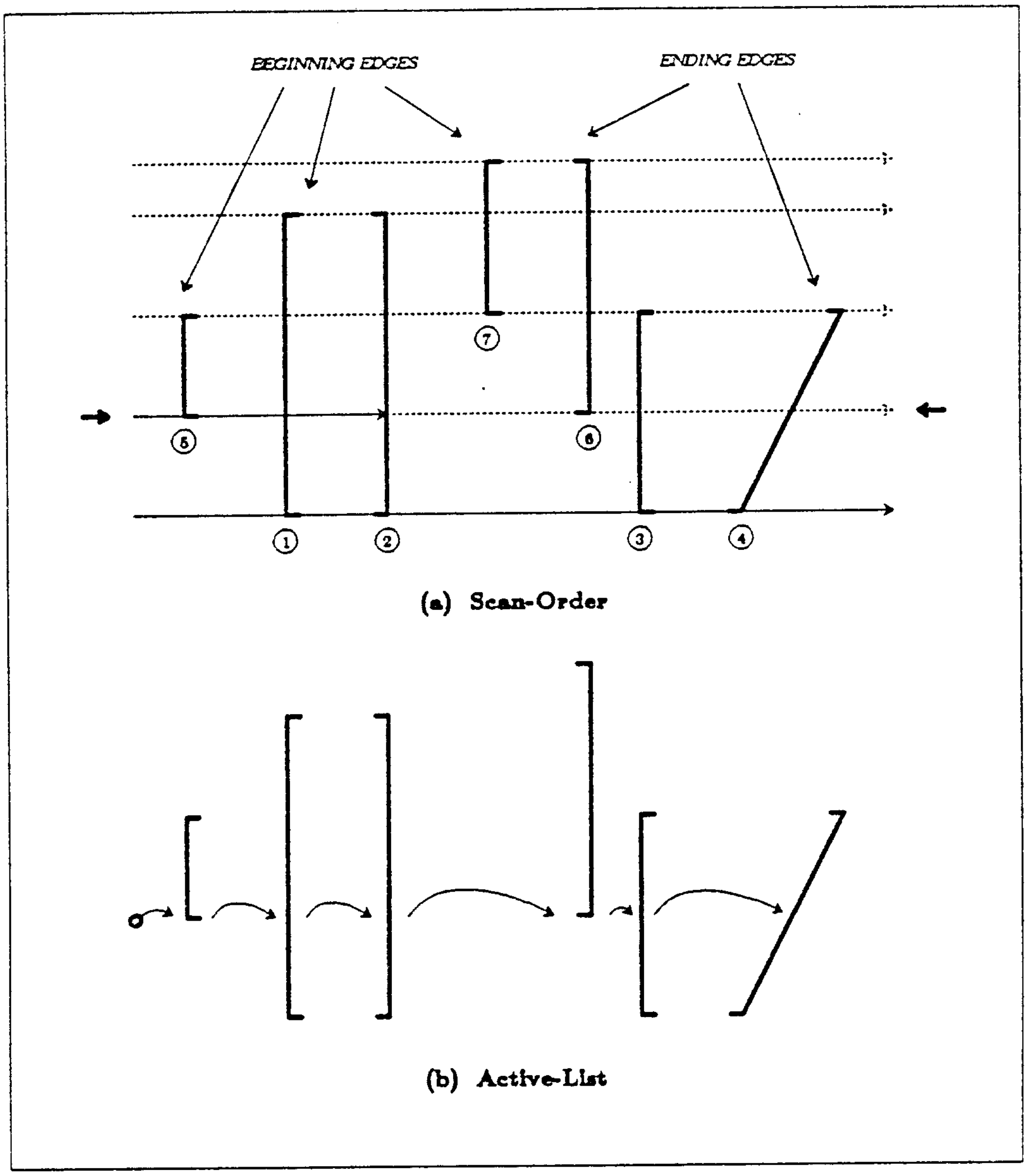


Figure 3.14. - Scanline processing. In scanline processing, regions are bracketed by *beginning* and *ending* edges, that are processed in left to right scans, beginning at the bottom of the design and working up (a). The numbers indicate the order in which the edges will be first encountered during scanning. Edges are sorted into this *scan order* prior to processing. An *active-list* of edges crossing the current scanline is maintained in main memory during processing.

strictly in scan order it will have to be sorted prior to further scanline processing. The other boolean operations are implemented in a similar manner.

Tolerance checks are implemented with an augmented active-list that allows ready access to all data in proximity to the current point. A halo region is computed for each edge as it is encountered and intrusion into the halo is checked with the help of the augmented active-list. In addition actual overlap between the layers is checked. This is analogous to the boolean **AND** operation.

Most other region-operations have straightforward scanline implementations. Sometimes a second pass through the data is needed to tag or select regions based on the computation of the first pass. For example topological and connectivity operations involve

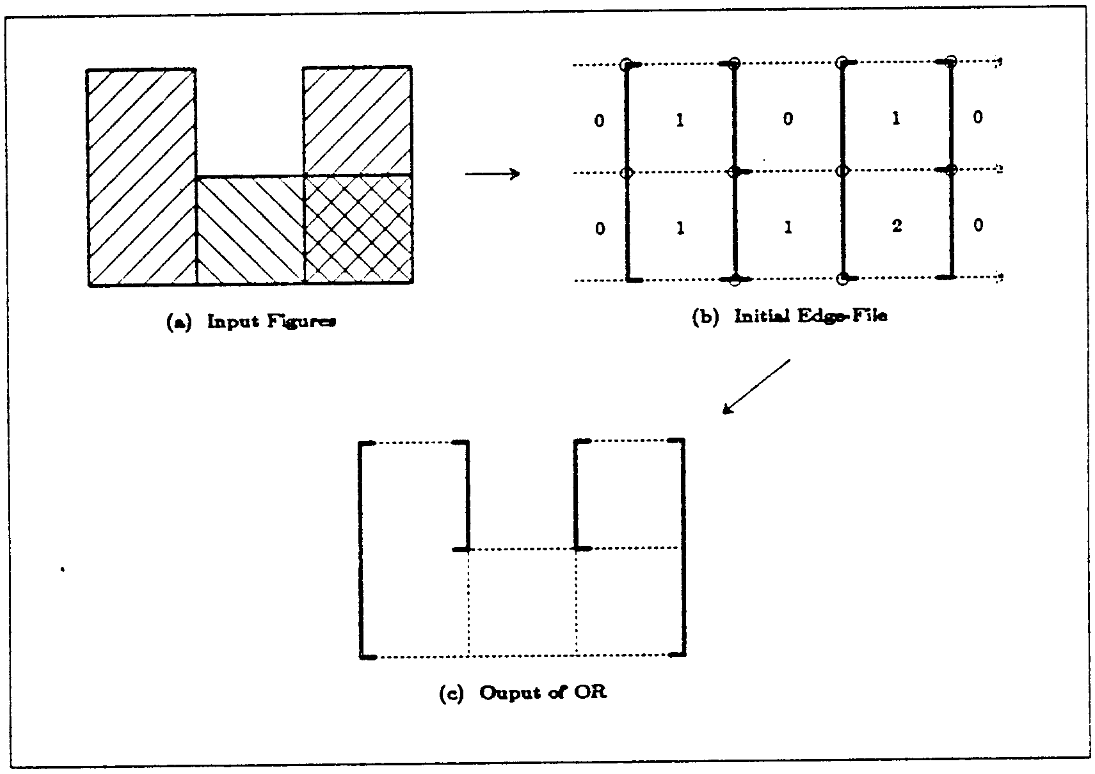


Figure 3.15. Single-layer **OR** implementation. A single-layer **OR** eliminates nonboundary edges (b) resulting from overlapping or abutting figures (a), by outputting only those portions of edges where the nesting count (numbers in b) undergoes a transition to or from 0 (c).

the determination of junctures between regions. This information is readily computed in one scan, but a second scan is needed to to output selected regions in topological operations, and to tag regions with node numbers in connectivity operations. Similarly, area and perimeter computations can be performed in one pass through the data provided edges are tagged with region numbers. A second pass is required to output selected regions or to tag the data with the computed values. Sizing operations require a single pass. They generate a new, shifted, edge for each edge in the input. Since scanline processing permits access to adjacent edges at edge end-points, the amount that the edges need to be extended to meet properly is readily determined.

An important feature of scanline algorithms is that they permit sequential processing of the input after the initial sort, and only the edges crossing the current scanline need to be stored in main memory at any given time. Since VLSI designs have relatively uniform edge density, and are roughly square, the number of edges on the active-list and hence the main memory requirement is $O(\sqrt{n})$, where n is the total number of edges in the design. The processing time, excluding the initial sort, is $O(n)$.

Note that the processing time is very sensitive to the average number of scanlines per unit-y: if the number of scanlines doubles, the processing time doubles. This is because an edge must be handled once for *every* scanline it crosses. The number of scanlines per unit-y is just the number of distinct y-coordinates of end-points per unit-y. Thus scanline processing is sensitive to the "utilized" resolution of the design.

3.5. Pros and Cons of the Region-Operation Approach

The region-operation approach to design rule checking has a number of strong points. The bag of tools approach, providing a set of operations that can be pieced together to provide the required functionality, is extremely flexible. The system can be retargeted to new design rules by simply piecing together the appropriate operations. If functions are required which are not supported by existing operations, new operations can be added without impacting any existing components of the system.

The region-operation approach neatly integrates design rule checking with the closely related functions of circuit extraction and compensation. This saves redundant software and user interfaces, and gives the DRC complete access to extraction functions as needed.

The tolerance, compensation, and boolean operations match the way designers think about design rules. They provide a natural language for expressing design rule checks.

The use of scanline algorithms allows data files (which are often very large) to be read and processed sequentially, with relatively small main memory requirements ($O(\sqrt{n})$).

The decomposition of design rule checks into sequences of simple operations, each with its own input and output, intrinsically provides frequent check points. Since design rule checks of large designs involve many hours of processing, this is a very useful feature.

However the region-operation approach does have shortcomings. A full design rule check involves many region-operations with the consequent generation of many intermediate layers. Typically the edge files are too large to be kept in main memory and hence are kept on disk resulting in great amounts of disk I/O with relatively little CPU processing. DRC runs tend to be disk-bound and slow.

Because most region-operations apply uniformly in all directions rules involving directional context are difficult to check. Such rules are commonly involved in the specification of transistor or contact form. For example, transistors must have polysilicon and diffusion extensions; see Figure 3.16. The size of extensions, if present are readily checked via an extension tolerance operation. However checking for the presence of extensions is tricky. The regions directly opposite gate edges must be checked for the presence of polysilicon or diffusion extensions. However the presence of polysilicon extensions outside two adjacent gate edges signals a bent transistor; see Figure 3.16(b). In this case an additional check must be made to ensure that the polysilicon encloses the corner. Thus the tolerances that need to be checked depend on *which* layers abut adjacent edges at gate corners. Such a rule is extremely

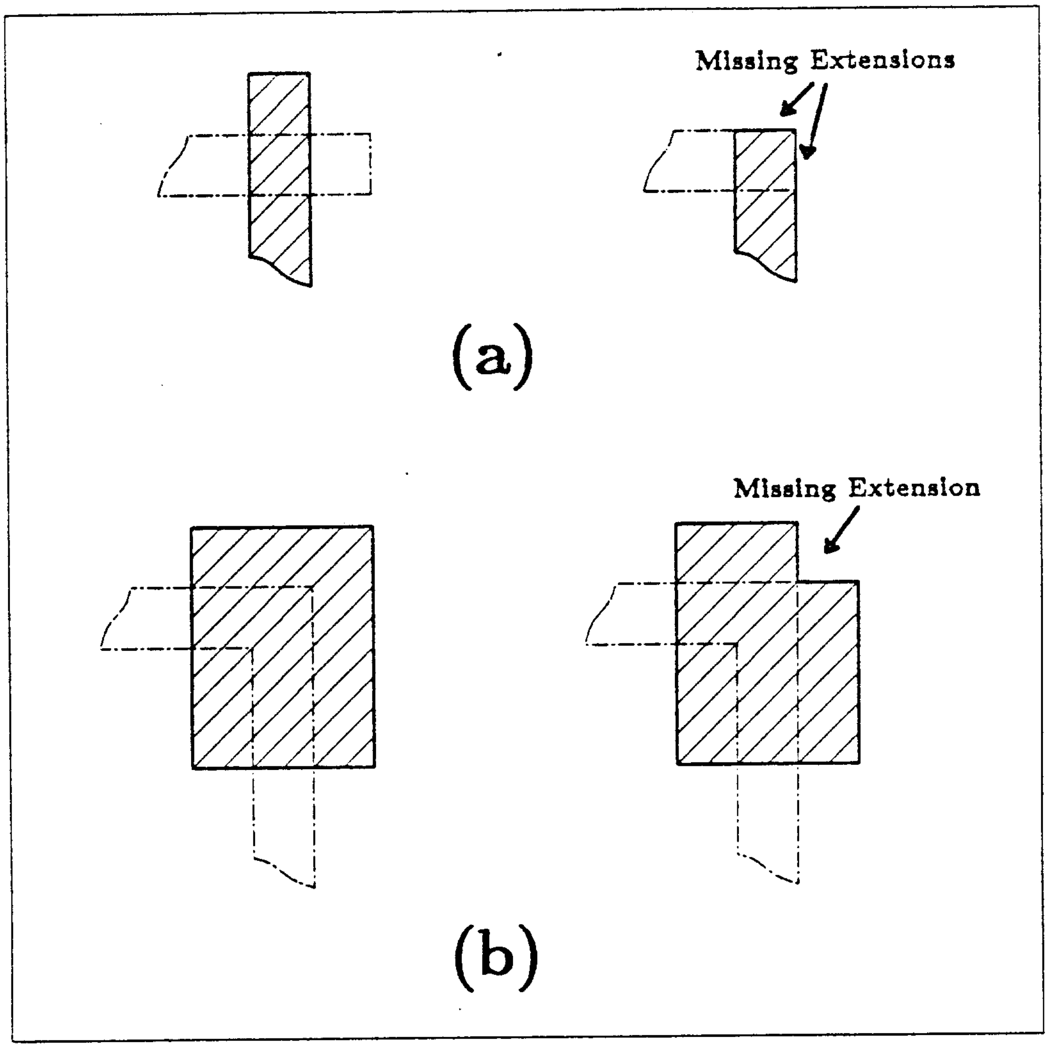


Figure 3.16 - Transistor Rule Involving Directional Context. In MOS processes polysilicon and diffusion must extend laterally beyond transistor gate regions (a). In addition extensions must be present around corners of bent transistors (b). To check this rule the layers present outside each of the two edges at a gate corner must be compared. Such directional context is difficult to generate in the region-operation approach: it requires long sequences of operations.

difficult to check through a sequence of region-operations.

3.6. Summary

The region-operation approach to design rule checking is used almost universally. This approach includes DRC functions as part of an integrated mask artwork system which is also used to do circuit extraction, and compensation. All of these functions are implemented by sequences of region-operations. Region operations take one or two input files and generate an output file.

Mask data is converted to sorted edge-files, one per layer, and region-operations are generally implemented using scanline algorithms. This allows for sequential access of input files, and approximately $O(\sqrt{n})$ main memory requirement where n is the total number of edges in the input.

The region-operation approach provides a flexible, natural, and powerful system for formulating design rule checks. It also has the advantage of neatly integrating DRC with related functions, thus avoiding redundant code and user interfaces, and allowing the DRC full access to extraction functions when needed.

On the negative side, a full DRC requires the sequential application of many primitive operations and the generation of many intermediate layers. This requires a great amount of I/O with relatively little CPU processing, leading to slow I/O bound processing. In addition, anisotropic rules, involving directional context, are clumsy to check, requiring long sequences of operations.

3.7. References

For an example of a mask artwork format, see the definition of CIF in [Mead & Conway 1980]. For a discussion of connectivity operations, and scanline implementation of region-operations see [Szymanski & Van Wyk 1983]. A general theoretical discussion of scanline algorithms, is given in [Bentley 1979]. References to region-operation systems are at the end of the next chapter.

[Bentley 1979]

J.L. Bentley, "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computers*, Vol. C-28, No. 9, September, 1979, pp. 643-646.

[Mead & Conway 1980]

C. Mead, L. Conway, *Introducetion to VLSI Systems*, Addison-Wesley, Reading, 1980, pp. 115-127.

[Szymanski & Van Wyk 1983]

T.G. Szymanski, and C.J. Van Wyk, "Space Efficient Algorithms for VLSI Artwork Analysis," *Proc. 20th Design Automation Conference*, June, 1983, pp. 734-739.

THIS SPACE INTENTIONALLY LEFT BLANK

CHAPTER 4

Survey of Non-Corner-Based DRC Systems

4.1. Introduction

There are three types of design rule checking systems in existence today: region-operation, pixel-based, and corner-based. The vast majority of systems use variants of the region-operation approach presented in the last chapter. They perform design rule checks through sequences of region operations that process boundary-edge data.

A few systems use a pixel-based approach: these systems process two-dimensional pixel-arrays rather than edges. Each pixel position is marked with the mask layers present there, and in some cases with additional state information during processing. Pixel-based design rule checking is of interest largely because it is amenable to highly parallel hardware implementation.

Corner-based systems are the topic of this thesis. They apply conditions at each location in a design according to the pattern of layers present at that location. These systems differ from region-operation systems in that there is no sequencing of operations: all rules are applied concurrently.

This chapter surveys region-operation and pixel-based systems. Corner-based systems, (and the related Magic, and Intel DRC's) will be considered in Chapter 7, after the presentation of the corner-based approach.

Performance data for some of the systems presented in this chapter is given at the beginning of Appendix I.

4.2. Region-Operation Systems

The vast majority of DRC systems employ variations of the region-operation approach. This section examines a number of such systems. All of these express design rule checks in terms of sequences of the same basic region-operations, and they all use edge-based data representations and processing.

The systems vary from each other mainly in the functionality they provide and the internal data organization employed. Functionality ranges from very high for the commercial DRC vendors such as NCA Corporation, Phoenix Data Systems (PDS), and ECAD Corporation to minimal as in university software to support Mead-Conway-style design activity. The commercial packages handle arbitrary angles in the input data, and allow a full complement of conditional checks, with full access to extracted network information and an interface to user-supplied selection routines, as described in the previous chapter. University software is generally restricted to orthogonal mask data (for speed and simplicity) and does not implement conditional checks, since these are not needed for Mead-Conway designs.

A variety of data organizations are used in these systems, including scanline, two dimensional bins of edges, and swaths of sorted trapezoids, (see Figure 4.1). Each of these seeks to organize the data in a way that allows edge/edge and edge/quadrilateral intersections to be computed quickly and systematically, since these are the basic data operations required to implement the region operations. The choice of data organization impacts the speed of the operations. Scanline processing, presented in the last chapter, is conceptually simple and elegant and is used by most systems. However, a typical edge is handled many times during scanline processing: once for each scanline crossing the edge. This multiple handling of data slows down processing. The fastest systems today use alternative data structures: such as two dimensional bins (PDS), or the more exotic swaths of sorted trapezoids (ECAD). The latter is particularly effective in minimizing the number of times each edge is dealt with.

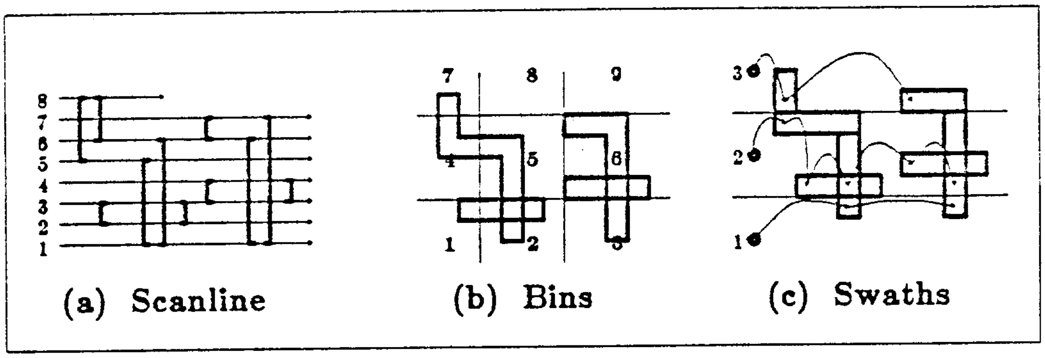


Figure 4.1. - Internal Data Representations. The internal data representations used by region-operation systems vary. Traditionally the scanline organization presented in the last chapter is used (a). Some recent systems (e.g. PDS) organize edges or polygons into square bins (b). At least one system (ECAD) organizes data into sorted horizontal swaths of quadrilaterals (c). All these organizations facilitate processing by allowing quick and systematic computation of edge/edge and edge/quadrilateral intersections.

The choice of data structure also determines how much of a design needs to be maintained in main memory at a time. The scanline and swath methods require a thin slice of the design to be in main memory, thus the main memory required is proportional to $\sqrt{n}$ where n is the size of the design. If bins are used, only the current bin (and possibly the 8 neighboring bins) need be in memory at one time, so the amount of main memory required remains constant regardless of the size of the design.

Note that the data organization used impacts the speed and order of the computation, but not its character. Regardless of the organization employed the same edge-based processing occurs, thus the user model remains the same.

The following subsections examine systems one by one. Special attention is given to the main dimensions of variability outlined above: functionality and internal data organization.

4.2.1. Baird's System

In his masters thesis at Rutgers, on artwork verification for integrated circuits [Baird 1976], Henry Baird surveyed a number of early artwork systems and observed surprising uniformity: the same primitive operations came up again and again. He proposed an integrated artwork system including all the types of operations presented in the last chapter. Baird noted that to avoid spurious errors, operations must be region-based, not figure-based as in some of the early systems, and he showed how region-based operations could be implemented using scan-order edge-based processing of region boundaries. He allowed for circular arc edges as well as straight edges, to permit true grow operations (see Figure 3.11).

Though his proposed system employed scan-order processing, the data structures and algorithms were much more complicated than the current scanline algorithms. Internally, data was maintained in fully intersected form, i.e. edges were split at intersection points and intersections were represented explicitly as vertices. Data structures were kept both for edges and vertices and these structures were cross referenced. Processing actually proceeded vertex by vertex.

Baird's thesis was the first thorough exposition of region-operation based mask artwork processing. He documented his algorithms with pseudo-code and careful analysis. Later he implemented many of his ideas in a DRC system at RCA.

4.2.2. Lauther's Algorithm

Ulrich Lauther of Siemens Corporation, presented a paper at the 18th Design Automation Conference [Lauther 1981] showing how a scanline algorithm proposed by Bentley could be modified to perform boolean operations efficiently. Lauther's algorithm uses a true scanline approach, as described in the previous chapter. It has $O(n \log n)$ time, and $O(\sqrt{n})$ main memory requirements. Edges are restricted to straight lines.

The Siemens DRC employs this algorithm to implement region-operations.

4.2.3. Haken's System

Dorothea Haken developed a DRC program at Carnegie-Mellon University [Haken 1980] for orthogonal mask data. This program was written to support the Mead-Conway design activity at CMU. The system includes boolean operations, topological operations (which were required to handle butting contacts in the Mead-Conway nMOS process) and a basic tolerance-check primitive for checking widths, spacings, extensions and enclosures. Extraction functions were not supported, and node information was not maintained, thus spurious same-node spacing violations could not be avoided. A simple filter was written to eliminate most of these spurious violations.

Tolerance checks were implemented by building exclusion rectangles for each edge, and then checking for intersections between mask features and the exclusion rectangles, i.e. tolerance checks were implemented via manhattan halos (see Section 3.1 of Chapter 3). All primitives were implemented using scanline algorithms.

4.2.4. Hitachi

T. Kazowa described the Hitachi artwork system in a paper presented at the 18th Design Automation Conference [Kozawa 1981]. The Hitachi system uses exactly the region-operation approach as described in the previous chapter. All the operation types are supported. Implementation is via scanline algorithms. For tolerance checks, the Hitachi system uses a modified scanline algorithm that maintains a list of edges within a thin swath below the current scanline.

4.2.5. NCA

NCA [Alexander 1978, 1983] has been the major vendor of DRC services for many years. Their primary aim has been to capture all the rules employed by the industry. Processing speed has been secondary.

The NCA system uses standard region-operations and mainly scanline processing. Tolerance checks use a modified scanline algorithm that maintains edges within a swath of fixed width below the current scanline. A few operations, such as **GROW**, actually operate on a figure-based data representation. User written selection routines are used to implement complex conditional checks.

4.2.6. Phoenix Data Systems

Phoenix Data Systems (PDS) [Spink 1983] has recently emerged as a major competitor with NCA for mask artwork verification and preparation services. The PDS system employs a preprocessing step that cross-indexes connected mask data for efficient extraction and simulation. The DRC again uses the region-operation approach. Output of the tolerance check operation can be all violating edges, or just the portions of edges which are in violation. This gives flexibility in building up conditional checks. Edges are sorted into two-dimensional bins and processed bin-by-bin rather than in scan order.

4.2.7. ECAD

ECAD [Huang 84] has recently entered the DRC vendor market (their first product was announced in 1983). Their system uses region-operations, providing about the same functionality as the PDS system. ECAD uses sorted swaths of trapezoids (see Figure 4.1c.) as their basic data structure. This data structure allows very fast implementations of the primitive operations.

4.2.8. Other Systems

There are many other systems employing region-operations on boundary-edge mask data, with variations on functionality, special restrictions on input data, and varied internal data organizations. But the examples given above illustrate the type and range of variation in these systems.

4.3. Pixel Systems

A few DRC systems represent mask data by a pixel-array, rather than by boundary edges. Each pixel is marked with the layers present in it. The algorithms employed by these systems have a decidedly different flavor from those employed in the region-operation approach, since they deal with pixels rather than region edges.

Pixels are square, and mask data is usually required to be orthogonal, although tricks have been developed for handling 45s and other angles as well (given enough resolution). The computation in these systems is organized so that the processing of each pixel depends only on its own state and that of a small number of nearby pixels. The main appeal of the pixel-based approach is that, since the pixels can be processed independently, highly parallel hardware implementations are possible.

The method for expressing design rules varies. In some systems primitive operations are implemented that can be pieced together to implement traditional region-operations (though in terms of pixels not edges). In others each rule is expressed in terms of a finite state machine. In one system the Mead-Conway design rules are hard-coded: they can not be modified without rewriting the system.

The main problem with the pixel approach is that the required resolution can result in a very large number of (tiny) pixels in a design; see Figure 4.2. The size of pixels is determined not by feature sizes, but by the minimum amount by which region shapes and positions can be varied, i.e. the resolution used in specifying the design. If the resolution is doubled, the number of pixels increases by a factor of 4. As the resolution increases with respect to average feature size the number of pixels quickly becomes much greater than the corresponding number of edges in an edge-based representation. The large amount of data to be processed puts stringent requirements on the efficiency of reading, processing, and writing the individual pixels if the overall processing is to be faster than or even competitive with the region-operation approach. Recall that the amount of processing per individual data item is already small in the region-operation approach, so design rule checking will not be speeded up significantly by reducing this time, unless corresponding reductions are made in the time required for I/O.

Pixel-based systems are largely experimental. Though a number of interesting systems have been proposed, only one, Baker's, has been fully implemented.

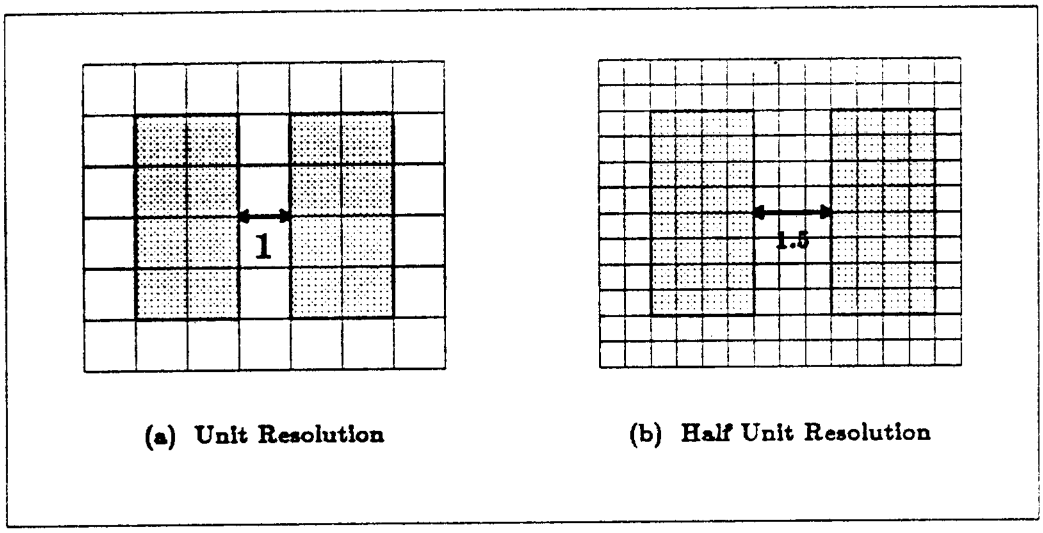


Figure 4.2. - Exponential Growth of Pixel-Array as Resolution Increases. Increasing the resolution of a design from full units (a) to half-units (b), quadruples the number of pixels required to represent a design. Because of this exponential growth, as resolution increases, the amount of data required to represent a design in a pixel-based system quickly becomes much greater than that required by a region-operation system.

4.3.1. Baker's System

Clark Baker developed a pixel-based DRC [Baker 1980] as part of his Master's thesis at MIT. For several years, this DRC was the principal one used in the university VLSI community to check Mead-Conway-style designs. It also may have been the first pixel-based DRC.

Baker's DRC is based on pattern matching on small windows into the design, see Figure 4.3. Pixels are processed in raster order, with 3 lines (plus 4 additional pixels) buffered for a 4x4 window check. Conceptually, the window is moved systematically across the design, and at each window position, a check is made to see if the pattern under the window is legal. Illegal patterns are reported as design rule violations. Baker's DRC employed 2x2, 3x3 and 4x4 window checks. A typical 4x4 check might be done by first checking the center 4 cells, and if they satisfy some criterion, then using the contents of the peripheral cells to generate an index into a table that specifies whether the pattern is acceptable. Pattern checks are hard coded and ad hoc. Consequently they can not be readily extended to more complex rules, or greater resolution.

Baker's program checks all of the Mead-Conway rules except for the rules involving implants. A postfilter eliminates many spurious violations involving same-node spacing.

4.3.2. Seiler's System

Larry Seiler, also at MIT, is currently working on a pixel-based DRC with hardware assist [Seiler 1982] that greatly extends Baker's concept. Each operation in Seiler's system outputs a mask in pixel form, allowing sequencing of operations. This is done by identifying a key cell position within a window. If the window does not pattern match, a 0 is output for the key position, and if it does match a 1 is output. As the window is moved over the design an entire new layer is generated. A set of hard-wired patterns implement the following operations:

i. **WIDTH-2** - check for width of at least 2.

ii. **WIDTH-3** - check for width of at least 3.

iii. **SHRINK-2** - shrink by 2.

The hardware also supports boolean operations. Larger width checks can be implemented by a sequence of width checks alternated with shrinks. (A width check must be done after each shrink, so that a too narrow region does not disappear entirely before a width violation is detected.) Spacing checks can be implemented as width checks on the complement of a layer. Grows can be implemented as shrinks on the complement. Thus tolerance checks and boolean operations, the main DRC primitives of traditional region-operation systems, can be built up from the more basic operations of Seiler's system.

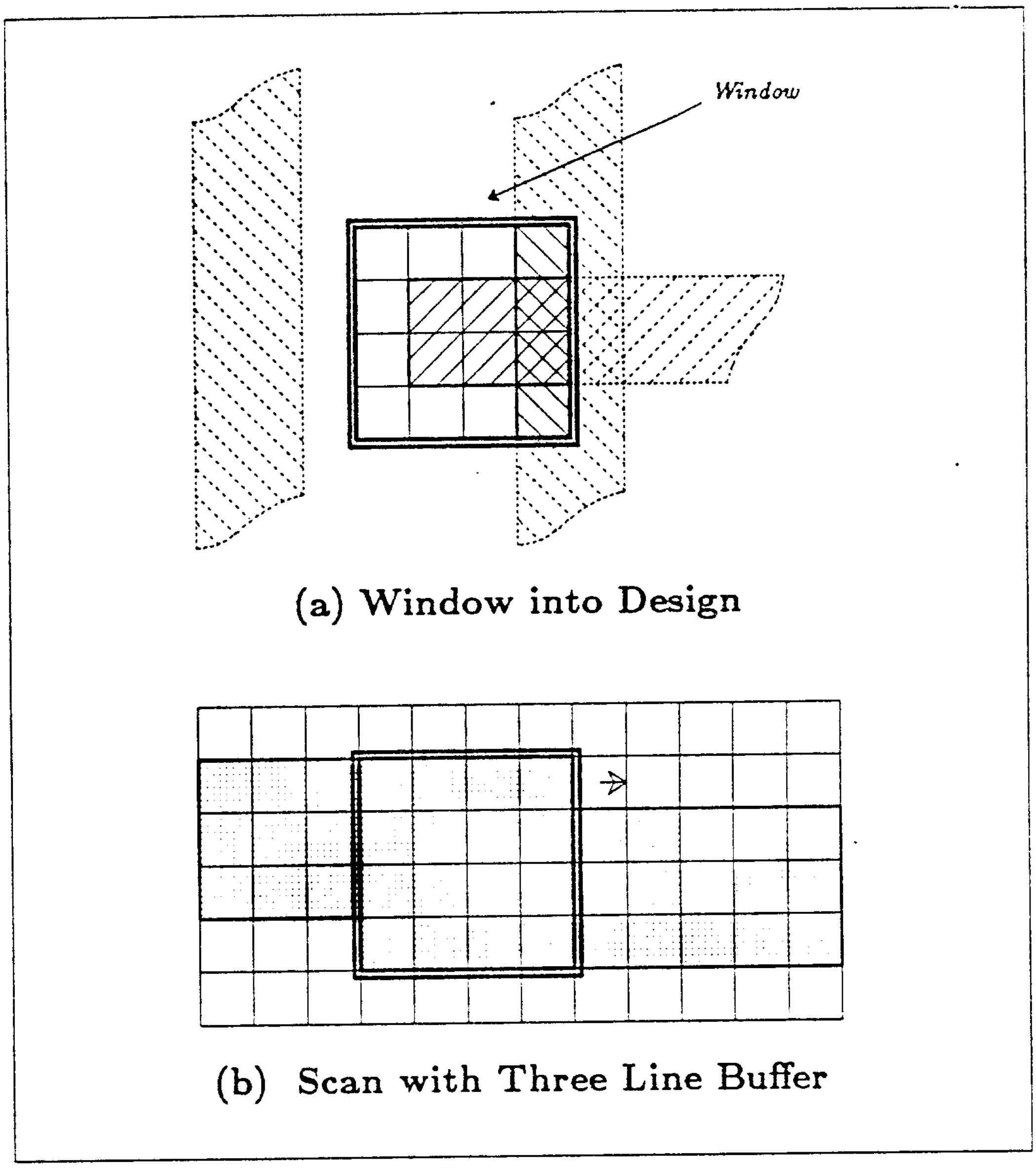


Figure 4.3. - Baker's DRC. In Baker's DRC, a small window (a) is systematically stepped across the design: the window is moved from left to right and bottom to top. This allows pixels to be read in scan-order, buffering 3 rows plus 4 pixels internally (b) for a 4x4 window. The pattern of pixels at each window position is checked for legality.

A few additional primitives are provided:

iv. **PRUNE** - remove narrow fingers.

v. **FILL** - fill narrow canyons.

vi. **EDGE-CONDITIONS** - check 2x2 user-programmable patterns.

These operations allow spurious violations to be avoided in single-layer spacing checks, and permit some simple conditional checks.

Seiler's system is designed to handle 45 degree data. This is done largely by handling partially filled pixels "appropriately" in converting data from figure-based to pixel form. Appropriate handling means marking such pixels as empty or full, depending on the operation to be performed.

The actual pixel processing is to be implemented with a special-purpose hardware DRC engine. Figure-based input data will be scanned in software via a scanline algorithm, and intervals covering the current scanline will be passed to the engine for conversion to a pixel stream and processing. Internally, data paths in the DRC engine allow for recirculation of data for sequencing of operations. Output from the engine will also be in interval form, and will need to be postprocessed for error reporting.

The hardware is designed around four custom VLSI chips which are currently only partially designed and implemented.

4.3.3. Mudge's Approach

In a paper given at the 19th Design Automation Conference [Mudge et. al. 1982], T. N. Mudge of the University of Michigan at Ann Arbor suggested how the Cytocomputer could be used to implement a pixel-based DRC check.

The Cytocomputer is a general-purpose pixel-based image processing engine currently under development. The engine is built up of an expandable series of identical stages. Pixels enter and exit each stage serially (in scan order). Each stage buffers two rows plus 3 pixels internally and outputs a bit depending on the current bit and its 8 nearest neighbors. Looked at another way, each stage implements a 3x3 window operation.

The basic operations implemented by Mudge on the cytocomputer differ from those in Seiler's DRC engine. Mudge's operations, based on the *image algebra* formalism developed by Sternberg [Sternberg 1980], rely on generalized grows and shrinks called *erosion* and *dilation*. Like Seiler's primitives, Mudge's operations can be combined to implement the traditional region-operations.

If a Cytocomputer with a sufficient number of stages is available an entire DRC check can be made with one pass through the pipeline. Mudge estimates that a full Mead-Conway check would require 250 stages. Thus given a 250 stage Cytocomputer with a 1 microsecond cycle time, a 2000 x 2000 lambda design could be processed in 10.5 seconds. However this timing estimate does not take into account the conversion of the design from figure-based to pixel form and the postprocessing of the output, both of which are likely to be very significant in practice.

Mudge illustrated how DRC operations might be implemented with a 3x3 spacing check. It is not apparent to me what form a general spacing check would take. Much work would be required to code a complete set of region-operations. At the time of the 1982 Design Automation Conference paper, only a one stage TTL prototype of the Cytocomputer was available.

4.3.4. Eustace's Approach

R. Alan Eustace and Amar Mukhopadhyay of the University of Florida at Orlando have proposed yet another pixel-based design rule checking system [Eustace & Mukhopadhyay 1982]. In their system a good deal of state is stored along with each pixel. *Two-dimensional transition functions* determine the state of each pixel based on the state of its immediate left and bottom neighbors and on the layers present at the pixel itself; see Figure 4.4. The state of all pixels can be computed by processing from left to right and bottom to top. One transition function is required for each rule to be checked. The number of states required depends on the type of check and the maximum dimension involved, for example a 3x3 spacing check can be implemented with 12 states.

No systematic method for obtaining transition functions was suggested. The authors generated transition functions by hand for a subset of the Mead-Conway rules using ad hoc methods, specifying the functions in tabular form.

The major drawbacks to this method are the large amount of state information associated with each cell, and the difficulty of specifying and implementing transition functions. The number of states required grows quickly with the maximum dimension

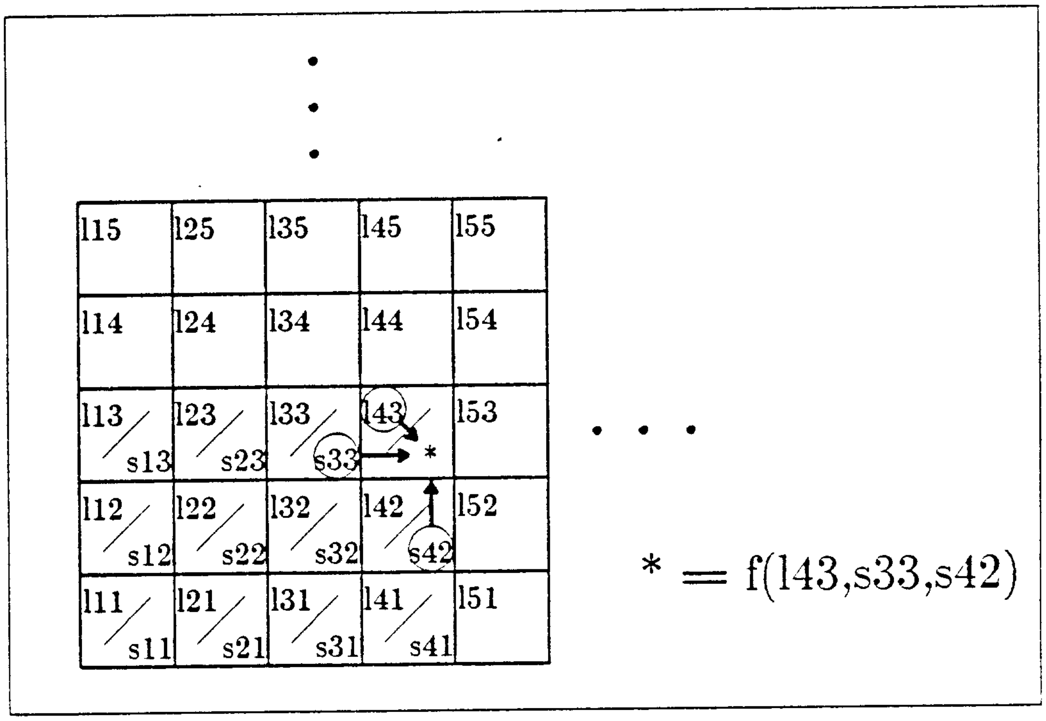


Figure 4.4. - Eustace's Approach. In Eustace's approach, state information (bottom right corners) as well as layer information (top left corners) is associated with each pixel. The state of each pixel is computed (by a *two-dimensional finite state machine* from the states of the left and bottom neighbors, and the layers present at the pixel itself. A separate state machine, and set of states, is required for each design rule. Whenever an error state is reached, a design rule violation is reported.

involved in a check, and the table space required to store a transition function grows as $O(n^2\log n)$ where n is the number of states. Thus as resolution increases, the method becomes untenable.

4.3.5. Zech's Architecture

Karl-Adolf Zech, suggests an elegant hardware organization for implementing the Eustace-Mukhopadhyay method of design rule checking [Zech 82]. This architecture requires the maintenance of state information for only a very few pixels at any one time, thus eliminating one of the major obstacles to a practical implementation of this method of design rule checking.

The basic building block for Zech's architecture is a processing element capable of computing the state of a pixel, given the states of its left and bottom neighbors, and the layers present at the pixel. Processing proceeds from left to right. Thus normally a processing element stores the state of the left neighbor internally (this is just the result of its previous computation) and takes the layer information for the current cell, and the state of the bottom neighbor as input, see Figure 4.5(a). A two-dimensional array of processing elements allows multiple rows of pixels and multiple rules to be checked simultaneously; see Figure 4.5(b) Each column of processing elements handles one rule, and each row handles one row of input pixels. The processing in each successive row is delayed one pixel with respect to the previous row, to minimizes the number of pixel states that must be maintained: each processing element stores one state internally.

Zech does not discuss the design of the individual processing elements, or the critical problem of constructing and representing transition functions for design rules.

4.4. Summary

All but a very few DRCs are of the region-operation type presented in the last chapter. However, these systems do vary in elaborateness. For example, the major DRC vendors allow

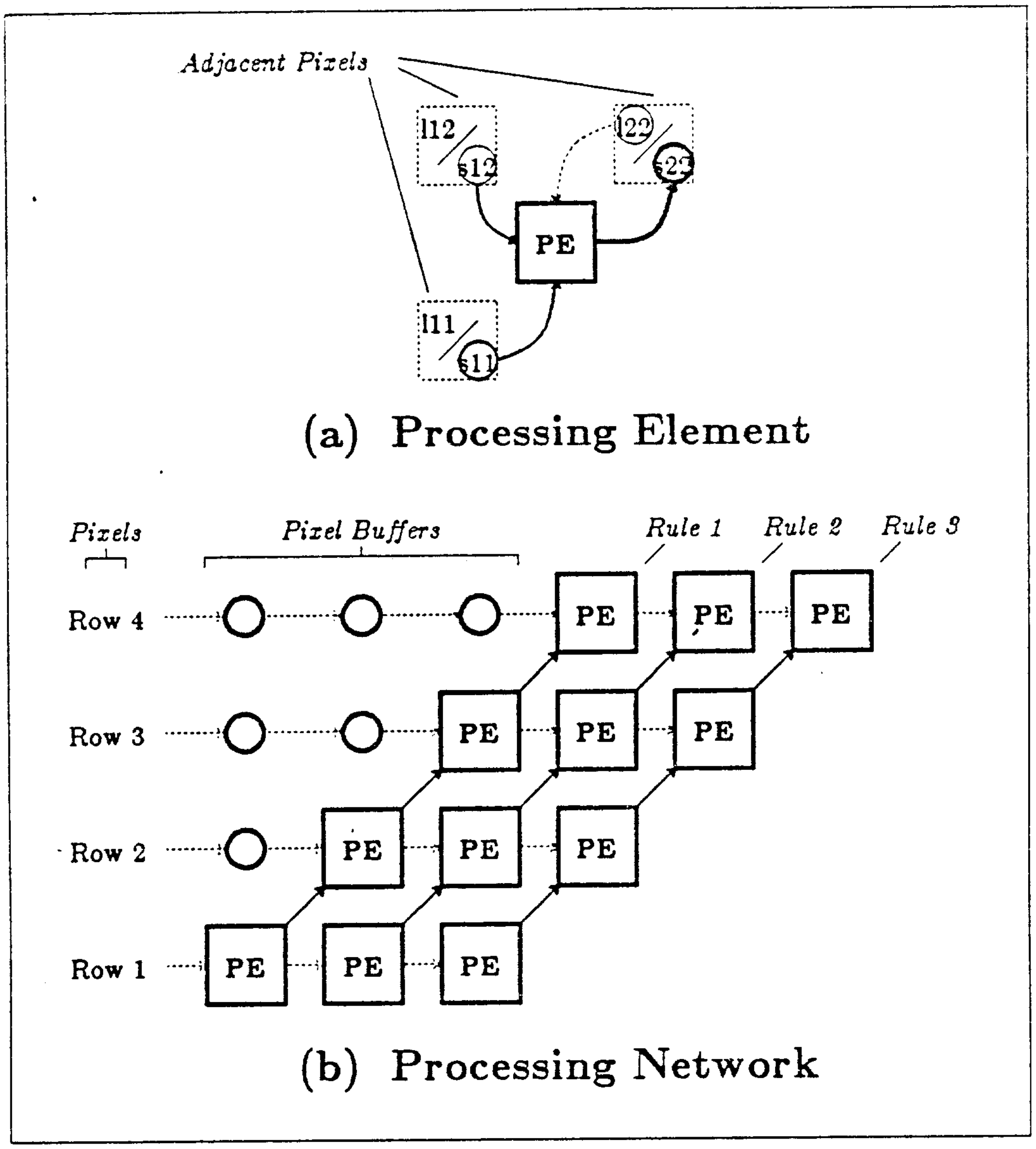


Figure 4.5. - Zech's Architecture. On each cycle, each processing element takes two states and one set of layers as input, and generates one output state, (a). Processing elements can be interconnected in a staggered array (b) to check multiple rows of pixels, and multiple design rules simultaneously.

arbitrary angle data to high resolution and permit access to extracted data to check sophisticated conditional rules. In contrast, software for Mead-Conway designs is often restricted to orthogonal data, has coarse resolution ((e.g. 1/4 the minimum transistor width), and has few provisions for conditional rules.

Region-operation systems also vary in the data structures they use to organize processing of the mask data. Typical organizations are scanlines, two dimensional bins, and sorted swaths. The basic data elements are usually edges, but may also be rectangles or trapezoids. Despite these differences, the systems remain very similar in flavor: the same primitives are used, and implementation is in terms of intersection calculations between boundaries and other boundaries, or boundaries and halo regions.

There are two other broad types of DRC's: pixel-based systems, and context-driven systems. Pixel-based systems differ from region-operation systems in the type of data representation they employ. In pixel-based systems the mask data is represented and processed as a two-dimensional array of pixels, rather than in terms of region boundaries. Pixel-based systems are largely experimental, and of interest mainly because they are amenable to parallel or pipelined special purpose hardware implementations. Only one pixel-based system, Baker's Mead-Conway DRC, has received a significant amount of actual use. There are serious questions as to the practicality of the pixel-based approach, centered around the $O(n^2)$ growth in the number of data elements needed to represent a design as the resolution requirements increase. (Here n is the diameter of the design in minimum resolvable units.)

Corner-based DRCs, the final type, differ from the region-operation approach in an even more fundamental way: there is no notion of sequencing primitive operations in context driven systems. Instead rules are represented in terms of local patterns of layers, and conditions to be checked wherever the patterns apply. Rules are independent of order and are all checked simultaneously in one pass through the data.

Corner-based systems are the topic of this thesis. Corner-based and related systems are discussed in Chapter 7, following the presentation of the corner-based approach in the next two chapters.

4.5. References

[Alexander 1981]

D. Alexander, "A Technology Independent Design Rule Checker," *3rd USA-JAPAN Computer Conference,* 1978.

[Alexander 1983]

D. Alexander, Personal Communication, NCA Corporation, Sunnyvale, California, 1983.

[Baird 1976]

H.S. Baird, *Design of a Family of Algorithms for Large Scale Integrated Circuit Artwork Analysis,* Masters Thesis, Rutgers University, 1976.

[Baker 1980]

C.M. Baker, *Artwork Analysis Tools for VLSI Circuits,* Masters Thesis, Massachusetts Institute of Technology, 1980.

[Baker & Terman 1980]

C. Baker & C. Terman, "Tools for Verifying Integrated Circuit Designs," *VLSI Design,* Vol. 1, No. 3, Third Quarter 1980.

[Eustace & Mukhopadhyay 1982]

R.A. Eustace & A. Mukhopadhyay, "A Deterministic Finite Automaton Approach to Design Rule Checking for VLSI," *Proc. 19th Design Automation Conference,* June, 1982, pp. 712-717.

[Haken 1980]

D. Haken, *A Geometric Design Rule Checker,* Internal document, Carnegie Melon University, June 1980.

[Huang 1984]

P. Huang, Personal Communication, ECAD, Inc., Santa Clara, California, 1984.

[Kozawa 1981]

T. Kozawa, "A Concurrent Pattern Operation Algorithm for VLSI Mask Data," *Proc. 18th Design Automation Conference,* June, 1981.

[Lauther 1981]

U. Lauther, "An $O(n \log n)$ Algorithm for Boolean Mask Operations," *Proc. 18th Design Automation Conference,* June, 1981, pp. 555-559.

[Mudge et. al. 1982]

T. N. Mudge, R.A. Rutenbar, R. M. Logheed and D.E. Atkins, "Cellular Image Processing Techniques for VLSI Circuit Layout Validation and Routing," *Proc. 19th Design Automation Conference,* June, 1982, pp. 537-542.

[Seiler 1982]

L. Seiler, "A Hardware Assisted Design Rule Check Architecture," *Proc. 19th Design Automation Conference,* June, 1982, pp. 232-238.

[Spink 1983]

P. Spink, Personal Communication, Phoenix Data Systems, Inc., Santa Clara, California, 1983.

[Sternberg 1980]

S.R. Sternberg, "Language and Architecture for Parallel Image Processing," *Pattern Recognition in Practice*, E.S. Gelsema and L.N. Kanal, eds., North Holland Publishing Co., 1980.

[Zech 1982]

Karl-Adolf Zech, submitted to *Journal of Information Processing and Cybernetics (EIK)*, Akademie-Verlag, Berlin, 1982.

CHAPTER 5

The Corner-Based Approach

5.1. Introduction

This chapter begins the presentation of the corner-based approach to design rule checking. It presents corner-based checking in its most general form. The form of the rules presented is likely to appear some what varied and complex. The next chapter, shows how these rules can be implemented in a uniform and relatively straight-forward manner. Current corner-based systems are less general than this, and have considerably simpler rules and implementations. These will be considered in Chapter 7.

In corner-based design rule checking, conditions are verified at mask artwork corners; see Figure 5.1. Conditions specify circular sectors in which given layer combinations must be present or absent. For example, spacing checks are coded by conditions that require a layer to be *absent* within sectors located to the outside of corners, while width checks are coded by conditions that require a layer to be *present* within sectors inside corners. There are a few embellishments. The angles of corners can be taken into account when specifying conditions: often rules specify one condition on convex corners and another on concave. Several conditions can be logically combined to specify more complex *conditional* rules. Attributes attached to the mask data can be referenced in conditions to express rules that are conditional on nongeometric information.

The above paragraph gives the entire corner-based mechanism. It has the following features:

i. Sector conditions are attached to corners of given angles and layer combinations.

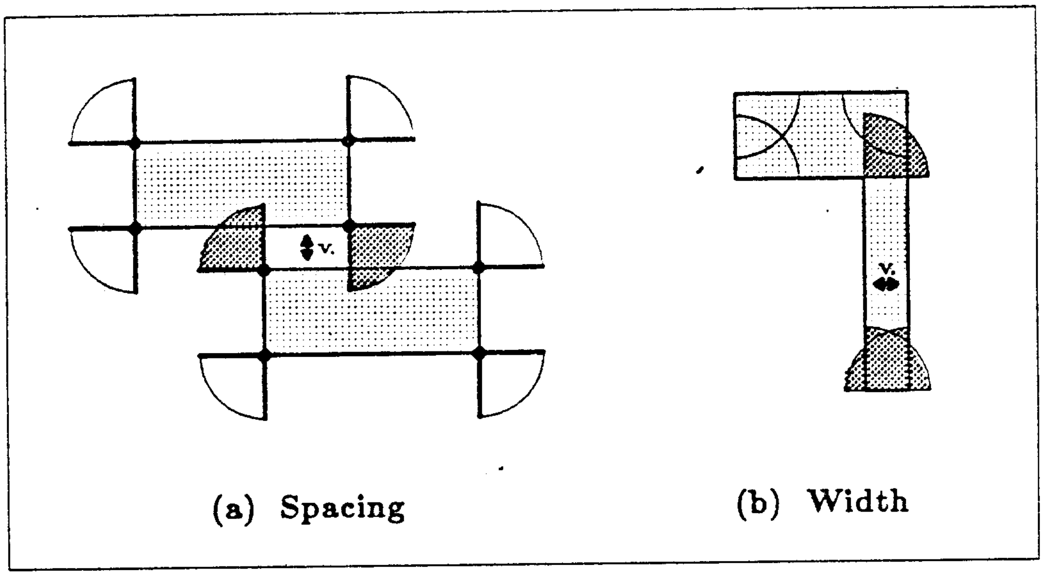


Figure 5.1. - Corner-Based Checking. The main idea of the corner-based approach is to verify circular sector conditions, attached to corners, that require the presence or absence of certain layers. Spacing is checked, (a), by verifying outward-facing sector conditions that require a layer to be absent. Width is checked, (b), by verifying inward-facing sector conditions that require a layer to be present. The shaded conditions have been violated.

ii. Sector conditions consist of circular sectors within which layer combinations must be present (or absent).

iii. Conditions can be logically combined to implement conditional rules.

iv. Layers can be qualified by reference to attributes tagged to the mask data.

The layer combinations in i. and ii. and the combination of conditions in iii. can be general boolean expressions. Features i. and ii., providing for sector conditions on corners, are powerful enough to replace all the boolean and tolerance operations of the region operation approach: they permit all unconditional rules to be checked. Feature iii., permitting the logical combination of sector conditions, allows many (though not all) rules conditional on geometric context to be checked without recourse to sizing operations. Feature iv., allowing reference to mask feature attributes when specifying layer combinations, provides an interface to nongeometric information, facilitating the checking of a variety of conditional rules.

Together these features provide a single flexible mechanism powerful enough to check most rules that can be checked by the region-operation approach. In addition, unanticipated variations on checks that would require a new primitive in a region-operation system can often be expressed without difficulty in corner-based systems.

Corner-based checking is *context-based*: the conditions that are verified at a location are determined by the local context, i.e., the corners present there, and sometimes by the validity of other conditions at the corner. Context-based checking facilitates the coding of directionally sensitive rules, common in the specification of transistors and other circuit constructs (see Figure 3.16). For example, a sector condition to the left of a corner may be made conditional on another sector condition to the right of a corner. Directionally sensitive rules are very clumsy to check with region-operations: they require long complicated sequences of operations. This is because region-operations are not easily biased by local context.

The use of context-based rules, rather than sequences of operations, permits all rules to be checked simultaneously in one pass through the mask data. Each corner in the mask data is identified, and all conditions applying to it are verified. This single pass through the mask layers, made possible by context-based rules, is a great boon to performance. It eliminates the I/O bottleneck encountered in region-operation systems where multiple passes are made through the mask layers and many intermediate layers are generated.

Corner-based checking differs from the traditional region-operation approach in another important way. In corner-based systems, tolerances are checked by sector conditions at corners: no checking is done along the length of edges; see Figure 5.2. Effectively, tolerances are implemented in terms of point/edge comparisons rather than the edge/edge comparisons traditionally used in region-operation systems. Point/edge checking allows for clean partitioning of designs: checking a piece of a design independently entails checking all the corners in that piece. Partitioning is more difficult in systems employing edge/edge processing, since edges crossing piece boundaries must be treated specially. In fact in region-operation systems, each type of operation may have to handle such edges differently. Thus the corner-based method is particularly well suited to hierarchical and incremental checking, where pieces of a design are checked independently.

The two innovations of the corner-based approach are a context-based rule description mechanism and point/edge tolerance checking. Since the introduction of corner-based checking, systems have emerged which employ each of these ideas independently. The Magic system, recently developed at Berkeley, employs a rule description mechanism, similar to the corner-based method but based on edges. Conversely, a recent region-operation system,

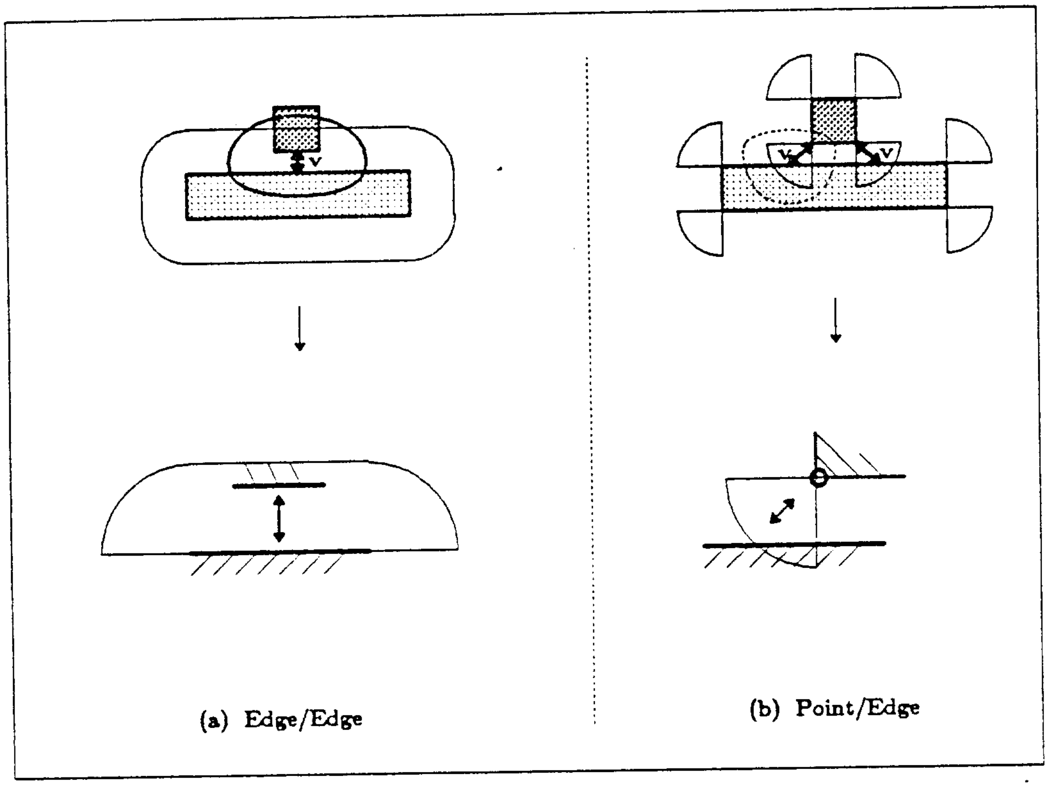


Figure 5.2. - Comparison of Tolerance Check Methods. Traditionally, tolerance checks on mask regions have been done by checking distances between region *edges*, (a). In corner-based checking, tolerances are measured from corner *points* to region boundaries, (b). Such Point/Edge checking has the advantage of very naturally splitting-up into piecewise checks: checking a piece of a design corresponds to checking tolerances from corner-points within that piece.

developed at Intel, employs point/edge tolerance operations to facilitate hierarchical checking. These systems will be considered along with the true corner-based systems in Chapter 7.

The remainder of this chapter looks at how corner-based checking works in more detail. The checking of unconditional rules, rules conditional on geometric context, and rules conditional on nongeometric information are considered in turn by the next three sections. Each section contrasts the way example rules are checked in corner-based systems with how they would be handled in a traditional region-operation system. The final section summarizes the chapter. Implementation issues will be taken up in Chapter 6, and actual corner-based systems will be considered in Chapter 7.

5.2. Unconditional Rules

This section considers how unconditional spacing, width, and enclosure rules are expressed in a corner-based system. Recall that in unconditional rules, tolerances apply to the layers throughout a design, regardless of context. In region-operation systems such a rule is checked by a sequence of boolean operations followed by a tolerance check. The boolean operations combine the mask layers to derive the regions to which the check applies, and the tolerance operation performs the actual check. In corner-based systems the required tolerance is checked by verifying circular sector conditions (equivalent to the halo corners in Figure 3.4) at corners. Since combinations of layers are permitted for corners and sector conditions, separate boolean operations are not needed.

At the beginning of this chapter, four features of the corner-based mechanism were given. Only the first two of these are needed for unconditional rules, namely:

i. Sector conditions are attached to corners of given angles and layer combinations.

ii. Sector conditions consist of circular sectors within which layer combinations must be present (or absent).

The Mead-Conway implant/enhancement-gate spacing rule will be used for illustration. This rule is shown in Figure 2.7. It requires that implant regions be spaced at least two units from enhancement gates. Enhancement gates are formed where ever polysilicon overlaps diffusion and implant is not present. In a region-operation system this rule would be checked with a sequence of operations like this:

```
Gate = Polysilicon AND Diffusion;
EnhGate = Gate AND_NOT Implant;
Violations = SPACING[EnhGate,Implant,2];
```

In the corner-based approach this rule is expressed as follows:

```
rule "Implant/Non-Implanted Gate Spacing"
  for Implant corners_require
    if corner.angle<180 then
      !(Polysilicon & Diffusion & !Implant) everywhere_in
        sector[edge1+90*,edge0-90*,2]
  for (Polysilicon & Diffusion & !Implant) corners_require
    if corner.angle<180 then
      !Implant everywhere_in sector[edge1+90*,edge0-90*,2]
```

The first **for** specifies sectors at convex (< 180 degree) implant corners; see Figure 5.3(a). Enhancement gate regions are not permitted inside these sectors. The layer of the corners and the angle restriction are indicated by:

```
for Implant corners_require
  if corner.angle<180 then ···
```

The condition on the sector (no enhancement gate regions) and the sector itself are specified by:

```
!(Polysilicon & Diffusion & !Implant) everywhere_in
  sector[edge1+90*,edge0-90*,2]
```

The layer combination '(*Polysilicon* & *Diffusion* & !*Implant*)' defines enhancement gate regions. Thus '!(*Polysilicon* & *Diffusion* & !*Implant*) **everywhere_in** ···' specifies that enhancement gate regions *not* be present anywhere in the sector. The sector specification itself has 3 parameters. The first two specify the angle of the beginning and ending edges of the sector, relative to the edges of the corner. The final parameter gives the depth of the

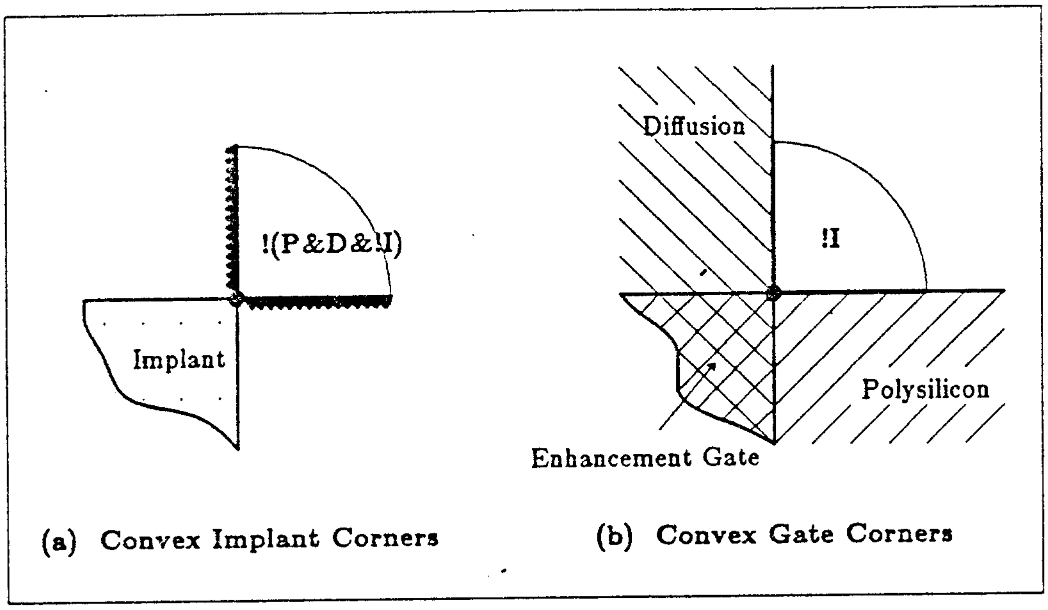


Figure 5.3. - Corner-Based Implant/Enhancement-Gate Spacing Check. Spacing between implant and enhancement-gate regions is checked by outward facing conditions on convex corners of each layer that require the absence of the other layer: part (a) checks that the *Enhancement-Gate* layer is not present outside convex implant corners, and part (b) checks that implant is not present outside convex enhancement-gate corners. The saw-toothed pattern along the sector edges in part (a) indicate that the edges are included in the sector region: features abutting these edges would be considered to intersect the sector.

sector. For the details of sector specification, see Figure 5.4. Similarly, the second **for** specifies sectors outside convex enhancement gate corners where implant must not be present; see Figure 5.3(b).

Both **for** conditions are necessary; see Figure 5.5. The sectors on the implant corners are needed to ensure that implant regions do not get too close to enhancement gate regions along implant edges (Figure 5.5(a)), and the sectors on the enhancement gate corners are needed to ensure that gate regions do not get too close to implant regions along implant edges (Figure 5.5(b)). The check must be done from both directions since checking only occurs at corners. In the region-operation approach where checking is done along the entire length of edges, it suffices to check halos around just one of the layers. This is a major difference between point/edge and edge/edge tolerance checking.

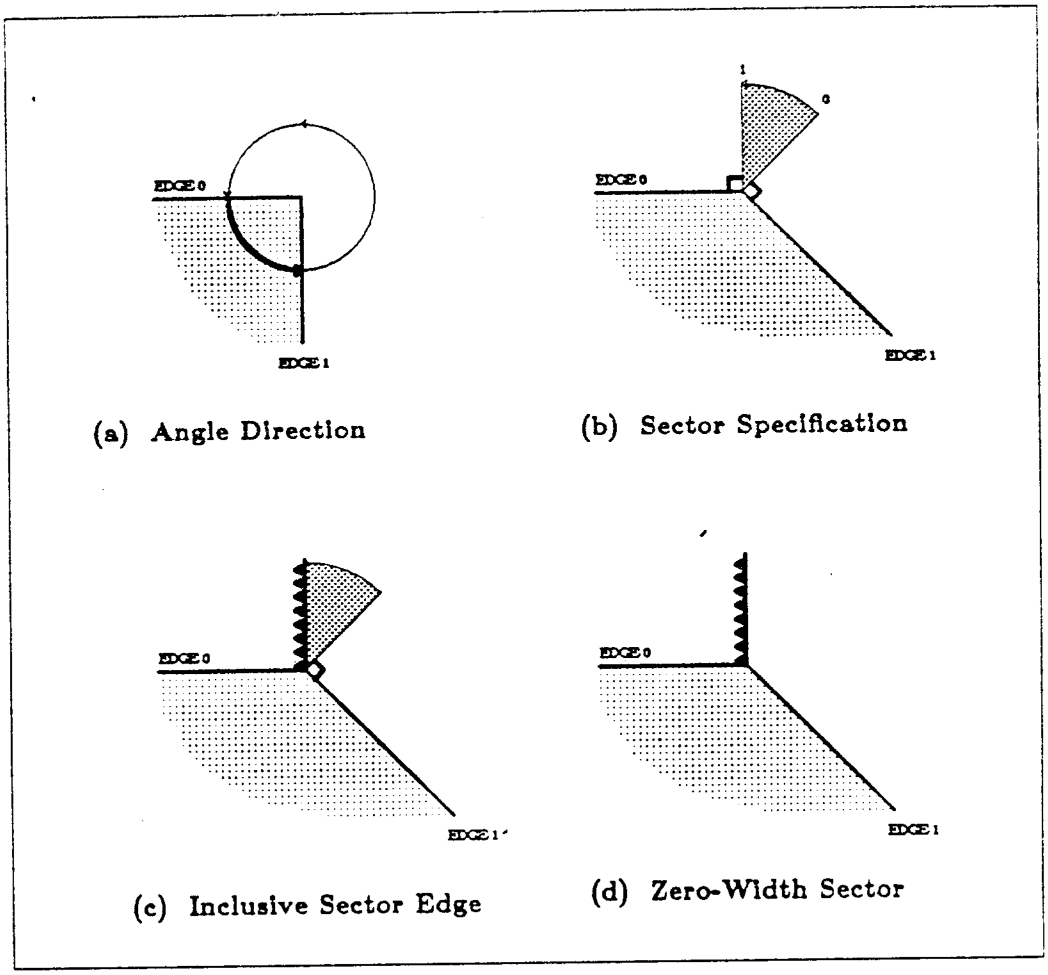


Figure 5.4. - Sector Specification. A corner's edges are arbitrarily labeled *edge*0 and *edge*1. The positive direction of rotation is defined as the direction through the interior of a corner from the corners *edge*0 to *edge*1, (a). The first and second argument of a sector specification give sector edge locations relative to the corner edges. For example, '**sector**[*edge*1+90,*edge*0−90,3]' defines a sector outside a corner with edges 90 degrees from the corner edges, (b). The interior of the sector lies between the first and second sector edges in the positive direction of rotation. The third argument of the sector specification gives its depth. Sector edges are excluded from a sector by default. Appending an asterisk, ('*'), to an edge specification indicates that the edge is to be included in the sector region, i.e. features abuting the edge from the outside are considered to intersect the sector. For instance '**sector**[*edge*1+90,*edge*0−90*,3]' specifies a sector with first edge excluded and second edge included, (c). Zero-width and zero-depth sectors are also allowed. Zero-width sectors are rays extending from a corner. The zero-width sector '**sector**[*edge*0−90*,*edge*0,3]' is shown in (d). Features abutting from the left are considered to intersect the "sector". Zero-depth sectors (not shown) are used to specify adjacencies. For example '**sector**[*edge*0,*edge*1,0]' is concerned with layers inside the corner and adjacent to the corner vertex.

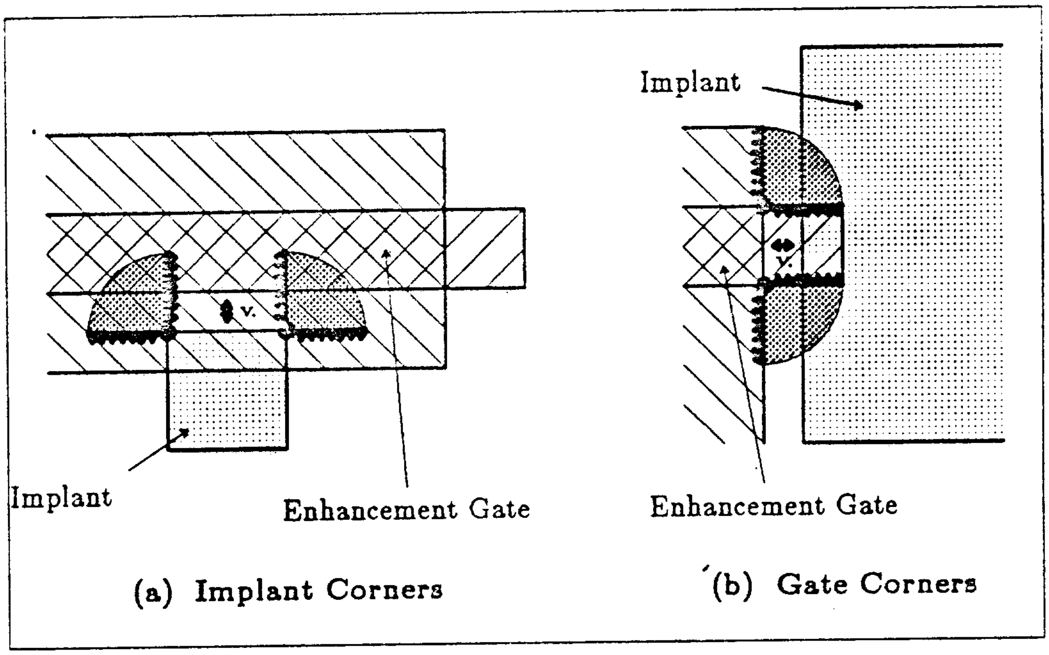


Figure 5.5. - Checking from both Directions. Both parts of the Implant/Enhancement-Gate spacing check are necessary. The check at implant corners detects spacing violations along the length of enhancement-gate edges, where no enhancement-gate corner is present, (a), and the check at enhancement-gate corners detects violations along implant edges, (b).

It is also necessary that the sectors are at least as wide as specified in the example, i.e., that they extend to within 90 degrees of the corner edges, and that regions touching the edges of the sector are considered to intrude (as indicated by '*' in the sector specification). If this were not the case, violations between parallel edges of equal extent could be missed; see Figure 5.6

It has been shown that the conditions specified in the interlayer spacing rule are necessary. But why are they *sufficient*? Why is it sufficient to check only at corners in a design? Corner-based checking depends on the fact that the closest approach between two edges always occurs at a corner. There are three cases:

i) The two edges cross.

ii) The two edges are parallel.

iii) The edges get closer together in one direction or the other.

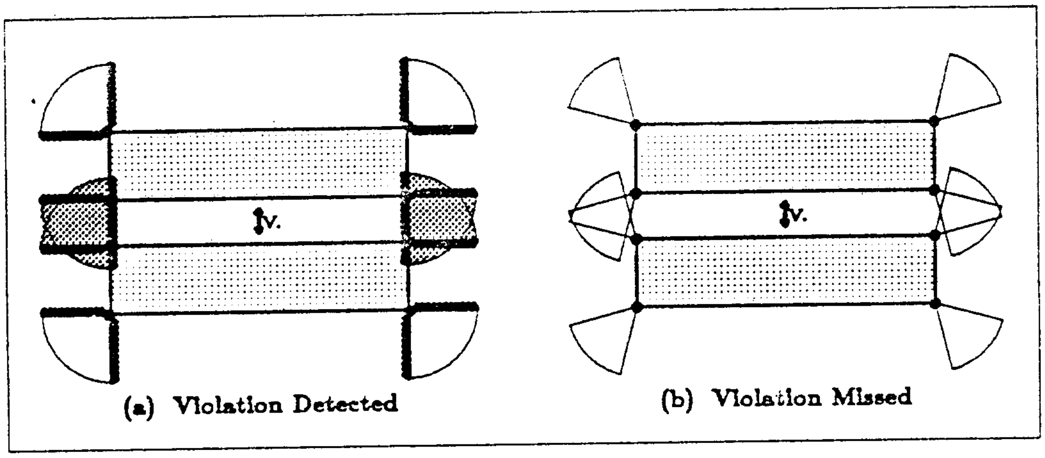


Figure 5.8. - Sector Width. In spacing checks, the condition sectors must extend to within 90 degrees of the corner edges and must inclde their bounding edges, (a). Otherwise spacing violations between parallel edges of equal extent can be missed, (b).

This is illustrated in Figure 5.7. If the edges cross, they actually *meet* at a corner. If the edges are parallel they are equally close over the entire extent they run parallel. The end of this extent must coincide with an end-point on one of the edges, again a corner. In the final case, the edges get closer together in one direction, and hence they are closest together at the end-point of one of the edges in that direction, again a corner. Thus in every case, the closest approach between two edges occurs at a corner.

The corner-based interlayer spacing check, given above, can now readily be shown correct. This check verifies that each of the layers is not present outside sectors at convex corner on the other layer extending to within 90 degrees of the corner edges. By the above argument, the closest approach between implant and enhancement gate regions involves either a corner on implant, enhancement-gate, or a corner formed by crossing implant and enhancement gate edges. Since enhancement-gate regions are only possible where implant is not present, the two layers never cross, hence there are no corners formed by crossing implant and enhancement-gate edges: the closest approach must occur at an implant corner or enhancement-gate corner. It remains only to show that it must occur within the sector

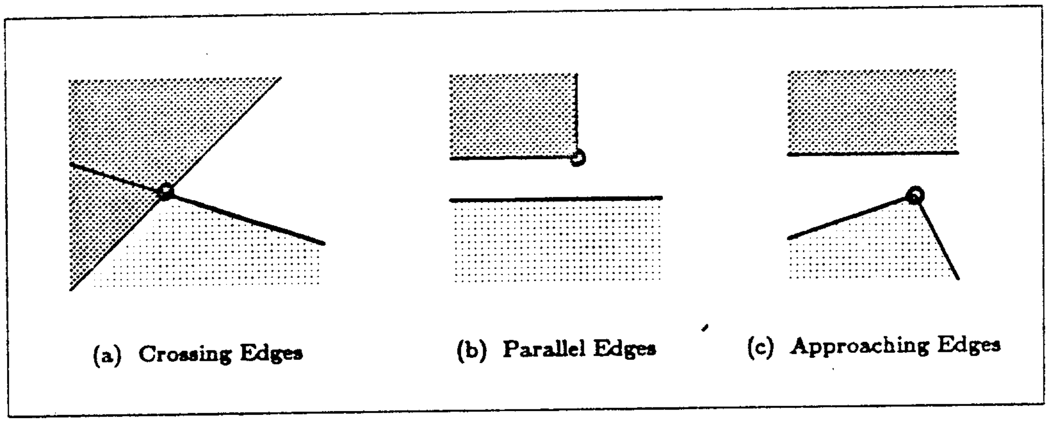


Figure 5.7. - Relationship of Corners to Edge Spacings. The closest spacing between two edges always occurs at a corner. There are 3 cases: the edges cross (a), the edges are parallel (b), or the edges approach each other in one direction or the other (c). In all three cases the closest spacing occurs at a corner.

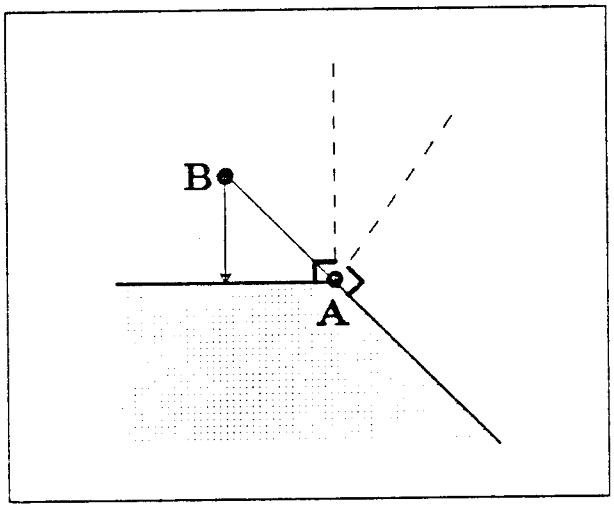


Figure 5.8. - Sufficiency of Sector Extents The sectors in the spacing check need only extend to within 90 degrees of the corner edges, since the closest approach between two layers occurs at *some* corner. Suppose it occurs between a corner A on the implant layer and some point B on the enhancement-gate layer. Then the angle between AB and the corner edges can not be less than 90 degrees, as drawn in the Figure. In this case B is closer to some other point on the enhancement-gate layer than it is to A, which is contradictory.

conditions specified by the rule for these two types of corners.

Since the cases are are symmetrical, only one need be considered. Suppose the closest approach occurs at an implant corner. Call the corner point A and the closest point of an enhancement-gate region B. Figure 5.8 shows that if the angle between the line segment AB and one of the corner edges is less than 90 degrees, then B is closer to some other point on the corner-edge than it is to A. Thus AB must be at least 90 degrees from each corner edge. This is not possible at concave corners, hence the implant corner at A must be convex, and B must be in the pie slice between perpendiculars to the corner-edges. A violation occurs if and only if the length of AB is less than the tolerance d required by the rule, i.e. if B is in the circular sector bounded by the perpendiculars and of depth d. But this is exactly the condition checked in the rule. In summary, it has been shown that if a spacing violation occurs then a point of enhancement-mode gate must be present in a circular sector of depth d outside a convex implant corner extending to within 90 degrees of the corner edges, or (by symmetry) a point of an implant region must be present inside a similar sector at an enhancement mode gate corner - exactly the conditions checked by the corner-based spacing rule given above.

To generalize the implant/enhancement-mode spacing check to arbitrary layers A and B it is necessary to explicitly check for overlap between the two layers, since in general two layers are not logically precluded from overlapping. Overlap can be detected by looking for corners on the layer A & B:

```
for A & B corners_require
   not_allowed
```

Since the presence of A & B corners alone signals a violation, no sector condition is required at these corners. Instead the degenerate condition '**not_allowed**' is used. This condition can never be satisfied. Adding the overlap check to the checks at convex corners on the two layers, yields the following general interlayer spacing rule:

```
rule "A/B Interlayer Spacing Check"
  for A corners_require
    if corner.angle<180 then
      !B everywhere_in sector[edge1+90*,edge0-90*,2]
  for B corners_require
    if corner.angle<180 then
      !A everywhere_in sector[edge1+90*,edge0-90*,2]
  for A & B corners_require
    not_allowed
```

Corner-based enclosure, single layer spacing, and width rules are implemented in similar ways; see Figure 5.9. Single layer spacing is checked with sectors outside convex corners requiring the same layer to be absent. In addition sectors outside concave corners are checked to guard against spacing violations caused by small holes in a region. A width check is equivalent to a spacing check on the complementary layer. It is implemented with inward facing sectors. The enclosure of A by B can be thought of as a spacing between A and the complement of B. This corresponds to requiring that B be present outside convex corners on A and that A be absent inside concave corners of B. All these checks can be shown correct with arguments similar to the one given above for interlayer spacing. The crux of these arguments is that the closest approach between two edges always occurs at a corner.

It would be cumbersome to express all spacings, widths, and enclosures directly in terms of the corner-based mechanism. Instead macros are used for the most common rules, e.g.,

```
Spacing2[A,B,2];
Spacing[A,3];
Width[A,3];
Enclosure[A,B,2];
```

These macros expand into the corner-based rules described above. More unusual rules can always be written directly in terms of the underlying mechanism, and thus exploit the full flexibility of the corner-based approach.

To illustrate the flexibility of the corner-based mechanism, one more unconditional rule will be developed: a facing edge rule. In region-based systems such rules require a special primitive (e.g. a perpendicular-only grow operation).

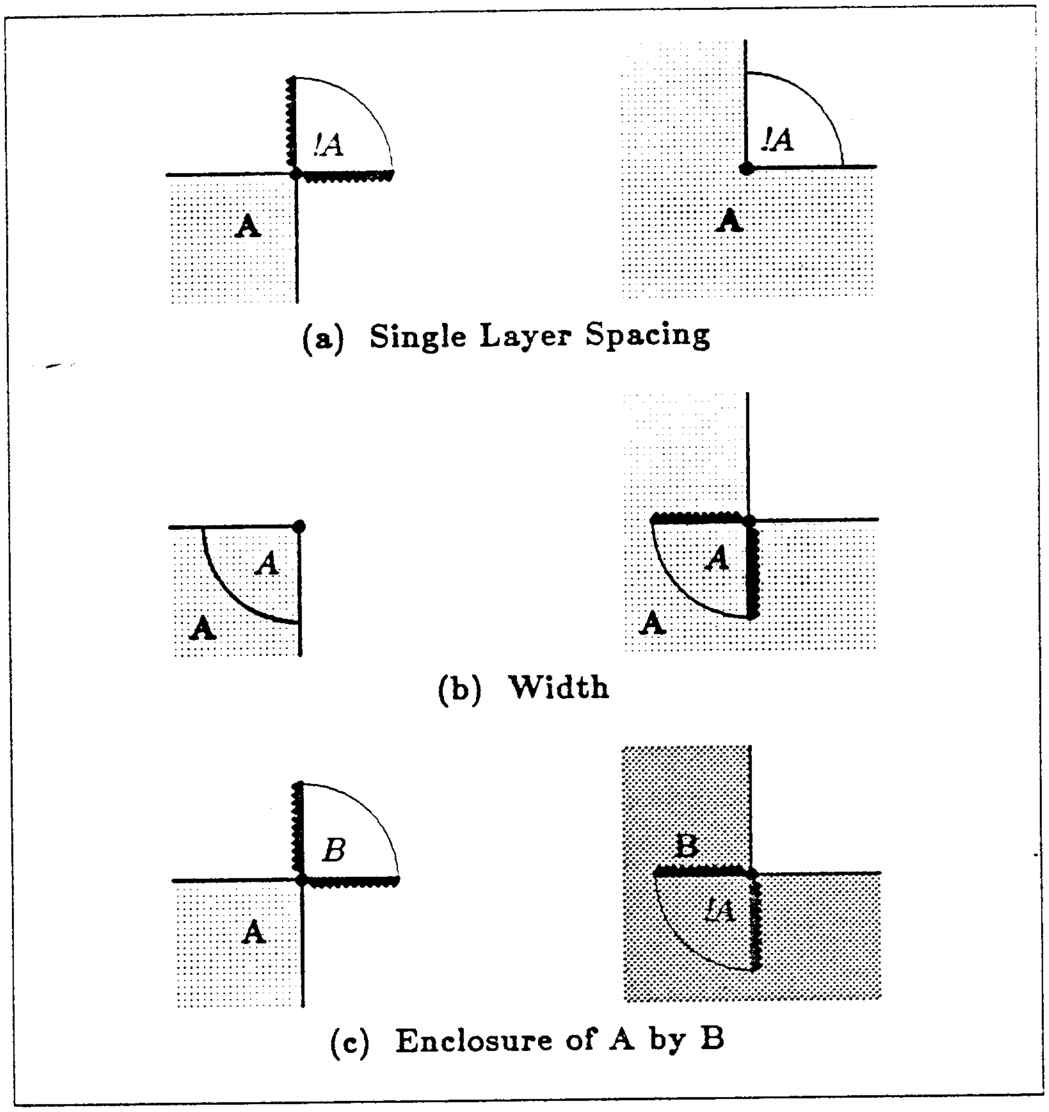


Figure 5.9. - Other Unconditional Tolerance Checks. Corner-based unconditional single layer spacing, width, and enclosure checks are all similar to the interlayer spacing check developed above. Single layer spacing, (a), involves sectors outside corners prohibiting the presence of the *same* layer. Sectors at concave corners check for spacing violations resulting from small holes in a layer. Width, (b), is checked by inward-facing sectors that require the presence of the layer for a minimum radius. Enclosure, (c), is checked by verifying the presence of the enclosing layer outside convex corners of the enclosed layer, and verifying the absence of the enclosed layer inside concave corners of the enclosing layer.

Facing edge checks, as in Figure 3.5, specify minimum tolerances between *facing edges*, but do not restrict diagonal spacings, as in Figure 3.5. Such rules are motivated by the physical properties of the resist used in patterning the layer: narrow resist slivers resulting from closely spaced facing edges might tear off and float to another sight on the wafer causing a fatal flaw there. For example one ruleset requires facing implant regions to be separated by at least 2 units. Such a rule can be checked with *zero width* sector conditions perpendicular to corner-edges, run out as feelers, as in Figure 5.10. Zero-width sectors can be visualized as having a slight width in the direction(s) of the *'ed edge(s). Conceptually this check is derived by removing the sector interior from a spacing check. This rule can be written as follows:

```
rule "Two Unit Facing Edge Check for Implant"
  for Implant corners_require
    !Implant everywhere_in sector[edge0−90,edge0−90*,2] and
    !Implant everywhere_in sector[edge1+90*,edge1+90,2]
```

This section has shown how the basic idea behind corner-based checking: verifying circular sectors at corners for the presence or absence of layer combinations, can be used to implement the standard unconditional tolerance checks. In addition the flexibility of the method has been demonstrated with a more unusual check: a facing edge check. The

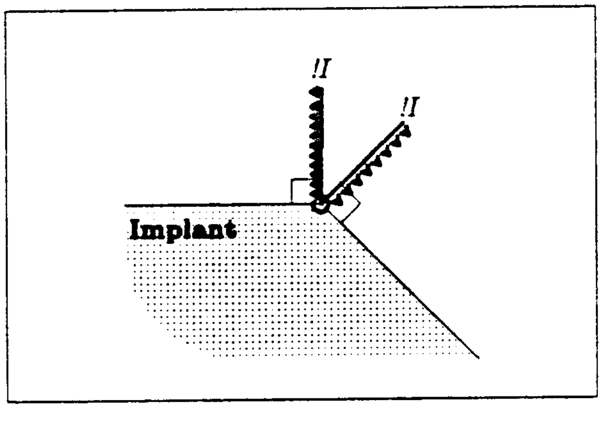


Figure 5.10. - Corner-Based Implant Facing Edge Check. Facing edge checks can be implemented with zero width conditions or *fingers* extending perpendicularly outwards from the corner edges. As usual, the saw-toothed sides of the fingers indicate inclusive sector edges: mask regions touching the fingers on those sides will be considered to impinge on the sector.

following two sections show how corner-based checking can be extended to conditional rules.

5.3. Geometric Conditional Rules

Geometric conditional rules are rules where tolerances between features depend on the geometric context in which they are found. Examples of conditional rules are spacing rules that depend on the configuration of an underlying layer (Figure 2.9), and width rules that depend on how densely features are spaced (Figure 2.11). Rules governing transistor or contact form, for example the extension form rule illustrated in Figure 3.16, also depend on geometric context.

In region-operation systems, such rules are checked by preceding a tolerance check operation by a sequence of, sizing, topological, and boolean operations that extract the regions to which the tolerance applies. In corner-based systems interrelated sector conditions are used to implement conditional checks. For example a tolerance might be enforced by one sector condition, only if a certain layer is detected in the vicinity by another sector condition. Such rules differ from the unconditional rules of the last section in that they use the third feature of the corner-based mechanism:

iii. Conditions can be logically combined to implement conditional rules.

Some geometric conditional rules can be checked using only the corner-based mechanism. In other cases the required context is not available at corners, and must be established by sizing or topological operations prior to checking. Even in these cases, one or two region operations usually suffice. This is many fewer than would be required by a traditional region-operation system. This section illustrates corner-based checking of geometric conditional rules with two examples. The first one, a transistor extension rule, is checked by the corner-based mechanism alone, without any preceding region-operations. This rule involves directional context, and is extremely difficult to check in a region-operation system (the best method I know requires 18 supporting region-operations in addition to the two extension checks). The second example is a reflection rule, where the minimum spacing between metal lines depends on the configuration of nearby polysilicon. Two region-operations, an AND and a GROW, are required to provide enough context for the corner-based check. In a pure region-operation system this check requires 8 operations.

5.3.1. Transistor Extension Rule

This section gives a complete transistor extension rule for MOS processes. The rule requires that transistor extensions are present at every gate edge, and that extensions enclose transistor corners. This is illustrated in Figure 3.16. Special effort is required to check that extensions are present along the entire length of gate edges, i.e. that extensions are not notched; see Figure 5.11. In addition to extension form, the size of extensions is checked: extensions are required to be at least 2 units long.

The following sequence of region operations checks this rule:

```
V1 = EXTENSION(P,D,2)
V2 = EXTENSION(D,P,2)

PorD = P OR D
Gate = P AND D
GateP1 = GROW_PERP(Gate,2)
V3 = GateP1 AND_NOT PorD
GateH1 = GROW(Gate,.01)
AtCorners = GateH1 AND_NOT GateP1
GateH2 = GROW(Gate,2)
Corners = GateH2 AND_NOT GateP1

PureP = P AND_NOT D
PCorner = TOUCHING(AtCorner,PureP)
NotPCorner = AtCorner AND_NOT PCorner
NeedD = OVERLAPPING(Corners,NotPCorner)
V4 = NeedD AND_NOT D

PureD = D AND_NOT P
DCorner = TOUCHING(AtCorner,PureD)
NotDCorner = AtCorner AND_NOT DCorner
NeedP = OVERLAPPING(Corners,NotDCorner)
V5 = NeedP AND_NOT P
```

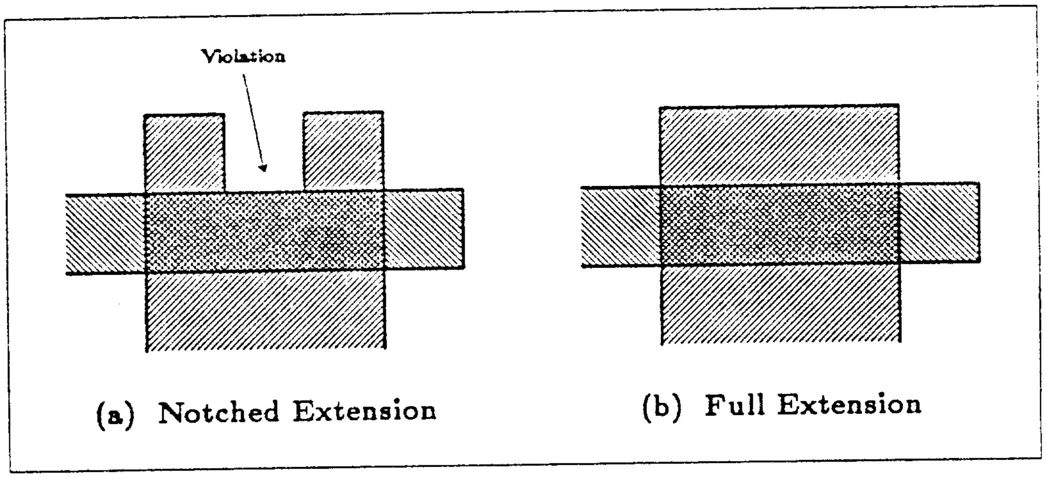


Figure 5.11. - Notched Extensions. Transistor extensions must not be notched as in (a). Rather they must be present along the full length of the transistor edge, (b). Notches must be checked for explicitly in corner-based extension checks.

Violations are written to the layers $V1$ through $V5$. The 2 extension operations check the size of polysilicon and diffusion extensions where ever they appear. The following 4 operations check that polysilicon or diffusion extensions are present along every gate edge. The remaining 14 operations check that transistor corners are enclosed by extensions of sufficient depth.

The corner-based version of this rule is shown in Figure 5.12. It involves conditional checks at polysilicon corners, diffusion corners, gate corners, pure-polysilicon (no diffusion present) corners, and pure-diffusion (no polysilicon present) corners. Together, checks at these corners suffice to verify the form and dimensional constraints for transistors.

The complete text for this rule is given below. The graphical representation, given in Figure 5.12, is probably easier to understand. Once conceived of graphically, the translation into the rule language is not difficult. Also, a few macros would make the rule briefer and more readable.

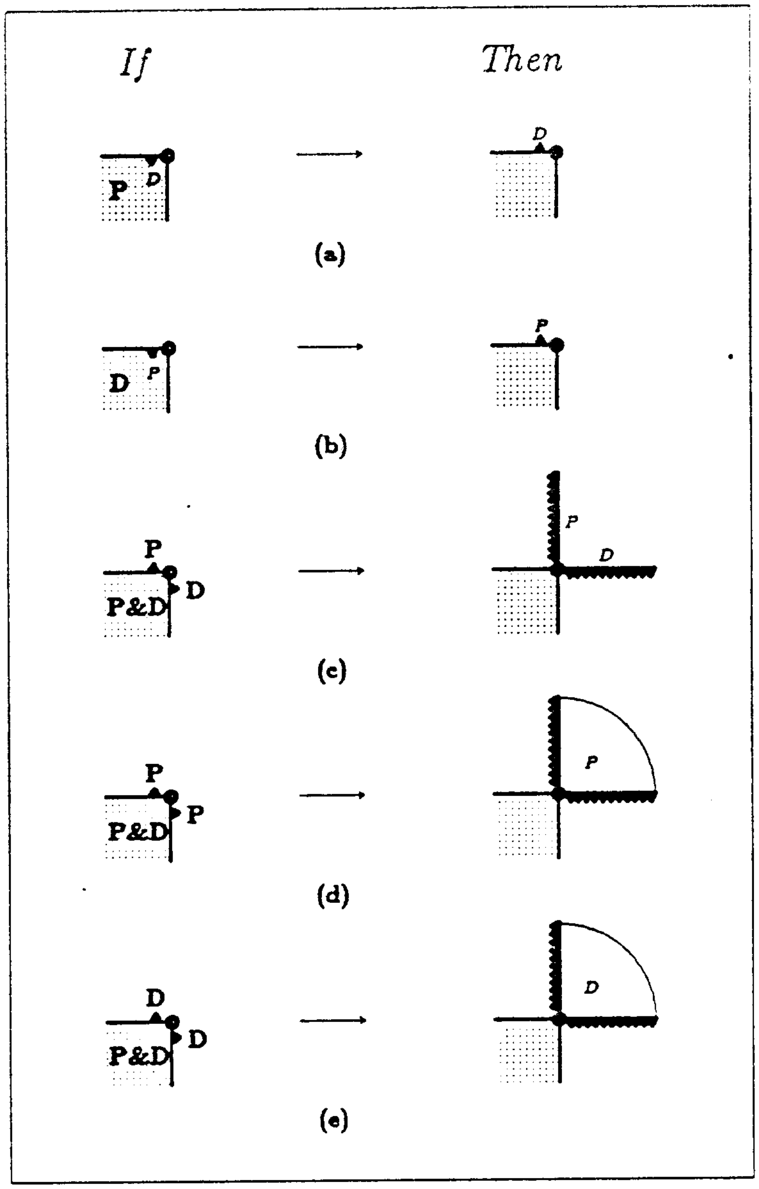


Figure 5.12. - Corner-Based Extension Rule; First Section. Parts (a) and (b) check all polysilicon and diffusion corners to make sure that polysilicon and diffusion edges never coincide. This ensures that gate edges are always flanked by extensions. Part (c) checks the size of extensions at regular gate corners, i.e. corners where polysilcion extends out from one corner edge and diffusion from the other. Parts (d) and (e) check for extensions around corners of bent transistors.

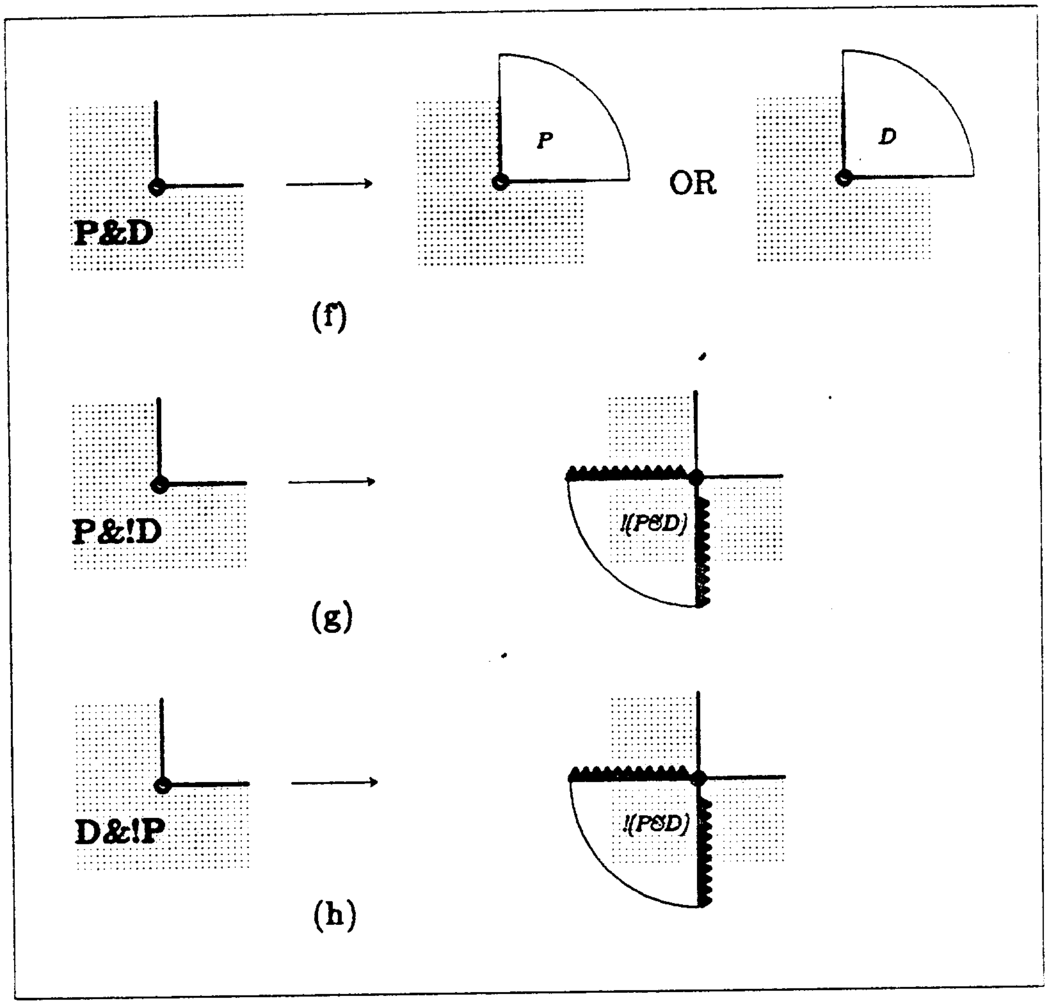


Figure 5.12. - Corner-Based Extension Rule; Final Section. Part (f) checks that either polysilicon or diffusion extension is present inside concave bends in transistors. Parts (g) and (h) check inside concave pure-polysilicon (no diffusion present) and pure-diffusion (no polysilicon) corners. They verify that notches in extension do not result in insufficient extension depth.

```
rule "Transistor Extensions"

  /* (a) Check for diffusion extensions along transistor edges
  for P corners_require
     if D everywhere_in sector[edge0,edge0*,0] then
          D everywhere_in sector[edge0*,edge0,0]
     and if D everywhere_in sector[edge1*,edge1,0] then
          D everywhere_in sector[edge1,edge1*,0]
```

```
/* (b) Check for polysilicon extensions along gate edges
for D corners_require
    if P everywhere_in sector[edge0,edge0*,0] then
        P everywhere_in sector[edge0*,edge0,0]
    and if P everywhere_in sector[edge1*,edge1,0] then
        P everywhere_in sector[edge1,edge1*,0]

/* (c) Check extension dimensions at normal gate corners. */
for (P & D) corners_require
    if corner.angle<180 then
        if (P everywhere_in sector[edge0*,edge0,0] and
        D everywhere_in sector[edge1,edge1*,0]) then
            P everywhere_in sector[edge0-90,edge0-90*,2]  and
            D everywhere_in sector[edge1+90*,edge1+90,2]
        elseif (D everywhere_in sector[edge0*,edge0,0] and
        P everywhere_in sector[edge1,edge1*,0]) then
            D everywhere_in sector[edge0-90,edge0-90*,2] and
            P everywhere_in sector[edge1+90*,edge1+90,2]

/* (d) Check that diffusion extensions enclose bends in transistors */
for (P & D) corners_require
    if corner.angle<180 then
        if (D everywhere_in sector[edge0*,edge0,0] and
        D everywhere_in sector[edge1,edge1*,0]) then
            D everywhere_in sector[edge1+90*,edge0-90*,2]

/* (e) Check that polysilicon extensions enclose bends in transistors */
for (P & D) corners_require
    if corner.angle<180 then
        if (P everywhere_in sector[edge0*,edge0,0] and
        P everywhere_in sector[edge1,edge1*,0]) then
            P everywhere_in sector[edge1+90*,edge0-90*,2]

/* (f) Check that concave transistor corners are enclosed by an extension */
for (P & D) corners_require
    if corner.angle>180 then
        P everywhere_in sector[edge1,edge0,2] or
        D everywhere_in sector[edge1,edge0,2]

/* (g) Check diffusion corners to complete D extension check */
for (D & !P) corners_require
    if corner.angle>180 then
        !(P & D) everywhere_in sector[edge0+90*,edge1-90*,2]

/* (h) Check polysilicon corners to complete P extension check */
for (P & !D) corners_require
    if corner.angle>180 then
        !(P & D) everywhere_in sector[edge0+90*,edge1-90*,2]
```

Parts (a) and (b) check all polysilicon and diffusion corners to make sure that polysilicon and diffusion edges do not coincide. These checks involve zero-depth sector conditions, for example:

P **everywhere_in sector**[*edge*0*,*edge*0,0]

Zero-depth conditions check adjacencies rather than tolerances; they can be visualized as sectors with very small radius. In the example, the condition specifies that either polysilicon is present directly outside the corner and just above one of the corners edges (*edge*0). Part (c) checks the size of the extensions at normal transistor corners (i.e. corners where polysilicon extends from one of the corner edges, and diffusion from the other). Parts (d) and (e) check to see that extensions enclose corners of bent transistors (recognized by the fact that either polysilicon or diffusion extends from *both* corner-edges). Part (f) checks for extensions at concave transistor corners. Finally parts (g) and (h) complete the extension tolerance check by looking back from polysilicon and diffusion corners. This final check is required to guard against violations resulting from notches.

This example illustrates how corner-based extension tolerances are checked. For example, polysilicon extensions are checked, at convex gate corners that have adjacent polysilicon (and at concave pure polysilicon corners). This is actually a conditional check: the tolerance applies only at those gate corner where polysilicon is adjacent. This is why extension checking was not considered alongside spacing, width and enclosure checks in the last section.

In the corner-based approach form and directional context are checked via conditional checks. The complete extension check above involved conditions on 5 corners: polysilicon, diffusion, gate, pure-polysilicon and pure-diffusion. By contrast the form part of the region-based extension check required 13 operations and 11 intermediate layers.

5.3.2. Reflection Rule

Recall that polysilicon edges to the inside of metal edges can reflect light laterally during patterning of the resist for the metal layer, moving the metal edge outward from its intended position; see Figure 2.4. To take this phenomenon into account, reflection rules require greater spacing between metal lines when polysilicon edges lie nearby. For example, metal edges may be required to be 1 unit apart everywhere, 1.5 units apart when one of the metal edges is affected by reflection (a suitable polysilicon edge lies within one unit of the metal edge) and 2 units apart when both metal edges are affected by reflection; see Figure 2.9.

In the region-based approach, this rule can be checked by a 1-unit spacing check on a modified metal layer where edges affected by reflection have been moved outward by .5 units.

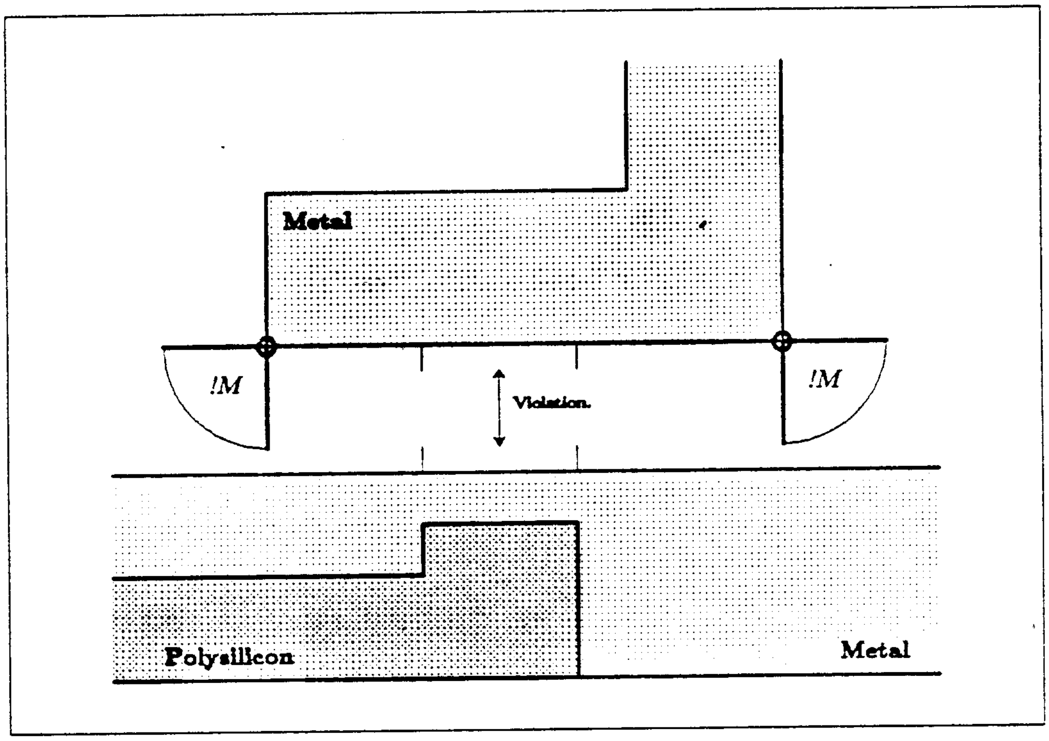


Figure 5.13. - Problem with Corner-Based Reflection Check. Above, a bulge in underlying polysilicon results in a metal spacing violation along a section of metal edge. Note that no *metal* corner, and in fact no corner of any kind is present at the site of the violation. The corner-based mechanism alone cannot detect such violations.

The details of this check were developed in Chapter 3. It requires 3 boolean operations, 4 sizing operations, and the spacing tolerance check.

This rule poses a problem for the corner-based approach, since the necessary information is not available at the corners of the design. For example, Figure 5.13 shows how checking at metal corners can miss a spacing violation involving a metal edge section affected by reflection. Note that there is no corner present along the section of metal edge where this spacing needs to be checked.

A corner-based check of this rule is possible, if the check is preceded by region operations that identify the metal edge sections affected by reflection. For example, the following two operations do the trick:

$$\begin{aligned} MP &= M \textbf{ AND } P \\ RefX &= \textbf{GROW}(MP, 1.0) \end{aligned}$$

All metal edges that are covered by the *RefX* layer (reflection context) but not by polysilicon are affected by reflection. Using the *RefX* layer, a corner-based version of the reflection rule can be constructed by checking both metal corners and corners of metal affected by reflection; see Figure 5.14. Sector conditions of radius 1.0 and 1.5 units at convex metal corners check tolerances to any metal and affected metal respectively. Similarly, sectors of radius 1.5 and 2.0 units at convex affected metal corners check tolerances to any metal and affected metal respectively. In situations such as the one depicted in Figure 5.13, affected metal corners occur in the middle of a metal edge. In these cases it is sufficient to check tolerances only in the direction perpendicular to the edge. The complete rule can be written as follows:

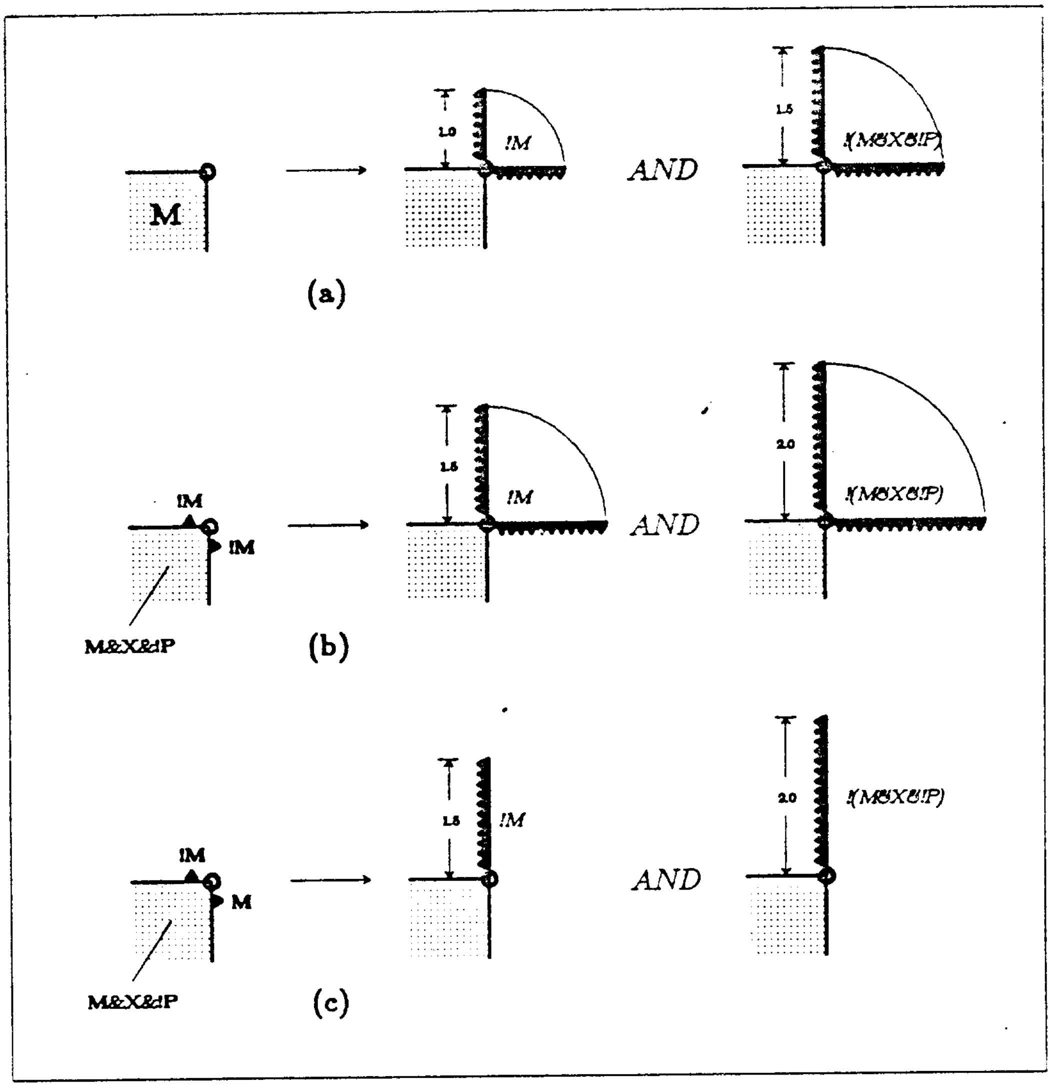


Figure 5.14. - Corner-Based Reflection Check. The corner-based reflection check distinguishes between ordinary metal and metal affected by reflection. Metal affected by reflection is defined as '*Metal & RefX & !Polysilicon*', where *RefX* is generated prior to checking as described in the text. Part (a) checks the spacing from ordinary metal corners to both ordinary metal and affected metal. Similarly part (b) checks the spacing from affected metal corners to ordinary metal and affected metal. Part (c) checks perpendicular spacings from affected metal corners occurring in the middle of a metal edge. Such corners arise in situations such as the one depicted in Figure 5.13.

```
rule "Metal reflection"

  /* Normal Metal corners */
  for M corners_require
  if corner.angle<180 then
    !M everywhere_in sector[edge1+90*,edge0-90*,1.0] and
    !(M & RefX & !P) everywhere_in sector[edge1+90*,edge0-90*,1.5]

  /* Corners of Metal affected by reflection NOT in middle of M edges */
  for (M & RefX & !P) corners_require
    if corner.angle<180 then
      if (!M everywhere_in sector[edge0*,edge0,0] and
      !M everywhere_in sector[edge1,edge1*,0]) then
        !M everywhere_in sector[edge1+90*,edge0-90*,1.5] and
        !(M & RefX & !P) everywhere_in sector[edge1+90*,edge0-90*,2.0]

  /* Corners of Metal affected by reflection in middle of M edges */
  for (M & RefX & !Poly) corners_require
    if corner.angle<180 then
      if (!M everywhere_in sector[edge0*,edge0,0] and
      M everywhere_in sector[edge1,edge1*,0]) then
        !M everywhere_in sector[edge0+90,edge0+90*,1.5] and
        !(M & RefX & !P) everywhere_in sector[edge0+90,edge0+90*,2.0]

  for (M & RefX & !Poly) corners_require
    if corner.angle<180 then
      if (M everywhere_in sector[edge0*,edge0,0] and
     !M everywhere_in sector[edge1,edge1*,0]) then
        !M everywhere_in sector[edge1+90*,edge0+90,1.5] and
        !(M & RefX & !P) everywhere_in sector[edge1+90*,edge1+90,2.0]
```

5.4. Nongeometric Conditional Rules

Some design rules are conditional on connectivity. For example, the spacing required between distinct nodes is often greater than the minimum spacing between pieces of a single node. Other rules are conditional on electrical information, such as the resistance of a node, or on the intended function of a node. Chapter 2 gives examples of rules of each of these types. All these rules depend on information that is not directly available in the mask description: rather it must be derived by an analysis program such as a circuit extractor, or supplied by the designer. Such rules are called *nongeometric* conditionals.

In region-based systems nongeometric information is tagged to the figures or line segments comprising the mask data. For example, each line segment is assigned a node number for use in connectivity-dependent checks. Special operations exist to generate the nongeometric information and tag the mask data. Once the mask data is tagged, *selection* operations can be performed that output only relevant mask features for tolerance checking. For example, a special width rule for VDD and GND buses can be implemented by preceding the width tolerance operation, with a selection operation that only outputs mask regions that are tagged as such. Some nongeometric conditionals, notably internode spacing rules, cannot be implemented with selection operations. The reason is that whether a tolerance applies between features in these rules does not depend on the individual values of the tagged data, but rather on the relationship between the tagged data of pairs of features. For example, internode spacing tolerance apply to pairs of features that have distinct node numbers. Such rules must be implemented by special tolerance operations that compare the tagged data of every edge-pair before doing the tolerance check on that pair.

The corner-based mechanism described in this chapter cannot generate nongeometric information. Thus any nongeometric information that is to be used in corner-based design rule checking must be provided by an external analysis program (e.g. by a region-based circuit extractor) or by the circuit designer. If nongeometric information is available, the corner-based mechanism can use this information for nongeometric conditional checks. Nongeometric conditional checks use the fourth and final feature of the corner-based mechanisms:

iv. Layers can be qualified by reference to attributes tagged to the mask data.

For example, '*Metal*[*function*="PWRBUS"]' is used to specify metal regions tagged with a *function* attribute of "PWRBUS", and '*Diffusion*[*node*≠*corner.node*]' refers to diffusion regions with node number distinct from the current corner.

There are three examples in this section. The first two illustrate how connectivity-dependent rules can be implemented in corner-based systems. The third example, a width check for VDD and GND buses, is typical of rules that are implemented with selection operations in region-based systems. It shows how references to tag-data in layer specifications can be used in place of selection operations.

5.4.1. Internode Spacing

The most common nongeometric conditional rules specify minimum spacings for distinct nodes on a layer. Such a rule can be implemented by a condition outside convex corners that does not permit the presence of the same layer unless it has the same node number. For example, a 3-unit minimum internode spacing on the diffusion layer is checked by the following rule:

```
rule "Diffusion Internode Spacing"
  for Diffusion corners_require
    if corner.angle<180 then
      !Diffusion[node≠corner.node] everywhere_in
       sector[edge1+90*,edge0−90*,3]
```

The expression '*Diffusion*[*node*≠*corner.node*]' refers to diffusion regions with node number not equal to the node number of the current corner. This rule is illustrated in Figure 5.15. Unlike the unconditional single layer spacing rule, this rule does not attach conditions to concave corners. Conditions on concave corners are not needed, in this case, because the material around a small hole must all belong to the same node.

5.4.2. Buried to Unrelated Polysilicon Spacing

Design rules require polysilicon and diffusion lines that do not actually join in a buried contact region to be spaced a certain minimum distance from the buried region. This guards against accidental contacts. For example the minimum spacing between a buried region and unrelated polysilicon might be 2 units. This is another example of a connectivity-dependent rule. If the common node number of the polysilicon and diffusion in buried contacts is assigned to the buried region as well, then buried/polysilicon spacing can be checked as follows:

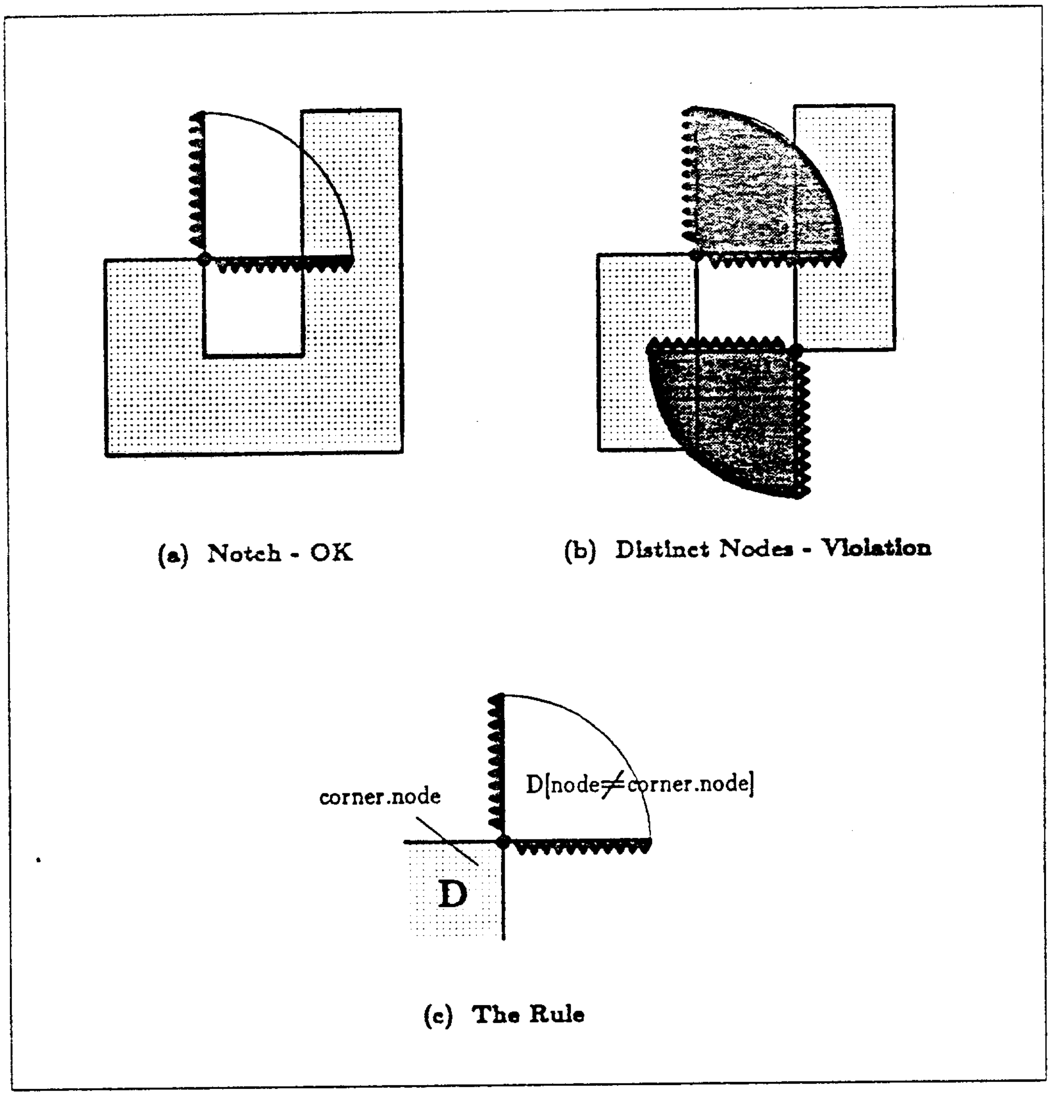


Figure 5.15. - Corner-Based Internode Spacing Check. An internode spacing rule permits close spacing between sections of the same node, (a), but guards against too-closely-spaced distinct nodes (b). Such a rule can be checked by qualifying the layer specification in the sector condition to refer only to regions belonging to different nodes, (c). Here *node* refers to the node of the region inside the sector and *corner.node* refers to the node of the corner.

```
rule 'Buried to Unrelated Poly Spacing'
  for Buried corners_require
    if corner.angle<180 then
      !Polysilicon[node=corner.node] everywhere_in
        sector[edge1+90*,edge0−90*,2]

  for Polysilicon corners_require
    if corner.angle<180 then
      !Buried[node=corner.node] everywhere_in
        sector[edge1+90*,edge0−90*,2]
```

This rule is illustrated in Figure 5.16. It is exactly like an unconditional interlayer spacing rule, except that the sector conditions are qualified by node information. A similar rule checks buried/diffusion spacing.

5.4.3. Power and Ground Bus Width

Since power and ground buses must carry large amounts of current, design rules sometimes require them to be wider than other metal lines. For example metal may be required to be 3 units wide everywhere, but 5-units wide in power and ground buses. If this

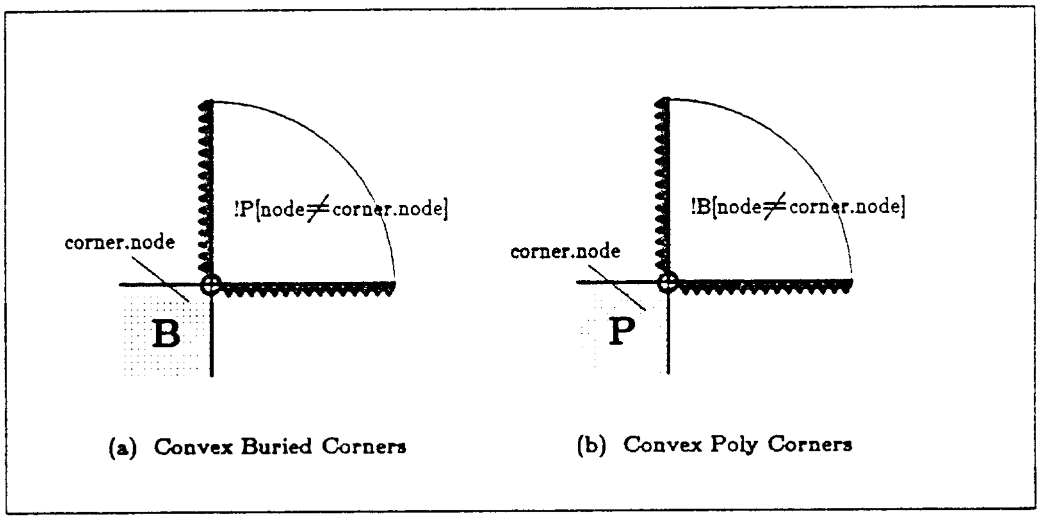


Figure 5.16. - Corner-Based Buried to Unrelated Polysilicon Spacing. The buried-to-unrelated-polysilicon rule is just like an interlayer spacing rule, except that the layer specifications in the condition sectors are qualified to refer only to distinct nodes. Part (a) checks outside convex buried corners for polysilicon belonging to a different node (i.e. unrelated) and part (b) checks outside convex polysilicon corners for any buried regions with a different node number.

rule is to be checked, power and ground buses must be identified in the mask data with appropriate tags, say '*function*="*PWRBUS*"'. In a region-based system such a rule would typically be checked by preceding the 5 unit width check with a selection operation that outputs only those metal regions with a function tag of "*PWRBUS*". In a corner-based system, such a rule can be verified with a width check on '*Metal*[*function*="*PWRBUS*"]'. This looks as follows:

```
rule "Power Bus Width"
  for Metal[function="PWRBUS"] corners_require
    if corner.angle<180 then
      Metal[function="PWRBUS"] everywhere_in
          sector[edge1−90,edge0+90,5]

  for Metal[function="PWRBUS"] corners_require
    if corner.angle>180 then
      Metal[function="PWRBUS"] everywhere_in
          sector[edge0+90*,edge1−90*,5]
```

This check is illustrated in Figure 5.17.

5.5. Summary

This chapter has introduced a corner-based mechanism for describing and checking design rules. The mechanism is summarized by the following four features:

i. Sector conditions are attached to corners of given angles and layer combinations.

ii. Sector conditions consist of circular sectors within which layer combinations must be present (or absent).

iii. Conditions can be logically combined to implement conditional rules.

iv. Layers can be qualified by reference to attributes tagged to the mask data.

The main idea is to express rules in terms of conditions that must hold at corners in a design. Conditions take the form of circular sectors that are checked for the presence or absence of certain layer combinations. Conditional rules are checked by logically combining multiple sector conditions. Rules that depend on nongeometric information are checked by referencing

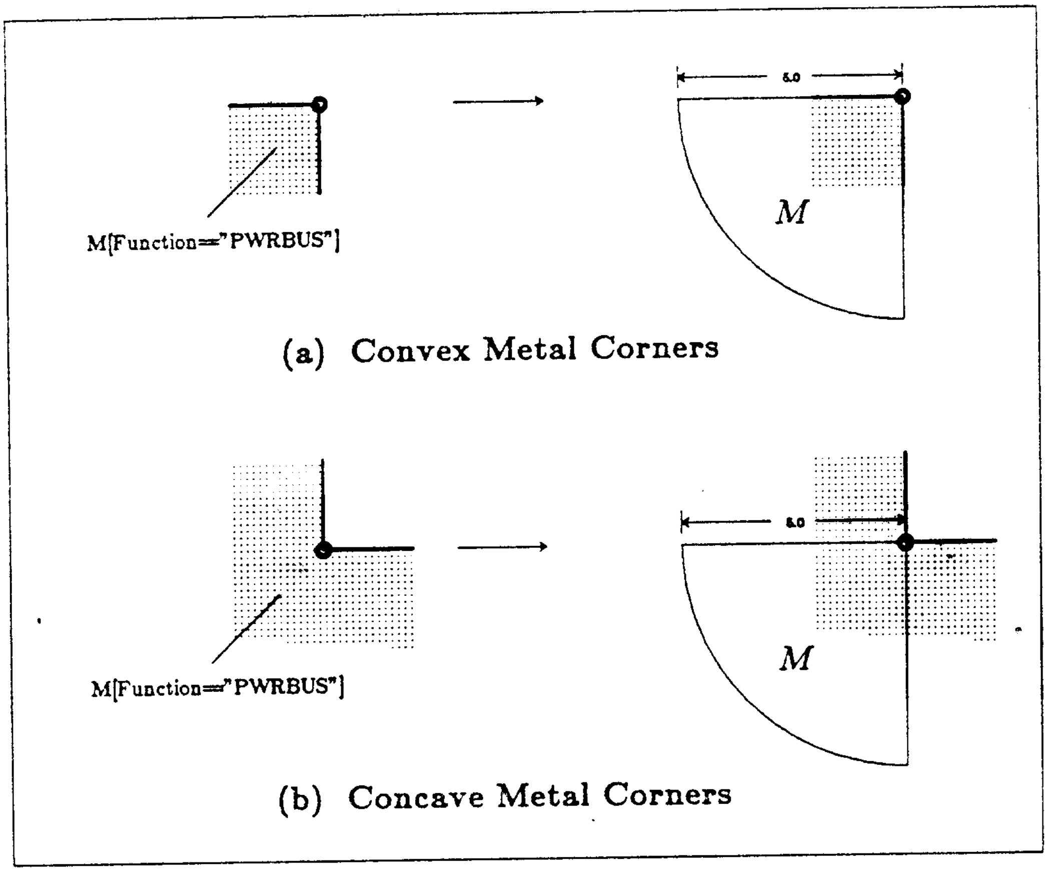


Figure 5.17. - Corner-Based Bus Width Check. Metal buses may need to be wider than other metal, since they carry a great deal of current. If buses are tagged as such, such a rule can be checked with condition sectors attached to corners of metal regions tagged as buses. Both convex and concave corners need to be checked, just as in a standard, unconditional, width check.

data attached to the mask features in layer specifications.

The first two features suffice for checking unconditional spacing, width, and enclosure rules (Figure 5.9). Spacing is checked with outward facing sectors that guard against intrusions of the same or a second layer. Width is checked with inward facing sectors that make sure a layer is present for some minimum distance inside of corners, and enclosure is checked with outward facing sectors on convex corners of the enclosed layer, and inward facing sectors on concave corners of the enclosing layers. The corner-based mechanism effectively replaces the sequences of boolean and tolerance operations required in region-based systems for checking unconditional rules. Preceding boolean operations are not required since both corners and sector conditions can involve layer combinations. The edge to edge tolerance checking of the region-based approach is replaced by the checking of circular sectors at corners: point to edge tolerance checking. This is possible since the closest approach between two edges always occurs at a corner. However, unlike edge/edge checking, point/edge tolerance checks must be done from (corners on) both of the involved layers.

Geometric conditionals can be implemented using the third feature of the corner-based mechanism: the logical combination of multiple conditions. Interdependent conditions at corners are particularly well suited for rules involving directional context. For example, a sector condition along the right edge of a corner may only be checked if another condition along the left edge holds. Such directionally sensitive rules, common in rules governing the formation of constructs such as transistors, are difficult to check in region-based systems. The context required to check geometric conditionals is not always available at the corners in the design, e.g. reflection rules (Figure 5.13). Corner-based checks of such rules must be preceded by region-based operations to establish the required context. In such cases, the use of the corner-based mechanism still reduces the total number of operations required.

Rules involving nongeometric information require the fourth feature of the mechanism: reference to nongeometric information tagged to mask features. Qualification of layer specifications with reference to tagged information accomplishes the same function as special selection operations and subroutines in region-operation systems. However the corner-based mechanism cannot be used to generate nongeometric information. Tagged information must be provided prior to the corner-based check, e.g., by region-operation-based analysis programs.

In corner-based systems macros are provided for common checks, such as spacing, width, and enclosure. These macros all expand into the same primitive formalism of the corner-based mechanism. More unusual checks can be written directly in terms of the underlying mechanism. Unlike the region operation approach, special primitives are not required for rule variants or unusual checks.

The use of point/edge tolerance checking makes the corner-based mechanism particularly well suited for partitioned checking, as required in incremental and hierarchical systems. Checking a piece of a design simply involves checking the corner-points in that piece: the tricky problem of edges straddling piece boundaries that is encountered in edge/edge checking is avoided.

Rather than depending on sequences of operations, corner-based checking is explicitly context-based: each rule consists of conditions that apply in a certain context, i.e., at certain corners. This permits all all rules to be checked in parallel in one pass through a design. The many intermediate layers, and operations of the region-operation approach, and the resulting I/O bottleneck are avoided.

CHAPTER 6

Implementation of Corner-Based Checking

6.1. Introduction

This chapter discusses the implementation of corner-based design rule checking with all the capabilites discussed in the previous chapter. To date, all existing systems are less general than this in one respect or another. Actual systems will be surveyed in the next chapter.

Corner-based checking involves finding corners in a design and verifiying the relevant (possibly interrelated) conditions the design rules associate with them. The efficiency of this process depends, to a large extent, on the internal representation used for the design rules. The main theme of this chapter is this internal representation: what it looks like, how it is used by the checking algorithms, and its automatic generation, by a rule compiler, from the human-readable and -writable format presented in the last chapter.

The three key issues of this chapter are:

i. Representation of layer expressions, which specify combinations of mask layers - to allow quick evaluation.

ii. Indexing of Rules - to allow quick access to relevant ones.

iii. Representation of condition expressions, which involve interrelated conditions - for efficient evaluation.

The evaluation of layer expressions is required both during the identification of corners and during the verification of sector conditions. *Many* expression evaluations are performed in the course of corner-based design rule checking. Leo, a commercial corner-based DRC, averages over 4000 expression evaluations per transistor; see Chapter 9. Thus efficient corner-based checking depends on a representation for layer expressions that permits quick evaluation. A bitmapped representation, based on the disjunctive normal form for boolean expressions, is developed for this purpose.

Rulesets contains many rules, only a few of which apply at any given vertex. For example the Mead-Conway nMOS rules for Lyra consist of 44 rules, of which, on the average, less than 5 apply at a given corner. Efficient rule checking depends on quickly eliminating most of the irrelevant rules from consideration at any vertex. An indexing scheme based on the layer expressions is used for this.

Corner-based systems spend much of their time evaluating condition expressions, (30% in Lyra, 55% in Leo, and 75% in Leo45). The evaluation of some conditions is relatively expensive. Thus the intelligent handling of condition expressions, evaluating cheap conditions first and avoiding whenever possible the evaluation of expensive ones, can significantly speed up design rule checking. A decision-tree representation is developed that completely specifies the order of condition evaluation. The rule compiler uses heuristics to choose an appropriate evaluation order when constructing these trees.

The body of this chapter contains three sections. The first section, below, develops an internal rule representation. The second section discusses rule-checking algorithms employing this representation. And the third section considers the generation of the internal representation by the rule compiler.

Chapter 9 contains data from actual systems relating to several issues disscussed in this chapter. These include data organization, layer-expression evaluation, condition-expression processing, and rule indexing.

6.2. Internal Rule Representation

Corner-based rules, as developed in the last chapter, specify *conditions* that apply to *corners* on certain *layers*. Conditions, corners, and layers are precisely defined in this section, and their internal representation is considered. Disjunctive normal form and decision trees are developed as means of representing layer and condition expressions (respectively). The end result is an internal representation for rules that is suitable for efficient checking. This representation is quite different from the external format, which is intended for convenient reading and writing by humans. The rule compiler translates external rule descriptions to the internal format.

6.2.1. Disjunctive Normal Form

In corner-based rules, general logical expressions are used for three purposes: to describe the combination of mask layers comprising the *layer* of a corner, to describe the combination of mask layers that must be present within a sector, and to describe an interrelated set of conditions applying at a corner. Here are examples of each kind (respectively):

for $(D\ \&\ !P)$ **corners_require** $\cdots$

$(P \mid D)$ **everywhere_in sector** $\cdots$

if *CONDITION*1 **and** *CONDITION*2 **then** *CONDITION*3

The first two types are layer expressions: they involve mask layer combinations. Layer expressions are conveniently represented in the disjunctive normal form, (DNF), [Hohn 1966] described below. The third type is a condition expression: it describes an interrelationship between conditions. Disjunctive normal form is used as an intermediate representation for condition expressions during rule compilation.

An expression is in disjunctive normal form, if it is the *or* of terms, that in turn are the *and* of simple variables and their complements. For example the expression

$!(A\ \&\ !(B \mid C))\ \&\ D$

is not in disjunctive normal form, but the equivalent expression

$!A\&D \mid B\&D \mid C\&D$

is. Every logical expression, no matter how complex, can be put in disjunctive normal form.

The next section shows how disjunctive normal form is employed to represent layer expressions. A method for converting expressions to disjunctive normal is given in the section on the rule compiler.

6.2.2. Layer Expressions

Layer expressions must be evaluated in the search for corners to which the rules apply, and in checking the sector conditions that apply at these corners. Consequently many layer expression evaluations are required in the course of a design rule check. For example a complete design rule check of the 44,000 transistor RISC-I chip with Leo requires approximately 200 million layer expression evaluations.

To permit fast evaluation, layer expressions are represented in disjunctive normal form, using bitmaps; this is illustrated in Figure 6.1 Each mask layer is assigned a bit position in the maps. The presence of variables in positive form are marked by 1's in the *posMap* for the term, and the presence of negated variables are marked by 0's in the *negMap* for the term. An expression is represented by *posMap* and *negMap* arrays, with one entry in each array for each term.

This bitmapped representation allows advantage to be taken of parallel bitwise logical operations. A term in a layer expression can be evaluated as follows:

```
((posMask[i] & layers) == posMask[i]) ||
(( negMask[i] | layers) == negmask[i]).
```

The C-language syntax is used: '&' and '|' are bitwise *and* and *or* operations (respectively), '==', compares for equality, and '||' is logical *or*. The variable '*layers*' is assumed to be a bitmapped representation of the mask layers present where the expression is to be evaluated. This computation requires only a few machine instructions. Since layer expressions generally contain only 1 term, and almost never more than 3, they are evaluated quite quickly using this method. (Leo and Leo45 average 22 microseconds per expression evaluation, using this method.)

Some expressions are complicated by layers that are qualified with attribute information. Two examples of such expressions (drawn from the rules in Chapter 5) are:

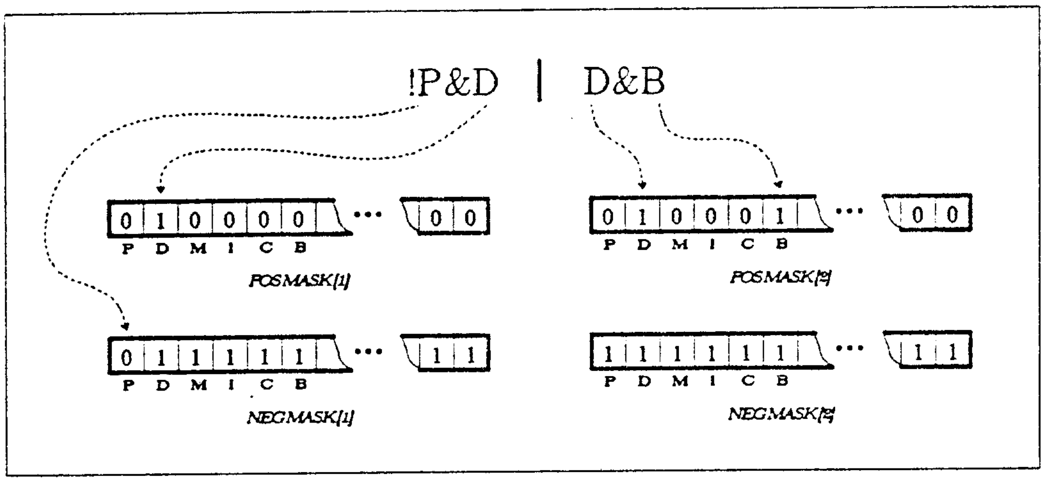


Figure 6.1. - Bitmapped Representation of Logical Expressions. After conversion to disjunctive normal form, logical expressions can be represented using pairs of mask words (one pair per term). Positive variables are indicated by 1's in the *PosMask*'s, and negated variables are indicated by 0's in the *NegMask*'s. Expressions represented in this way can be quickly evaluated, using machine level bit-parallel logic operations.

!*Polysilicon* [*node*=*corner.node*]
Metal [*function* =*PWRBUS*]

Qualified mask layers are assigned their own bit positions in the maps. The evaluation of such expressions is slowed by the need to reference the attribute information and establish the presence or absence of the qualified layers prior to the bitmap computations given above.

6.2.3. Corners

The *corners* of corner-based checking consist of a *vertex*, two *edges* and a *layer* (expression) that is present between the edges but not directly outside them. It is important to note that both edges and layer are necessary to the definition of a corner; see Figure 6.2. Corners with distinct edges may share a common vertex, and corners on more than one layer may share common edges. Each of these corners may have distinct conditions associated with it that must be checked.

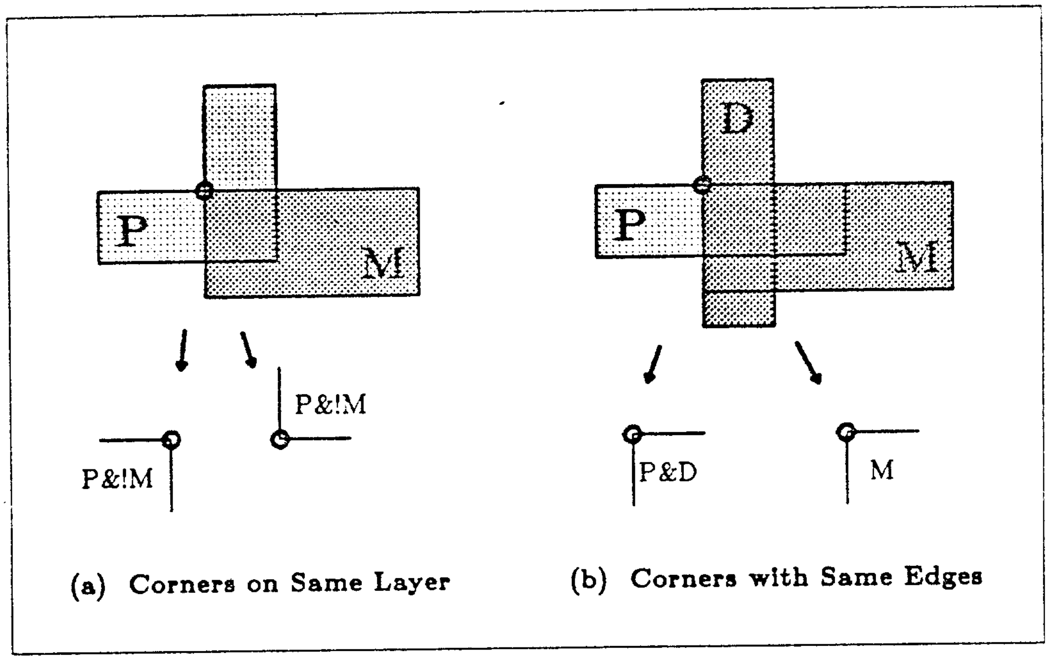


Figure 6.2. - Corners. Corners are defined by their *vertex*, two *edges*, and a *layer* that is present between the edges. The *edges* and *layer* are both important. Two corners on a common layer may share a vertex (a). Similarly two corners can share common edges (b). Each distinct corner may entail its own associated conditions that must be checked.

6.2.4. Conditions

Conditions, associated with corners by the design rules, must be verified during checking. Much of the processing time of corner-based systems is devoted to condition processing - 30% in Lyra, 55% in Leo, and 75% in the more complicated Leo45. There are four types of conditions, as illustrated in Figure 6.3.

The *sector condition*, shown in Figure 6.3(a), is the basic type. As explained in Chapter 5, a sector condition is specified by giving its two edges (relative to the corner edges), a depth, and a layer expression. The condition holds if the layer expression holds throughout the interior of the sector. The sector edges may be inclusive or exclusive. If a sector edge is inclusive, it is considered interior to the sector condition, and hence regions abutting the edge will be considered to infringe upon the sector. Zero-width sectors can be specified to check a layer expression outward from the corner in a particular direction (Figure 6.3(b)), and zero-depth sectors can be specified, to check for the presence of layers directly adjacent to the corner (Figure 6.3(c)).

Angle conditions, the remaining type, specify a range of angles for a corner (Figure 6.3(d)). They hold only at corners whose angles lie within the specified range. Angle conditions are generally used to distinguish between convex and concave corners, but they can also be used for other purposes, for example to disallow acute corners, (i.e., those sharper than 90 degrees), on mask layers.

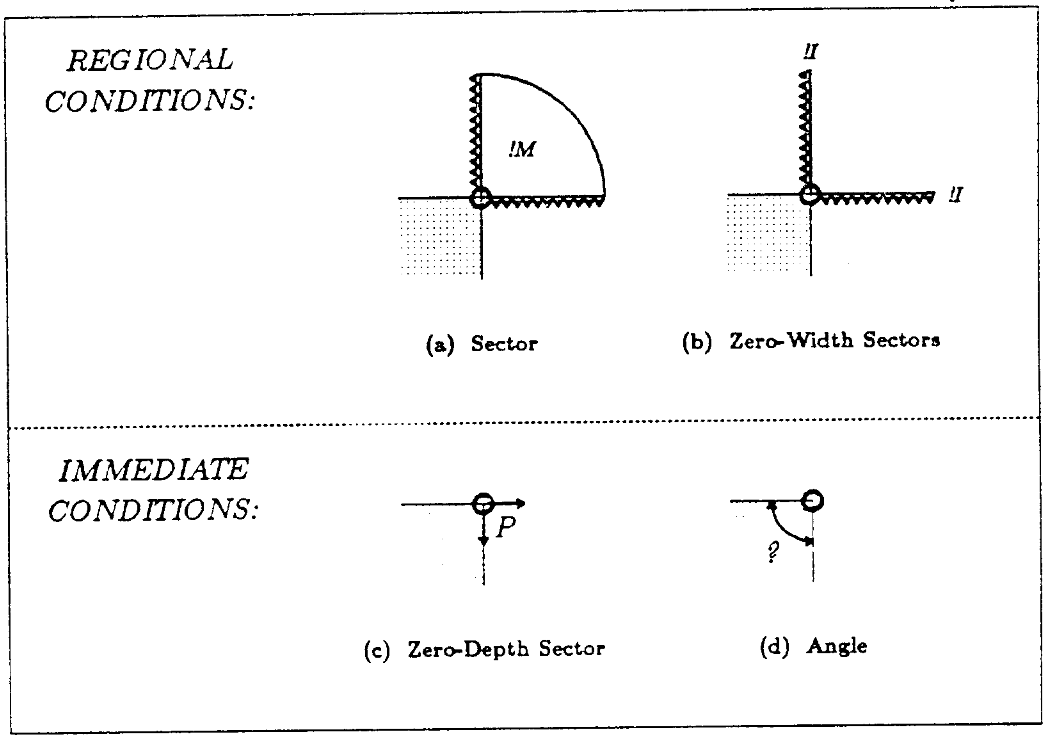


Figure 6.3. - Conditions. Conditions are of four types: sector conditions, which require a layer to be present in a circular sector about a corner, (a), zero-width sector conditions, which require a layer to be present outwards in a given direction from a corner, (b), zero-depth sectors, which require a layer to be present directly adjacent to a corner, (c), and angle conditions, which require the angle between the corner edges to fall within a certain range, (d). Conditions are classified as immediate, if they depend only on the mask configuration, directly at a vertex, and as regional, if they depend on additional mask data in the vicinity of the corner as well. Immediate conditions are cheaper to evaluate than regional ones.

Conditions are divided into *immediate* ones (angle conditions and zero depth sector conditions) and *regional* ones (all other sector conditions). This distinction is important because immediate conditions can be verified much more quickly than regional ones.

6.2.5. Condition Expressions

In general, a rule specifies *interrelated* conditions, i.e., *condition expressions*. For example, consider part (c) of the transistor extension check given in Chapter 5:

```
/* (c) Check extension dimensions at normal gate corners. */
for (P & D) corners_require
   if corner.angle < 180 then
      if (P everywhere_in sector[edge0*,edge0,0] and
     D everywhere_in sector[edge1,edge1*,0]) then
        P everywhere_in sector[edge0−90,edge0−90*,2] and
        D everywhere_in sector[edge1+90*,edge1+90,2]
      elseif (D everywhere_in sector[edge0*,edge0,0] and
     P everywhere_in sector[edge1,edge1*,0]) then
        D everywhere_in sector[edge0−90,edge0−90*,2] and
        P everywhere_in sector[edge1+90*,edge1+90,2]
```

Here a set of nine interrelated conditions is specified for transistor corners. If the details of the conditions themselves are ignored, simply writing $I1$ through $I5$ for the immediate conditions, and $R1$ through $R4$, for the regional ones, the expression becomes:

```
/* (c) Check extension dimensions at normal gate corners. */
for (P & D) corners_require
   if I1 then
      if (I2 and I3) then
      R1 and R2
      elseif (I4 and I5) then
        R3 and R4
```

Note that it is not necessary to evaluate all the conditions in the expression. If $I1$ (the angle condition) is evaluated first, and is found not to hold, none of the other conditions need be evaluated. Further if all the immediate conditions are evaluated first, it will not be necessary to evaluate more than two of the four regional conditions. Evaluating conditions in the appropriate order, and avoiding the evaluation of conditions that have been rendered irrelevant by previous ones, significantly reduces the total amount of computation. The representation of condition expressions as decision trees, allows the rule compiler to assign an optimal order of condition evaluations, and avoids the evaluation of irrelevant conditions.

Figure 6.4 gives a decision tree for the expression above. The internal nodes of the tree give conditions to evaluate, and the leaves of the tree specify whether the entire expression evaluates to *true* or *false*. The root condition is evaluated first. Computation then proceeds along the T or F branch of the tree, according to how the condition evaluates. When a leaf is

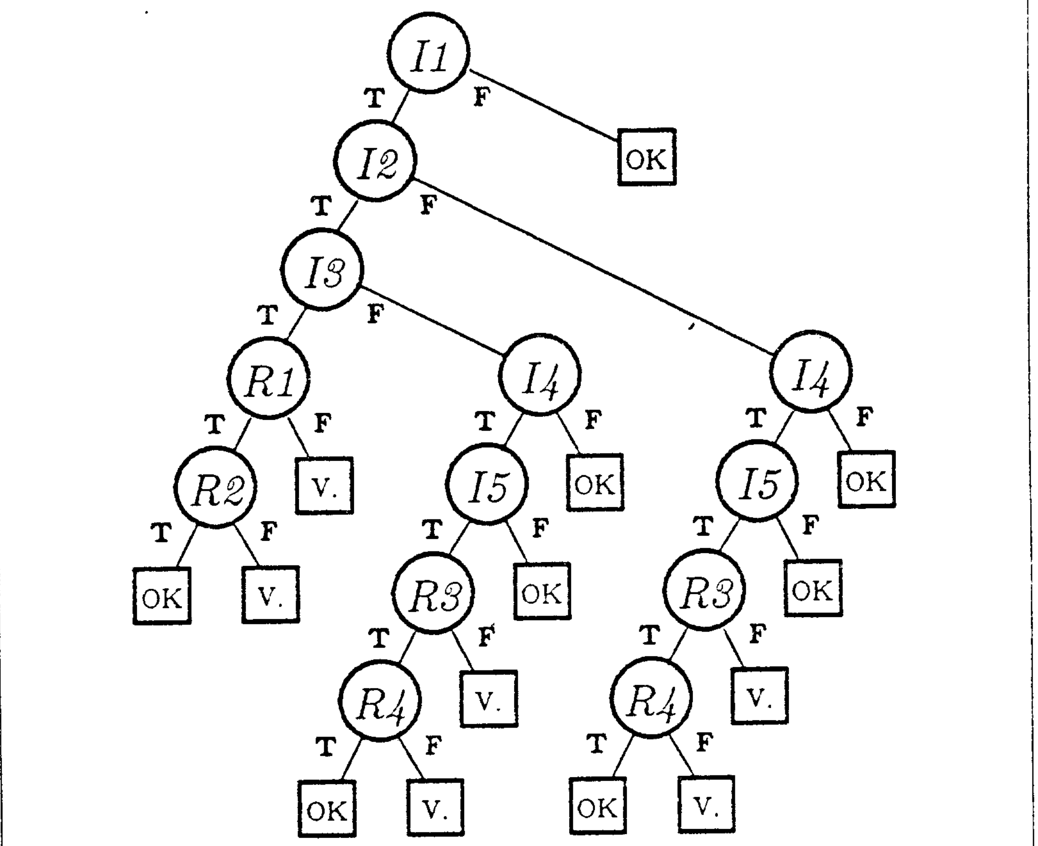


Figure 6.4. - Decision Tree for Condition Expression. The decision tree representation of a condition expression completely specifies the order of condition evaluation. The condition at the root of the tree is verified first; evaluation then proceeds down the T or F subtree according to the result. When a leaf node is reached the computation is complete; the leaf indicates whether the expression is satisfied or not. The use of decision trees minimizes the number of conditions that must be verified during expression evaluation.

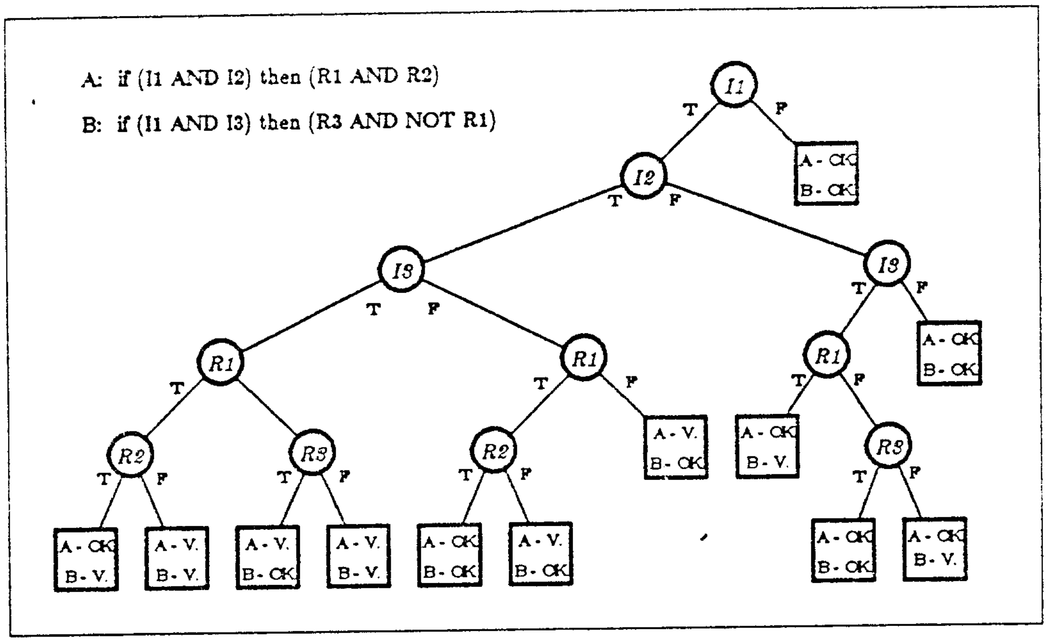

A: if (I1 AND I2) then (R1 AND R2)
B: if (I1 AND I3) then (R3 AND NOT R1)
I1
I2
I3
R1
R2
R3
T
F
A - OK
B - OK
A - V.
B - V.

Figure 6.5. - Multiple Expression Decision Tree. This decision tree allows the expressions *A* and *B*, (at top left), to be simultaneously evaluated. Such simultaneous evaluation avoids redundant verification of conditions that are common to expressions, (in this case *I1* and *R1*).

reached, the computation is complete. The value indicated in the leaf is the value of the expression. Putting the angle condition at the root of the tree eliminates the need for any further expression evaluation whenever it is not met. The location of the immediate expressions toward the root of the tree minimizes the number of regional conditions that need to be evaluated. The automatic generation of optimal decision trees is taken up in the section on the rule compiler below.

Decision trees are indexed under the layer of the corners they apply to. If expressions applying to corners on the same layer share common conditions, redundant evaluation of these conditions can be avoided by combining them into a single decision tree. This is illustrated in Figure 6.5. The leaves of the combined tree indicate which expressions (i.e. rules) have been violated.

6.3. Checking Algorithm

Rule checking consists of two steps:

i. Detecting corners.

ii. Checking the conditions that apply there.

This section considers how these two steps are accomplished.

A *sorted* mask-data representation that allows quick access to the mask regions and mask region boundaries is assumed. A pixel-based representation, for example would not be suitable, since it does not allow quick access to region boundaries. It is also assumed that regions are split where attributes change, so that edges are always present along the transitions; see Figure 6.6. As long as these conditions are met, the details of the mask data representation are not important to the overall checking algorithm: two-dimensional bin, sorted swath, scanline, and corner-stitched representations are all suitable. Several of these have been employed in corner-based systems; see Chapter 7.

6.3.1. Corner Detection

Corner vertices occur at boundary edge crossings; see Figure 6.7. Thus they can be found by searching for intersections between pairs of edges. The use of sorted mask data allows the search to be limited to edges that are near each other, thus making it efficient.

Recall that corners are determined not only by a vertex, but also involve a layer and edges. The second step in corner detection is to construct a *pie-slice* data structure, (Figure 6.8), giving the position of edges around a vertex and the mask layers present in the pie-slices between the edges. In the simplest case there will be only two pie-slices (e.g. the interior and exterior of a metal corner as in Figure 6.8(a)), and in almost all cases there will be no more than four slices.

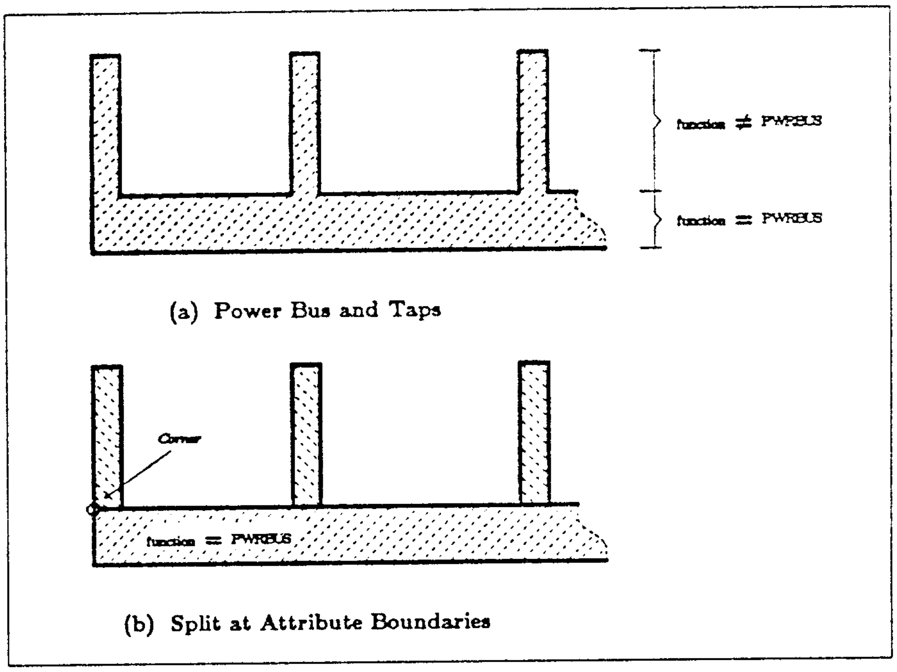


Figure 6.6. - Splits at Attribute Boundaries. Attributes may only apply to sections of a mask region. For example, the major trunks of the power and ground network may be tagged as buses, while the smaller taps into these trunks are not, (a). It is assumed that the mask regions are split at attribute boundaries, so that attribute values are constant throughout the individual figures in the mask data, (b). This ensures that corners on attribute-qualifed layers will not be missed. For example the *Metal[function="PWRBUS"]* corner indicated in (b) will be detected.

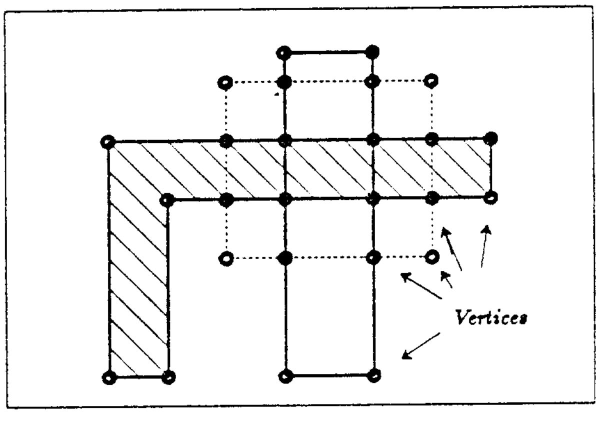


Figure 6.7. - Corner Vertices. Corner vertices occur where one or more edges cross in the mask data. The first step in corner-detection is to search for these edge crossings.

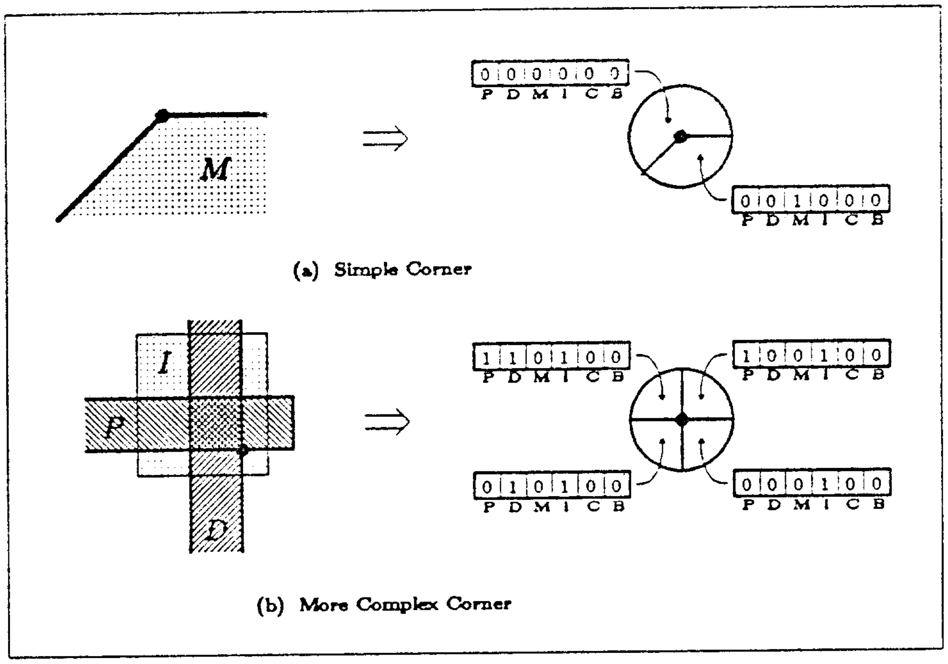


Figure 6.8. - Pie-Slice Data Structure. After a vertex is detected, a pie-slice data structure is created for it. The pie-slice data structure gives the position of all edges about the vertex and, in bitmapped form, the mask layers present between the edges. At simple corners, (a), only two pie-slices are present. However, since in general many edges may meet at a common point, many pie-slices are possible.

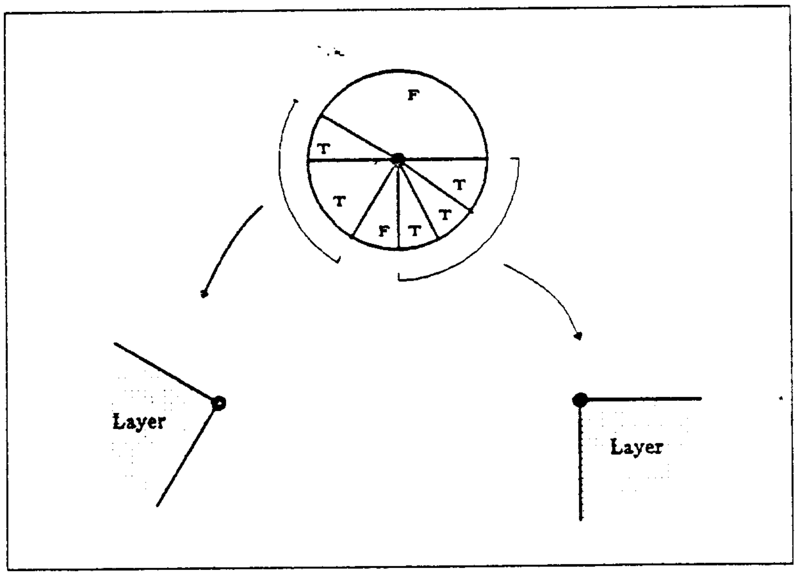


Figure 6.9. - Extracting Corners from Pie Slice Data. To find corners on a particular layer, the corresponding layer expression is evaluated at each pie-slice about a vertex, and then contiguous slices where the expression holds are consolidated.

However in general, several edges may intersect a vertex, and thus a large number of pie-slices is possible. The mask layers present in each pie-slice are represented in bitmapped form, so that layer expressions can be evaluated on them. The pie-slice data structure is constructed incrementally, by considering the effect of each mask region present at the vertex. Efficiency depends, again, on the utilization of sorted mask data.

Corners on specific layers can be extracted from the pie-slice data structure by evaluating the layer expression on each pie-slice and then consolidating contiguous sectors where the layer expression is satisfied; see Figure 6.9. This computation is relatively expensive, and in a typical rules set conditions are associated with 15 or so layers of which only 1 or 2 apply at each vertex; see Appendix I. Thus checking for the presence of each layer at each vertex is too slow. This is circumvented by an indexing scheme that greatly reduces the number of layers considered at each vertex.

The idea behind the indexing is simple: there is no point in looking for *Metal* corners at a vertex where *Metal* is not present in any pie-slice. Similarly corners on more complex layers can only occur when certain mask layers are present; for example a corner on '*(Polysilicon | !Diffusion) & Buried*' can only occur when *Buried* is present. Such facts are exploited by the rule compiler to build a *relevant-layer* table that gives, for each combination of mask-layers, an associated set of relevant layers. The relevant-layers are those with associated conditions defined in the ruleset, that in addition may occur at a corner where the given mask layers are present.

If there are 10 mask layers (a typical number), the relevant-layer table would have 1024 entries. After the pie-slice data structure for a vertex has been constructed, the bitmaps giving the layers present in each pie-slice are *ored* together to obtain a single map giving all the layers present at the corner. The entry in the relevant-layer table corresponding to this value is consulted to obtain the relevant layers for this vertex. Then the pie-slice data structure is searched for corners on each of these layers by evaluating the layer expression on the pie-slices and consolidating slices, as already described.

6.3.2. Condition Evaluation

Once a corner has been completely identified, vertex, edges, and layer, it is time to check the conditions that apply to it. One or more decision trees is associated with the layer of the corner. Each decision tree represents one or more condition expressions. The trees must be traversed from the root down, evaluating the root condition first, and then proceeding with the *T* or *F* subtree, according to how the condition evaluates.

The immediate conditions, i.e., angle conditions and zero-depth sector conditions, are readily evaluated: their value can be determined from the location of the corner edges, and from the pie-slice data structure for the corner's vertex.

Regional conditions, i.e., sector conditions with positive depth, are more difficult to evaluate. To evaluate them one must determine whether a layer expression holds throughout the interior of the sectors. This is accomplished by fracturing sectors at each mask region boundary, dividing them into "monochromatic" chunks, each containing a definite combination of mask layers throughout, and then evaluating the layer expression on each chunk separately. See Figure 6.10.

For totally-sorted mask representations, such as corner-stitching and scanline, the fracture lines are available so no special computation is required to do the splitting. If a partially-sorted mask representation is used, such as binning or sorted swaths, a clipping procedure must be employed to break the sector into the appropriate chunks. Though in the worst case this sector splitting process can be computationally intensive, on the average it is not unreasonable. The splitting need only involve mask layers occuring in the layer expression. For example a sector checking for the presence or absence of metal need only be split at metal boundaries. Consequently most sectors will not be split at all unless a violation is present.

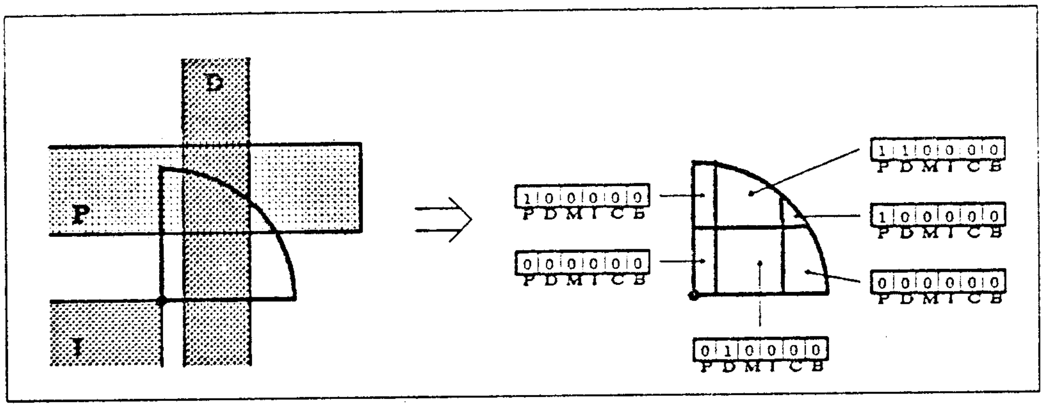


Figure 6.10. - Sector Condition Evaluation. Sector conditions are evaluated, by splitting their interior into "monochromatic" chunks, each with a definite combination of mask layers present, and then evaluating the layer expression for the condition on each chunk. This process is not as costly as it at first appears. The splitting need only be done for mask layers involved in the layer specified by the condition. In practice most sectors need not be split (they are monochromatic with respect to the relevant layers).

Since true circular sectors are difficult to work with, e.g., to split into monochromatic pieces as above, polygonal approximations are used in practice; see Figure 6.11. The greater the required accuracy (in diagonal tolerance checks), the more complex this polygonal approximation must be. A greater range of angles in the mask data also requires more elaborate sector approximations: sector approximations employing only 45 degree angles can be quite accurate if the mask data is similarly restricted to 45 degree angles. The cost of more elaborate approximations is considerable. The total time spent processing conditions in Leo45, which uses 45 degree approximations as described above, is 3.6 times that of Leo, its manhattan predecessor.

6.4. The Rule Compiler

The rule compiler converts the human readable and writable, external rule description to an internal form that permits efficient checking. Aspects of this task include:

i. Macro expansion

ii. Parsing of the input language

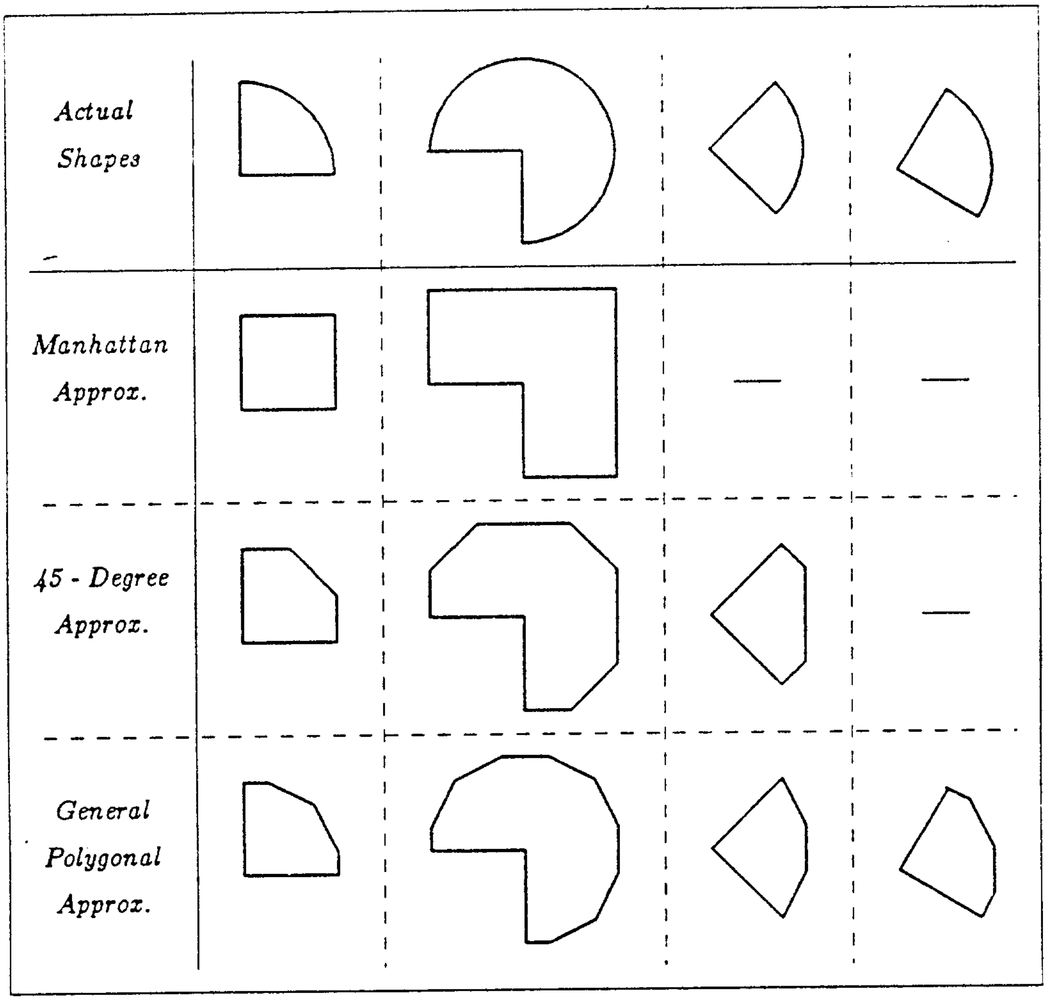


Figure 6.11. - Polygonal Sector Approximaltions. Polygonal approximations of sector regions are used during processing. The simplest approximations are the Manhattan ones. Though these result in excessively conservative diagonal tolerances, they have been successfully employed in Manhattan-only systems. The next simplest approximations, using only 45's, are quite satisfactory when the mask data is similarly restricted. General approximations are more accurate, but require more processing.

iii. Conversion of layer expressions to DNF

iv. Generation of the relevant-layers table

v. Conversion of condition expressions to (optimal) decision-trees

Macro preprocessers and parsers are well understood. They can be implemented, for example, with the Unix tools M4 [Kernighan & Ritchie 1977], Lex [Lesk 1975] and Yacc [Johnson 1975].

The remaining topics, conversion to DNF, generation of the relevant-layers table, and decision-tree construction, are more specific to rule compilation; they are considered below.

6.4.1. Conversion to Disjunctive Normal Form

Recall that conversion of layer expressions to disjunctive normal form permits an efficient bitmapped evaluation technique. It is also useful to convert condition expressions to DNF prior to generating decision trees. DNF facilitates the expression specialization and simplification required to create decision trees, and also permits heuristics based on the *term* structure of DNF; see the section on decision trees below.

Whether the primitives of the expression are mask layers or conditions, the process of conversion to DNF is the same. It consists of a sequence of transformations beginning with the parse tree for the original expression and ending with an equivalent tree in DNF form. The process is illustrated with the parse-tree for the expression in Section 2.5; see Figure 6.12(a). The goal is the OR/AND/NOT structure of Figure 6.12(d).

The first step is the elimination of *if* constructs, by the application of the transformations shown in Figure 6.13. The application of these transformations converts the tree of Figure 6.12(a) to that of 6.12(b). After this step all interior nodes are either **AND**, **OR**, or **NOT**.

The second step is to *push* **NOT**'s down to the leaves of the tree. This is accomplished by the transformations of Figure 6.14, which are based on DeMorgans laws and the law of double negatives. The transformations are applied to **NOT** nodes, working from the top of the tree down, until all remaining **NOT**'s are positioned just above the leaves. Applying these transformations to the tree in Figure 6.12(b) yields 6.12(c). After this transformation, only **AND**'s and **OR**'s are left in the top part of the tree.

The final step is to move the **AND**'s through the **OR**'s. The required transformation, based on the distributive law, is shown in Figure 6.15. Applying this transformation to the

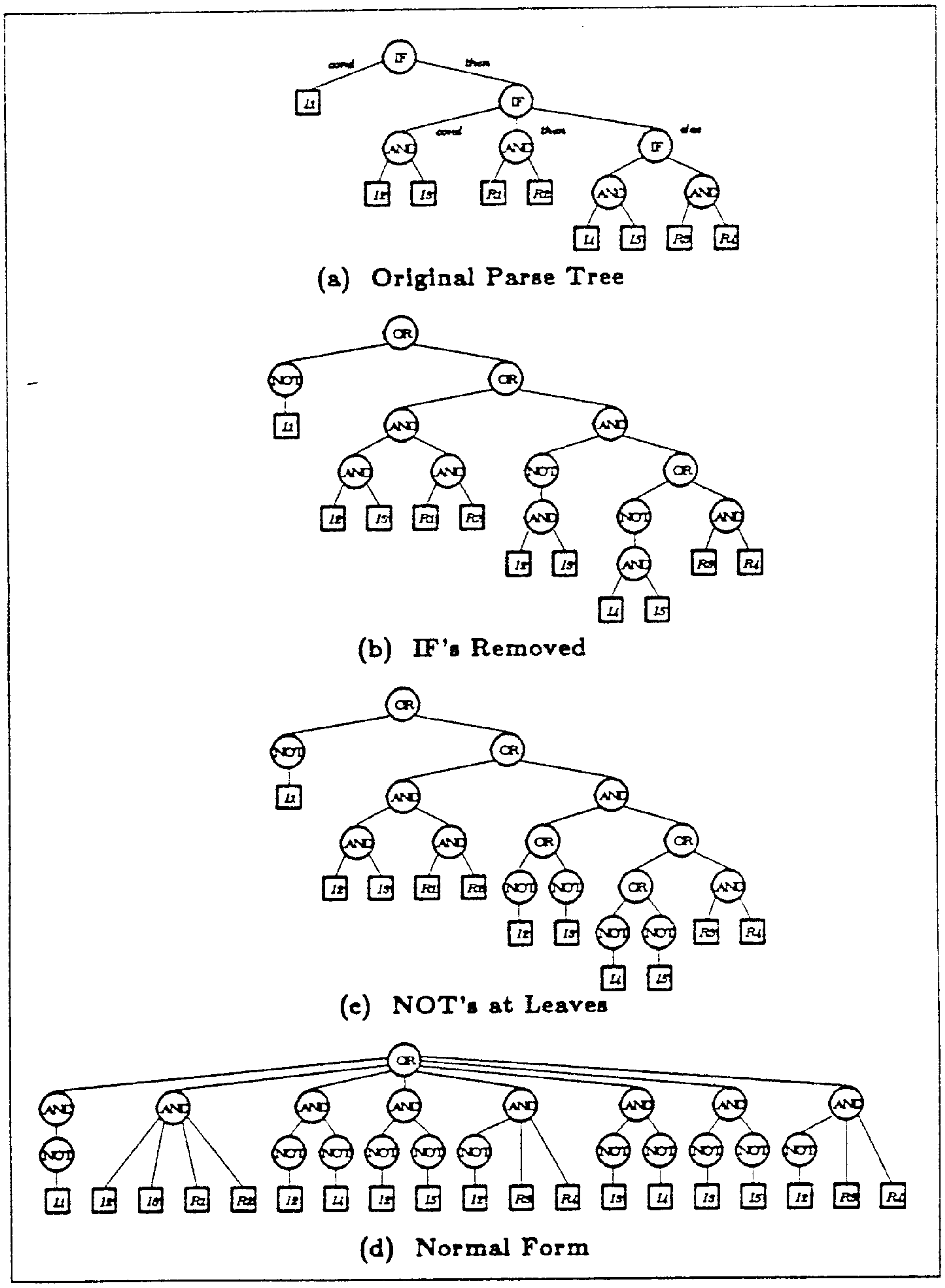


Figure 6.12. - Conversion to DNF. These are the stages in the conversion of an example parse-tree to DNF. Notice the stratification of **OR**, **AND**, and **NOT** nodes in the final, DNF, representation, (d).

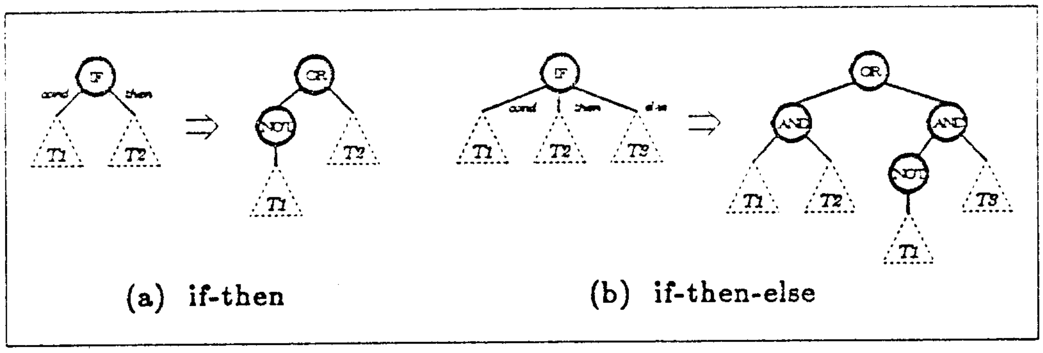


Figure 6.13. - Transformations eliminating IF's. These transformations replace if constructs with equivalent structures involving only AND, OR, and NOT.

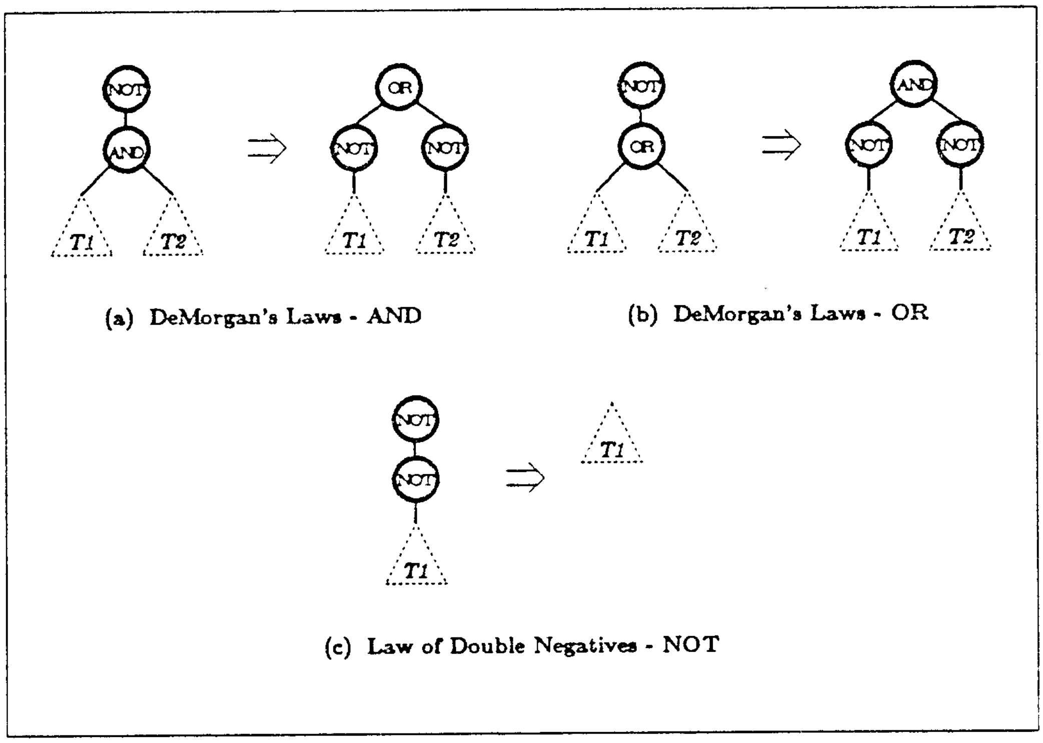


Figure 6.14. - Transformations Pushing NOT's to Leaves. A NOT is moved through an AND by (a), and through an OR by (b). Adjacent NOT's are collapsed by the transformation in (c). These transformations are applied systematically from the root of the tree down, leaving NOT's only directly above the leaves.

example, (Figure 6.12(c)), and consolidating adjacent AND's (and OR's) to single multi-argument functions, yields the final DNF tree shown in Figure 6.12(d).

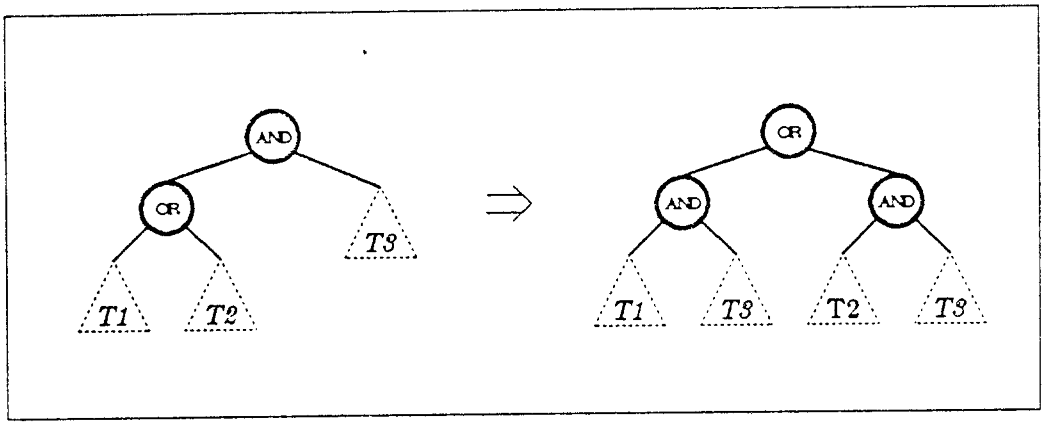


Figure 6.15. - Transformation moving AND's through OR's. Systematic application of this transformation from the root of a tree down, moves all the AND's below the OR's. This is the final step in conversion to disjunctive normal form.

DNF trees for layer expressions are translated to bitmapped form. DNF-trees for condition expressions are the input for the decision-tree generation algorithm.

6.4.2. Generation of Relevant-Layers Table

Recall that the purpose of the relevant-layers table is to limit the layers considered in searching for corners at vertices. The table has an entry for each combination of mask layers. Each entry consists of a list of layers for which corners are possible at a vertex where the given mask layers are present.

The table is created by considering, for each mask layer combination, the entire list of layer expressions associated with corners, and retaining those for which a corner might be present. However, simply evaluating the layer expressions for each set of mask layers does not work. For example suppose polysilicon and diffusion are both present at a vertex. Then the expression '*P&!D*' evaluates to *false*, yet, since diffusion need not be present in every pie-slice around the vertex, a corner on '*P&!D*' *is* possible. In general, since the presence of a layer somewhere at a vertex does not imply its presence everywhere, the value of negated variables in expressions can not be predicted. The trick is to modify the evaluation of expressions to *ignore* negated variables. Specifically, the computation:

```
((posMask[i] & layers) == posMask[i]) ||
((negMask[i] | layers) == negmask).
```

is replaced by simply

```
(posMask[i] & layers) == posMask[i]
```

If an expression evaluates to *true* under this modifed evaluation scheme, a corner on that layer is possible at vertices where the given mask layers are present.

6.4.3. Conversion to Decision-Tree Form

Condition expressions are put in decision-tree form to specify the exact order of evaluation and avoid unnecessary condition evaluations. A decision tree is built by choosing a condition for the root of the tree, and then proceeding (recursively) to build the T and F subtrees. This process involves choosing root nodes and deriving expressions for the subtrees. Both steps assume a DNF representation of the input expressions.

Huristics are used to choose, at each step, the root condition that is most likely to minimize the amount of computation required during evaluations of the expression. The following heuristics (approximately in order of priority) are the most important:

i. Prefer immediate conditions.

ii. Prefer conditions appearing in the most terms.

iii. Prefer conditions whose appearances in positive and negated forms are most nearly balanced.

Immediate conditions are chosen first because they are cheap, and their early evaluation is likely to reduce the number of more expensive region condition evaluations required. Conditions appearing in many terms are evaluated early, since they potentially greatly reduce the number of conditions left to evaluate. Between two conditions appearing in many terms, the one with the number of positive and negated appearances most nearly equal is chosen first, since, true or false, it is guaranteed to reduce the number of conditions left to evaluate. Notice that the second and third heuristics utilize the term structure of DNF.

Once a root condition is chosen, expressions for the *true* and *false* subtrees must be derived so that the process can be continued. The idea is to replace the root condition with *true*, ('**T**'), and *false*, ('F'), and simplify to obtain the expressions for the T and F subtrees (respectively). Since the expressions are in DNF form, the simplification can be managed as follows:

i. Cancel **T**'s appearing as positive variables, and **F**'s appearing as negated variables. If this reduces the number of variables in any term to zero, reduce the entire expression to **T**.

ii. Cancel terms where **F** appears as a positive variable or **T** appears as a negated variable. If this reduces the number of terms in an expression to zero, reduce the entire expression to **F**.

A third simplification is useful for removing redundant terms:

iii. If T_i and T_j are terms of the same expression, and term T_i contains all variables in term T_j, the negated variables in negated form, and the positive variables in positive form, then cancel term T_i from the expression.

These simplifications are illustrated in Figure 6.16. The end result of simplification is **T**, **F**, or an expression involving neither **T** nor **F**. If an expression simplifies to **T** or **F**, its value is completely determined, and no further conditions need be evaluated: a leaf node is created. Otherwise, a root condition is chosen for the subtree and the process is continued.

Recall that when two condition expressions for corners on a layer share common conditions, redundant evaluation of those conditions can be avoided by combining the expressions into a single decision tree. A decision tree for multiple expressions can be created using the same procedure as above, but simplifying each expression separately. A leaf is created when all the expressions have been reduced to **T** or **F**.

6.5. Summary

Corner-based checking involves the detection of corners in a design and the verification of the conditions specified for the corners by the design rules. These steps can be executed

(i)

$$A\bar{B}\cancel{T} + C\cancel{\bar{F}} \Rightarrow A\bar{B} + C$$

$$\cancel{T} + CE \Rightarrow \mathrm{T}$$

Term becomes null.

(ii)

$$\cancel{FA\bar{B}} + C\bar{D} + \cancel{\bar{T}E} \Rightarrow C\bar{D}$$

$$\cancel{\bar{T}A\bar{B}} \Rightarrow \mathrm{F}$$

Expression becomes null.

(iii)

$$\cancel{AB\bar{C}} + AB\bar{C}D + EF \Rightarrow AB\bar{C}D + EF$$

Subterm.

Figure 6.18. - Expression Simplification. DNF expressions containing **T**'s and **F**'s can be simplified as in (i) and (ii). Such simplifications may lead to redundant terms or subterms. Such terms should be eliminated as illustrated in (iii). After these simpilifications, an expression will either be **T** or **F**, or will not contain **T** and **F**.

simply and efficiently if a good internal rule representation is used. In particular the bit-mapped DNF representation of layer expressions speeds up both steps. It allows machine-level, bit-parallel, logic operations to be used for quick evaluation of layer expression, facilitating both the detection of corners, which occur on *layers*, and the verification of sector conditions, which requre *layers* to be present or absent throughout a sector's interior.

The indexing of rules is also important: a typical ruleset contains many rules, only a few of which apply at any given vertex. The relevant-layer table is used for indexing. Given the combination of mask layers surrounding a vertex, the table is consulted for a list of layers on which corners (with associated rules) might be present. Only corners on these layers, and hence the rules associated with them, are considered at a vertex. The rule compiler computes the relevant-layer table, using a modified evaluation scheme for layer expressions that ignores negated varaibles. Layers that evaluate to *true* under this scheme may be present at the vertex in question.

The representation of condition expressions is another important aspect of rule representation. The order in which conditions are evaluated can effect the total cost of the computation, because the evaluation of one condition will in some cases eliminate the need to evaluate another. Condition expressions are represented by decision trees that specify the exact order of condition evaluation. The rule compiler employs heuristics to choose a good order of evaluation when constructing decision trees. For example, cheap conditions are evaluated early in the hope of eliminating the need to check more expensive ones, and conditions that appear in multiple terms of an expression are checked early since they can potentially eliminate the need to evaluate many other conditions.

Because of the importance of a good internal rule representation, the rule compiler is a crucial component of corner-based implementations. The use of a rule compiler to translate rules to an efficient internal format makes corner-based design rule checking simple and fast

6.6. References

Normal forms and logic expression manipulation are discussed in [Hohn 1966]. Compiler writing tools, for automatic generation of lexical analyzers and parsers are presented in [Lesk 1975] and [Johnson 1975]. A general macro preprocessor, suitable for use in corner-based DRC systems, is detailed in [Kernighan & Ritchie 1977].

[Hohn 1966]
F.E. Hohn, *Appplied Boolean Algebra*, The Macmillan Company, New York, 1966, pp. 41-51.

[Johnson 1975]
S.C. Johnson, "Yacc: Yet Another Compiler Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975. Reprinted in *UNIX Programmer's Manual, Supplementary Documents,* 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California, Berkeley, CA, March, 1984.

[Kernighan & Ritchie 1977]

B.W. Kernighan, and D.M Ritchie. "The M4 Macro Processor," Comp. Sci. Tech. Rep., Bell Labs, Murray Hill, New Jersey, 1977. Reprinted in *UNIX Programmer's Manual, Supplementary Documents,* 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California, Berkeley, CA, March, 1984.

[Lesk 1975]

M.E. Lesk, "Lex - A Lexical Analyzer Generatior," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975. Reprinted in *UNIX Programmer's Manual, Supplementary Documents,* 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California, Berkeley, CA, March, 1984.

CHAPTER 7

Survey of Corner-Based Systems

7.1. Introduction

This chapter considers actual corner-based systems. None of these systems implement the corner-based formalism of the previous chapters in complete generality, though the combined features of two of them, Mart and Leo45, come close. For each system, the nature of its features and restrictions are presented, its implementation is discussed, and its rule checking capabilities are analyzed. The structure of the systems and their implementations is similar to that of the general system developed in the previous two chapters. Differences are for the most part due to restrictions in the actual systems that simplify implementation and limit rule checking capability.

The Magic and Intel systems presented at the end of the chapter do not use the corner-based formalism of the previous two chapters. These systems are included because they independently employ the two key innovations of the corner-based approach: The Magic system uses context-based checking, and the Intel system uses point/edge comparisons to implement its tolerance checks.

Performance figures and other numerical data for all the systems discussed in this chapter are given in Chapter 9.

7.2. Lyra

Lyra, the first corner-based design rule checker, [Arnold & Ousterhout 1982], was coded by the author in the summer of 1981. Lyra was written as an experiment to test ideas for corner-based checking, and also to fill a need for an accurate and flexible DRC at Berkeley. Lyra was completed just in time to check the RISC-I chip. Later Lyra was extended to check designs **hierarchically** (see Chapter 8) and added to the Berkeley VLSI Tools distribution tape. An editor interface allows Lyra to be invoked interactively from Caesar, [Ousterhout 1981], Kic, [Keller & Newton 1982], and other graphic editors to check selected portions of the design currently being edited. Lyra is in active use both at university and industrial sites. Rulesets for a number of MOS processes have been written.

7.2.1. Features and Restrictions

Lyra is less general than the corner-based system described in Chapter 5 in three ways:

i. It is manhattan.

ii. The form of condition expressions is restricted.

iii. Layers can not be qualified by attributes.

Lyra processes only manhattan mask data, and uses manhattan *boxes* in place of sectors; see Figure 7.1. Zero-depth sectors are implemented by thin boxes, one unit of resolution wide, as are inclusive sector edges.

The manhattan nature of Lyra is its most apparent limitation. Manhattan design simplifies CAD tools and is widely used in university settings. The restriction to manhattan data is more problematic in industry. The use of manhattan boxes to approximate sectors results in overconservative checking of diagonal tolerances; see Figure 7.2. For example a 3 unit spacing rule would flag diagonal spacings of up to $3\sqrt{2}$ =4.23 units. However tight diagonal spacings are awkward in strictly manhattan designs anyway, and rarely come up. Circuit designers at Berkeley have not found manhattan distances difficult to live with.

Condition expressions in Lyra rules must take the form,

$$\textbf{if } \{I_1 \textbf{ and } I_2 \textbf{ and } \cdots \textbf{ and } I_j\} \textbf{ then}$$
$$\{R_1 \textbf{ and } R_2 \textbf{ and } \ldots \textbf{ and } R_k\}$$

where I_1 through I_j are immediate conditions and R_1 through R_k are regional conditions. One of the immediate conditions in each rule must be either '*corner.angle*=90' or

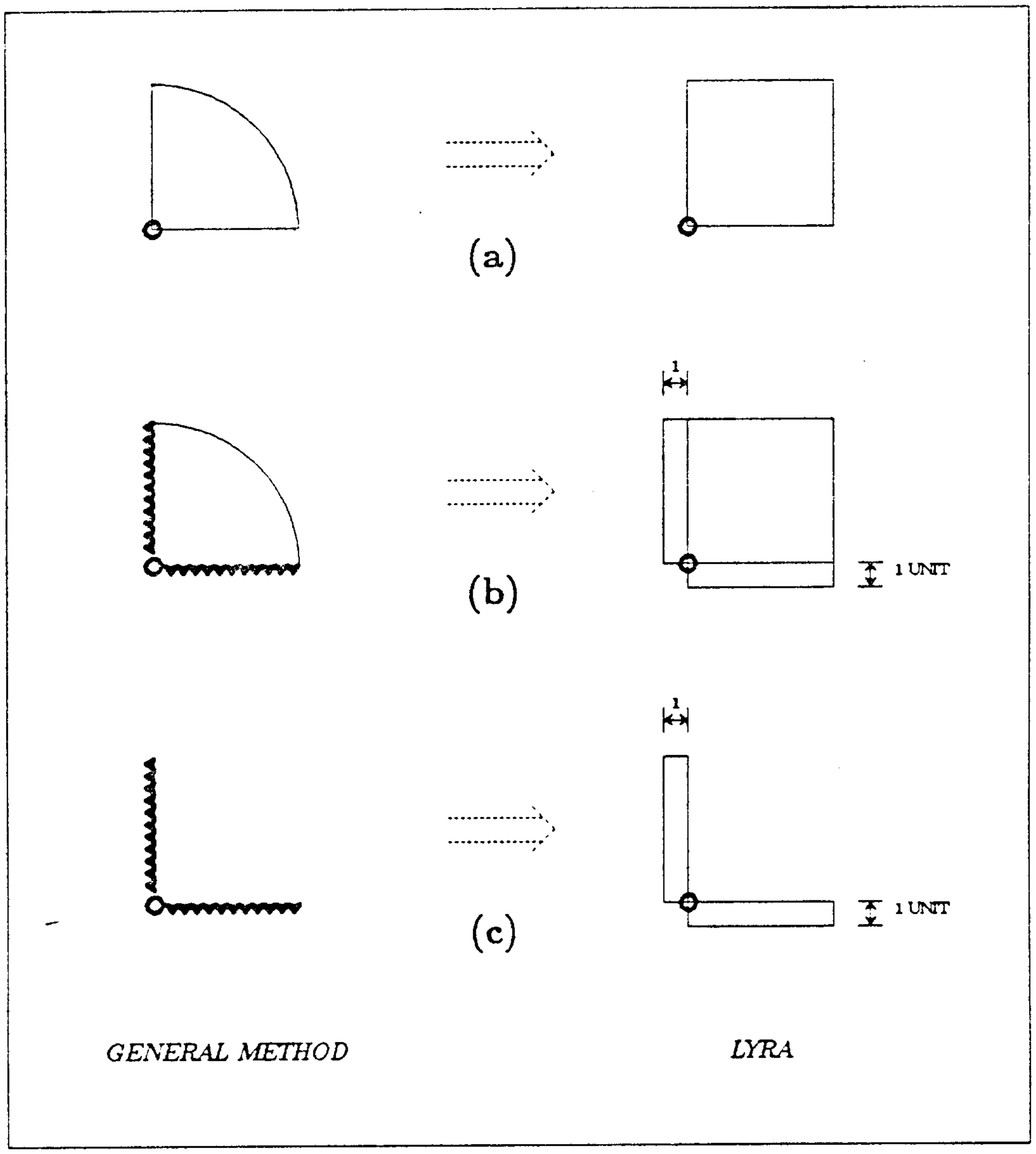


Figure 7.1. - Substituting Manhattan Boxes for Sectors. In Lyra, manhattan boxes are used in place of circular sectors, (a). Thin boxes (one unit of resolution wide) are used for inclusive edges, (b) and (c).

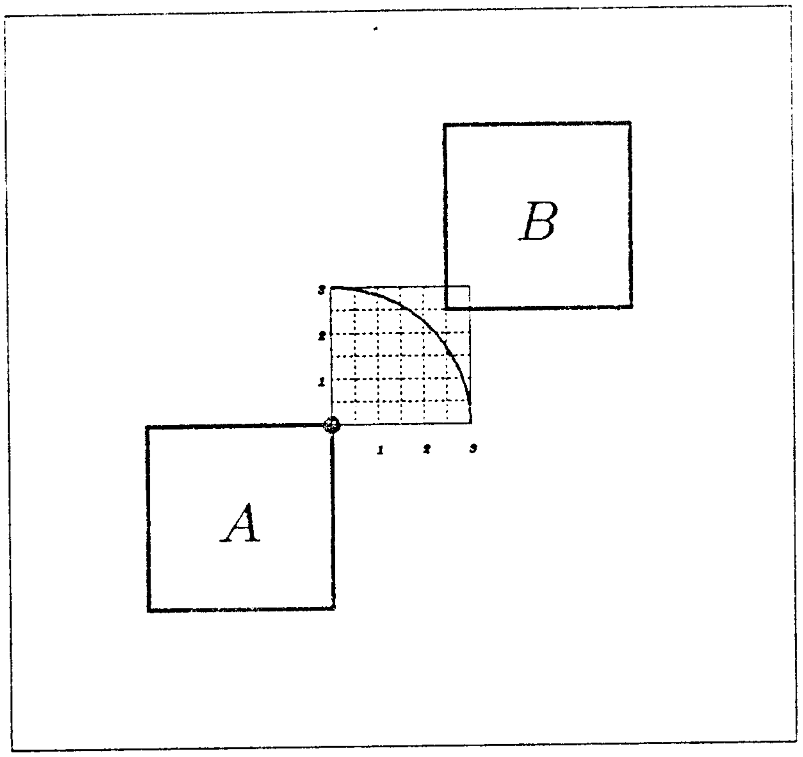


Figure 7.2. - Over-Conservative Diagonal Tolerances. Using manhattan boxes in place of circular sectors results in over-conservative diagonal tolerances. For example, above, although A and B are more than 3 units apart they would be flagged by a 3-unit spacing check using manhattan boxes.

'*corner.angle*=270'. (These are the only posisible corner angles in manhattan designs). Thus condition expressions in Lyra consist of two parts, a list of immediate *preconditions*, and a list of regional conditions. If all the preconditions hold at a corner, then all the regional conditions must also hold.

Condition expressions in Lyra are more restricted than the general expressions developed in Chapter 5. In particular expressions such as 'R_1 **or** R_2', specifying that one of two region conditions must hold, and '**not** $R1$', useful for checking that a layer is present at least *somewhere* within a sector, are not permitted. Thus complex relationships between conditions, such as might occur in the more involved industrial rules, cannot be handled by Lyra. However the great majority of rules take the form required by Lyra. Among the examples in Chapter 5, only the transistor extension rule involve condition expressions too complex to be handled by Lyra: parts (a) and (b) of this rule involve two conditionals, part (c) involves an 'else', and part (f) involves an 'or' between two region conditions. All of these expressions can be split into simpler ones that have the form required by Lyra.

The third restriction in Lyra is that layers may not be qualified by attribute information. Because of this restriction, rules depending on connectivity information, such as rules for single-node spacing and rules for spacing between buried regions and unrelated polysilicon or diffusion, can not be accurately checked by Lyra. Instead, approximations must be used that sometimes generate false violations. The major complaints of Lyra users have concerned false diffusion spacing violations, and false buried-contact-related violations, both stemming from Lyra's inability to handle connectivity. Users cope with these problems by stylizing their designs to avoid these idiosyncrasies of Lyra. Users who began designing with interactive access to Lyra (e.g. from Caesar) complain much less; they *learn* the idiosyncrasies along with the design rules, directly from Lyra.

Lyra can check the great majority of rules occurring in practice, including simple width, spacing, enclosure and extension rules, transistor and contact form rules and the exotic facing-edge and anisotropic implant rules. The capabilities of Lyra are satisfactory for simple nMOS and CMOS rulesets on manhattan designs.

7.2.2. Implementation

The Lyra system consists of 4500 lines of Lisp code for the checker, and an additional 2500 lines for the rule compiler. The implementation follows the general method described in Chapter 6 but is much simpler. Several factors contribute to the relative simplicity of the Lyra implementation.

Mask data in Lyra is organized into square bins. Lists of rectangles intersecting each bin are maintained, one list for each mask layer. This simple data organization minimizes the complexity of the low-level routines for sorting and traversing the mask data: about 100-200 lines of code suffice where 1000-2000 lines are required for scanline or corner-stitched data organizations.

Since Lyra does not support layer expressions qualified by attribute information, attribute information need not be maintained internally, and layer expression processing is simplified. In addition vertex detection is simpler, since attribute transitions need not be considered.

Lyra's restricted condition expressions simplify the rule compiler. The condition expressions, as specified by the user, can be used directly by the checker. The code to convert condition expressions first to DNF and then to optimal decision trees is not required.

The use of Lisp allows further simplifcations in the implementation of the rule compiler. Rules in Lyra are written using Lisp syntax. This permits the direct use of the parsing and macro processing facilities of the Lisp environment. In addition layer expressions are treated as Lisp expressions and compiled by the Lisp compiler, avoiding the conversion to DNF-based bitmapped form.

Together, restricted condition expressions and the use of Lisp make the Lyra rule compiler almost trivial: Macro preprocessing, parsing, and layer-expression processing are done "for free" by the Lisp environment, and condition expressions require little processing. The remaining significant function of the Lyra rule compiler is the generation of the relevant-layers index. This is accomplished as described in Chapter 6, except that, since layer expressions are not represented in DNF form, the evaluation of candidate layers is done in a slightly more complicated way that bypasses DNF.

The restriction to manhattan data, and the use of manhattan approximations for sectors, simplifies all phases of rule checking. The intersection calculations involved in finding vertex points, determining the pie-slice data structure and verifying region conditions are all simpler when only horizontal and vertical edges are involved: the coordinates of intersections can be read from the coordinates of the intersecting elements, and no slope computations are necessary.

The pie-slice data structure is also simpler in a manhattan system. Instead of an arbitrary number of pie-slices, with arbitrary edge positions, the pie is simply divided into four quadrants, and the pie-slice data structure reduces to four bit-mapped words, one for each quadrant. Corners can be found by evaluating the appropriate layer expression in each quadrant, and then doing a 16-way table lookup based on the result; the more cumbersome pie-slice consolidation of the general method is avoided.

The processing of region conditions is also simplified because Lyra is manhattan. Regions are just boxes, and hence are easy to construct. The box is wholly determined by the location of the corner vertex, and the corner orientation: since the exact angle of the corners is fixed, the size and shape of region conditions do not depend on the positions of the corner edges. The use of narrow boxes in place of inclusive edges allows uniform treatment of region edges, further simplifying condition processing.

Overall, making Lyra manhattan and restricting condition expressions allowed great simplifications in the implementation, without significantly reducing Lyra's usefulness in the university environment. The other simplifications came at greater costs. Ignoring attributes made Lyra blind to connectivity, leading to many false violations in situations related to single-node spacing and buried-contact rules. The major complaint of Lyra users concerns this inability to properly check connectivity-related rules. Implementation in Lisp made Lyra dependent on a large runtime system, and relinquished control of low-level data management to the Lisp system. This resulted in a much bulkier and slower system than would have otherwise been the case. Nevertheless the functionality of Lyra is sufficient to make it the preferred design rule checker at many universities, and its speed is comparable with traditional region-based checkers.

7.3. Mart

Mart, an integrated design rule checker and circuit extractor, was developed by Mark Shand & Bruce Nelson at CSIRO in Australia, [Nelson & Shand 1983]. Mark Shand later continued work on Mart at Xerox PARC. The DRC part of Mart is based on Lyra. Mart is neither hierarchical nor incremental.

7.3.1. Features and Restrictions

Like Lyra, Mart checks manhattan data only, and uses box-shaped condition regions rather than circular sectors. Several extensions of Lyra's capabilities were incorporated in Mart. The side edges of box regions can be inclusive or exclusive; see Figure 7.3. This eliminates the need for the long narrow sliver regions simulating inclusive edges in Lyra, reducing the total number of regions to check by about 40% and improving performance dramatically. Note however that, unlike the general case, the side edges of a region are either both inclusive or exclusive: they can not be set independently. Mart supports somewhere-style conditions as well as everywhere-style ones, that is, a layer expression can be required to hold everywhere in a region condition or just somewhere inside it. Regional conditions can be combined with **NOT** and **XOR** as well as **AND** operations, thus regional conditions can qualify each other. This allows more complex conditional rules to be checked.

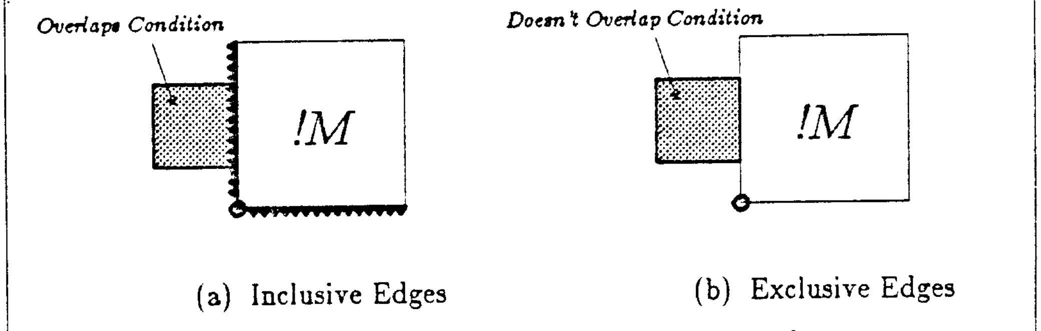


Figure 7.3. - Inclusive/Exclusive Box Edges. In Mart, the two edges of a condition box adjacent to the corner-point, can be either inclusive, (a), or exclusive, (b). As in the general system of Chapters 5 and 6, a mask feature directly abutting an inclusive edge is considered to overlap the condition region.

Layer expressions in Mart are limited to circuit layers and logical combinations of at most two circuit layers under the operations AND, OR, and NOT. This does not restrict the functionality of the system however, since circuit layers consisting or arbitrary boolean combinations of the mask layers can be generated prior to design rule checking, as part of the circuit extraction phase of Mart. Layer expressions in region conditions can be qualified to apply only to layers with node numbers equal (or unequal) to the corner layer. Thus connectivity based rules can be checked.

7.3.2. Implementation

Mart is written in C and is scanline-based. However the scanline algorithm is not "pure": corners below but near the current scanline are kept track of so that regional conditions can be checked on the same pass that corner-points are identified.

Since the form of layer expressions is restricted, conversion to DNF is not necessary.

In the interest of better readability and writability, the input rule description for Mart does not use Lisp syntax, thus a special- purpose parser is required. The parser was constructed using the Unix tools YACC and LEX.

Since regional conditions are interdependent, regional condition expressions are represented with parse trees. These parse trees are not converted to DNF or decision-trees. Instead, all regional conditions are evaluated, and then the values are plugged into the leaves of the parse tree and the tree expression is evaluated directly. However rules have the general form:

if *<immediate conditions>* **then** *<regional-expression>*

Immediate conditions are kept separately from regional conditions and evaluated first. Thus the most important strategy of the decision tree approach (early evaluation of the cheap immediate conditions) is honored.

Connectivity information is generated as part of the circuit extraction function of Mart and is readily available to the DRC.

7.3.3. Rule Checking Capabilities

Like Lyra, Mart's most apparent limitation is its restriction to manhattan data. And like Lyra, Mart also uses manhattan approximations of circular sectors, leading to over-conservative checking of diagonal tolerances.

The extensions of Mart remove many of Lyra's other deficiencies. Connectivity information allows Mart to do single-node spacing and buried/polysilicon spacing rules. Complex regional expressions allow rules requiring context established by regional conditions, such as reflection rules and open-area-dependent width rules to be handled.

The lack of general attribute capabilities makes rules employing nongeometric information other than connectivity, such as special rules for regions of high current or for VDD and GND nodes, difficult to implement.

Mart is considerably faster than Lyra, for flat (nonhierarchical) checking. The reduction in the number of region conditions afforded by the use of inclusive/exclusive edges is one factor. The use of the C language for implementation, and general attention to efficiency, are others. See Appendix I for performance measurements.

7.4. Leo45

Leo45, the Metheus DRC, developed by myself and others at Metheus, is the first commercial corner-based DRC. It is hierarchical and incremental. True to its name, Leo45 handles 45 degree angle mask data. (The first version of this program, Leo, was manhattan only. Benchmarks for both Leo and Leo45 are included in Appendix I.)

7.4.1. Features and Restrictions

Leo45 extends Lyra in a number of ways. The processing of 45-degree-angle mask data is supported and circular sectors are approximated as pieces of an octagon, rather than a square; see Figure 6.11. These approximations are quite close to true circular sectors: it is difficult to generate designs restricted to 45 degree angles for which the octagonal approximation yields different results than would be obtained with true circular sectors.

Like Mart, Leo45 permits region edges to be specified as inclusive or exclusive. In Leo45, the two region edges can be specified independently, e.g. one as exclusive and the other as inclusive. This is useful for specifying zero-width condition *fingers* that check for a layer on only one side. Such conditions are simulated with long narrow boxes in Mart and Lyra.

The Leo45 rule language is a subset of the general language developed in Chapter 5. It anticipates general condition expressions and '**somewhere_in**' style region conditions, though these are not yet supported in Leo45. Currently rules in Leo45 are restricted to the following form:

> **for** <*layer expression*> **corners_require**
>   **if** <*angle condition*> **and** <*zero depth condition*> ...
>   **then** <*sector condition*> **and** ...

As in Lyra, rules consist of a corner layer, angle restriction, list of immediate conditions, and list of regional conditions that apply (each independently) where the immediate conditions are met. Attribute qualifications are not supported.

7.4.2. Implementation

The rule language parser for Leo45 is generated using the Unix tools YACC and LEX. A modified version of the Unix M4 macro preprocessor provides a macro capability for specifying standard rules such as simple width and spacing.

Layer expressions are represented using the bitmapped DNF scheme discussed in Chapter 6. Condition expressions are still simple however, and thus do not require conversion to DNF or decision-trees.

Two-pass scanline processing, is used. Corners are identified and appropriate region conditions are generated in the first pass. The second pass checks the region conditions generated in the first.

The processing of 45-degree-angle data adds considerable complexity. The underlying scanline manipulation is more complicated when 45s are permitted. Edges sloping forward or backward at 45 degrees must be handled along with vertical edges. Special processing must be done so that intersections between two 45-degree-angle edges, or a 45-degree edge and a vertical edge, are not missed, even if they do not occur on an existing scanline. Processing of corner points is more complicated since edges can meet at various angles.

A corner is divided up into octants rather than quadrants as in manhattan systems. Thus corner-detection falls into 256 cases rather than the 16 of manhattan systems. However this is still simpler than the general pie-slice data structure described in the previous chapter. A different rule indexing scheme is tried in Leo and Leo45: corner layers are indexed based on pairs of crossing edges, rather than on the mask-layers present at corner-points. Data comparing these two methods is given in Section 8 of Appendix I. The data suggests that the crossing-edge method was probably a mistake. The mask-layers present method presented in Chapter 6 and used by Lyra is more efficient.

The generation of region shapes is much more complicated in Leo45 than in manhattan systems. The construction of a polygonal sector region given corner edge positions involves a table lookup to obtain a prototype sector region shape, and then scaling and translation to obtain the actual sector.

7.4.3. Rule Checking Capabilities

Leo45 is the first corner-based system to support 45 degree angle data, a capability needed for many industrial applications. Leo45 does not yet support attribute conditions or arbitrary condition expressions. Thus like Lyra, Leo45 can not properly handle rules involving connectivity, such as single-node spacing rules and rules requiring one regional condition to establish the context in which another applies, such as reflection rules. However Leo45's rule language is general, and anticipates future enhancements.

7.5. Magic

The Magic DRC, developed by George Taylor and John Ousterhout as a component of the Magic layout system, [Taylor & Ousterhout 1984], is hierarchical and fully incremental. The Magic DRC runs in the background checking design modifications as they are entered, and in most cases providing instantaneous feedback. Though Magic is edge-based rather than corner-based, it is more closely related to corner-based systems than to region-operation systems. Like corner-based systems, Magic uses context-based rules, employs a rule compiler to convert input rule descriptions to an efficient internal form, and indexes rules, so that the rules applying to a particular location in a design can be quickly found. The phenomonal performance of Magic demonstrates that context-based checking can be very effective.

7.5.1. Features and Restrictions

The input to the Magic DRC is not simple mask data. Magic maintains *abstract layers*, representing various types of transistors and contacts. The designer works with the abstract layers rather than the detailed device structures, and the design rules are phrased in terms of these layers. Prior to fabrication, abstract layers are automatically replaced with the appropriate device constructs.

The use of abstract layers removes much of the complexity from design rules. In particular, boolean layer combinations are not required to identify device parts (e.g. transistor gates). The Magic DRC has no provisions for specifying layer combinations.

Rules are edge-based. They specify a left and right layer, a distance and a set of permissible layers. A rule applies to all edges with the left layer on the left and the right layer on the right; see Figure 7.4. For such edges only the permissible layers are allowed for the specified distance. Of course rules can apply right to left, top to bottom, and bottom to top also. Empty space is a special layer, thus width rules can be expressed by omitting empty space from the list of acceptable layers. An extension for checking at corners is provided so that diagonal tolerances can be checked correctly. The extension distinguishes between convex and concave corners.

Since edges and corners are checked, tolerance checks are essentially implemented by checking a complete halo about regions. One consequence of this is that tolerances between two layers need only be checked from one direction. This is an important consideration in Magic, since design rules are also used to direct *plowing*, an operation which compacts in a particular direction.

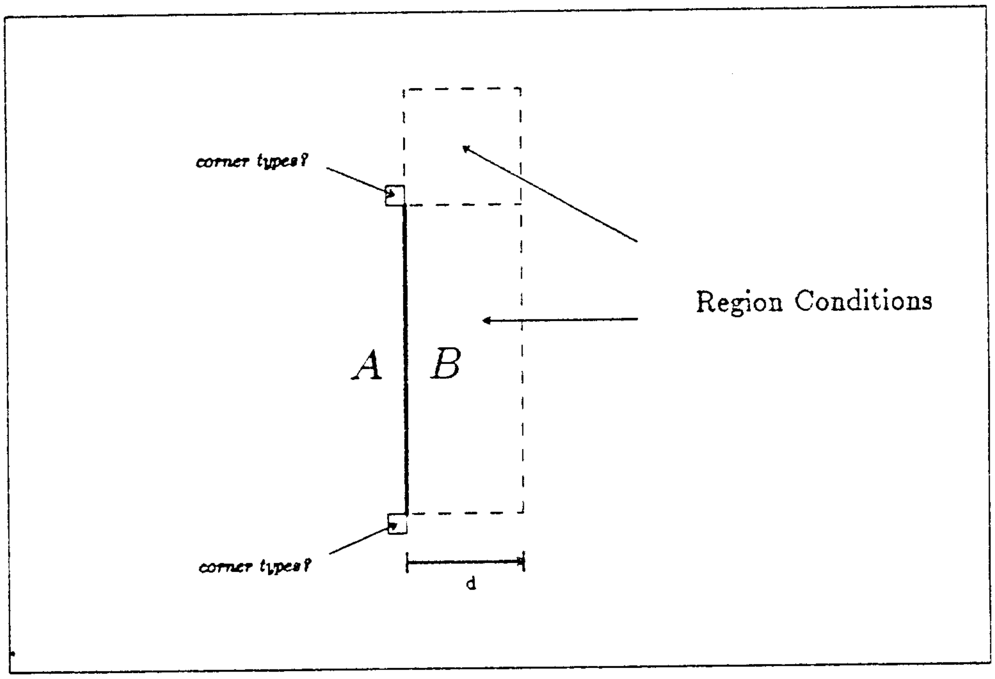


Figure 7.4. - Magic Rules. Magic design rules refer to edges. A typical rule specifies that whenever a layer A is present on the left side of an edge and a layer B on the right, then only certain layers may be present for a distance d to the right of the edge. This condition can be conditionally extended into the corners, depending on the layers present just beyond and to the left of the edge.

7.5.2. Rule Checking Capabilities

The rule checking capabilities of Magic are comparable to Lyra. Both systems can check simple spacing, width, extension and enclosure rules, and the context-based rules in both systems handle simple conditional rules. On the other hand both systems are restricted to manhattan data, use manhattan distances at corners resulting in overconservative diagonal tolerances, and can not properly handle rules involving connectivity. Since there is no provision for boolean combinations of layers in Magic's design rules, the DRC depends on the existence of suitable abstract layers for checking the more complicated rules.

7.6. Intel DRC

An internal hierarchical design rule checker at Intel, developed by Todd Wagner, [Wagner 1984], combines the region-operation approach with the corner-based idea of point/edge processing. Design rule checking is done with sequences of primitive operations. Primitives include boolean, topological, sizing and tolerance check operations, just as in traditional region-operation systems, but tolerance checks are implemented using point/edge comparisons. This "corner-based" processing, localizes violations to specific corner-points, facilitating clean hierarchical processing.

7.7. Summary

Current corner-based systems are all simpler than the general system presented in the previous chapter. Simplifications include restriction of input data to manhattan shapes, approximation of circular sectors with boxes, restrictions on how conditions can be combined in condition expressions, and ommission of attribute conditions. These simplifications limit the rule checking capabilities of the systems. Typical limitations are an inability to check nonmanhattan data, overconservative diagonal tolerance checks, the inability to check certain conditional rules such as reflection rules, and the inability to properly check rules involving connectivity, such as single-node spacing rules.

However not all the systems have all the limitations. Taken together, the systems come close to exhibiting all the features of the general system. While none of the systems have a general attribute capability, permit completely general combination of conditions, or can handle all angle input data, Mart has provisions for handling connectivity attribute information and allows interdependent region conditions to be specified, while Leo45 handles 45 degree angle design data.

The Magic and Intel systems presented at the end of the chapter show that the two key ideas of the corner-based approach, context-based checking and point/edge tolerance checks, can be applied independently. The Magic DRC demonstrates that context-based checking can lead to a very fast implementation, and the Intel system shows how point/edge tolerance checking can be combined with a traditional region-based system.

7.8. References

[Arnold & Ousterhout 1982]

M.H. Arnold & J.K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking," *Proc. 19th Design Automation Conference*, June, 1982, pp. 530-536.

[Keller & Newton 1982]

K. Kenneth and R. Newton, "KIC2: A Low Cost, Interactive Editor for Integrated Circuit Design," *Digest of Papers for COMPCON*, Spring 1982.

[Nelson & Shand 1983]

B.J. Nelson & M.A. Shand, *An Integrated, Technology Independent, High Performance Artwork Analyzer for VLSI Circuit Design*, Technical Report VLSI-TR-83-4-1, VLSI Program, Division of Computing Research, CSIRO, Adelaide, South Australia, April 1983.

[Ousterhout 1981]

J.K. Ousterhout, "Caesar: An Interactive Editor for VLSI Layouts.", *VLSI Design*, Vol. 2, No. 4, Fourth Quarter 1981.

[Taylor & Ousterhout 1984]

G.S. Taylor & J.K. Ousterhout, "Magic's Incremental Design-Rule Checker," *Proc. 21st Design Automation Conference*, June, 1984, pp. 160-165.

[Wagner 1984]

T. Wagner, Personal Communication, Intel Corporation, Santa Clara, California, 1983.

CHAPTER 8

Hierarchical and Incremental Checking

8.1. Introduction

Checking entire VLSI designs on each DRC run is wasteful. VLSI designs generally contain large amounts of repetition. Examples are arrays of memory cells, nearly identical bit-slices, and other repeated function blocks. It should not be necessary to recheck every instance of such repeated blocks. Also portions of a design that have not been modified since the last DRC run do not need to be rechecked. This waste is particularly acute at the end of the design cycle, when minor changes to a design, correcting problems detected by DRC or simulation runs, necessitate a follow-up DRC run requiring many hours of computer time. Never-the-less, most current design rule checkers still check an entire design each time they are invoked.

This chapter is concerned with hierarchical and incremental checking, two strategies for reducing unnecessary checking. Hierarchical processing eliminates redundant checking by processing subcells only once, regardless of how many instances of them are present in a design. Incremental processing eliminates redundant checking of unchanged portions of a design. Together these techniques greatly reduce the CPU time and memory requirements for design rule checking. A hierarchical check of the RiscI microprocessor chip with Leo45 takes only 17% of the time of a *flat*, (i.e., nonhierarchical) check, and a hierarchical/incremental recheck after modification of a small cell takes less than 1% of the time for a full flat check; see Chapter 9. Hierarchical/incremental checking also eliminates redundant violation reports that can hide important violations in a sea of repetitious output, and permits more interactive checking, giving users early warnings on design rule violations while they can still be easily fixed.

Hierarchical and incremental strategies are largely independent of the underlying DRC method, and hierarchical DRC's have been built on top of region-based systems. However the corner-based method has a nice property that makes it particularly well suited for use in a hierarchical or incremental system: violations are associated with specific points in the design, rather than edges or areas. Thus the problem of handling violations straddling the region currently being checked does not arise in corner-based systems. Except for this, the underlying DRC method is unimportant in this chapter.

Several approaches to hierarchical checking differing from mine have been proposed, and a few have been implemented. The next section introduces hierarchical checking and discusses these approaches. The following section presents my approach to hierarchical checking, as implemented in Lyra, Leo, and Leo45. This method is also used in Magic. It is distinguished from others in that it works directly with the hierarchy as the designer sees it, and imposes no constraints on cell overlap. The extension of this method to incremental checking is discussed in the following section. Incremental checking was originally implemented in Leo45, and further extended (by George Taylor & John Ousterhout) in Magic. I know of no other incremental design rule checkers. The chapter concludes with a summary.

8.2. Hierarchical Checking - Background

VLSI circuit designs are represented hierarchically; see Figure 8.1. The top level cell is composed of mask features and subcell instances. The subcells in turn contain more mask features and subcells. The hierarchy eventually terminates with leaf cells, which contain no subcells of their own. A design can contain many instances of a single subcell.

Hierarchical representation modularizes the design, and makes repetition explicit, namely, as repeated instances of subcells. This permits more structured designs, greatly reduces the size of the design database, and, potentially, facilitates fast hierarchical design rule checking and circuit extraction.

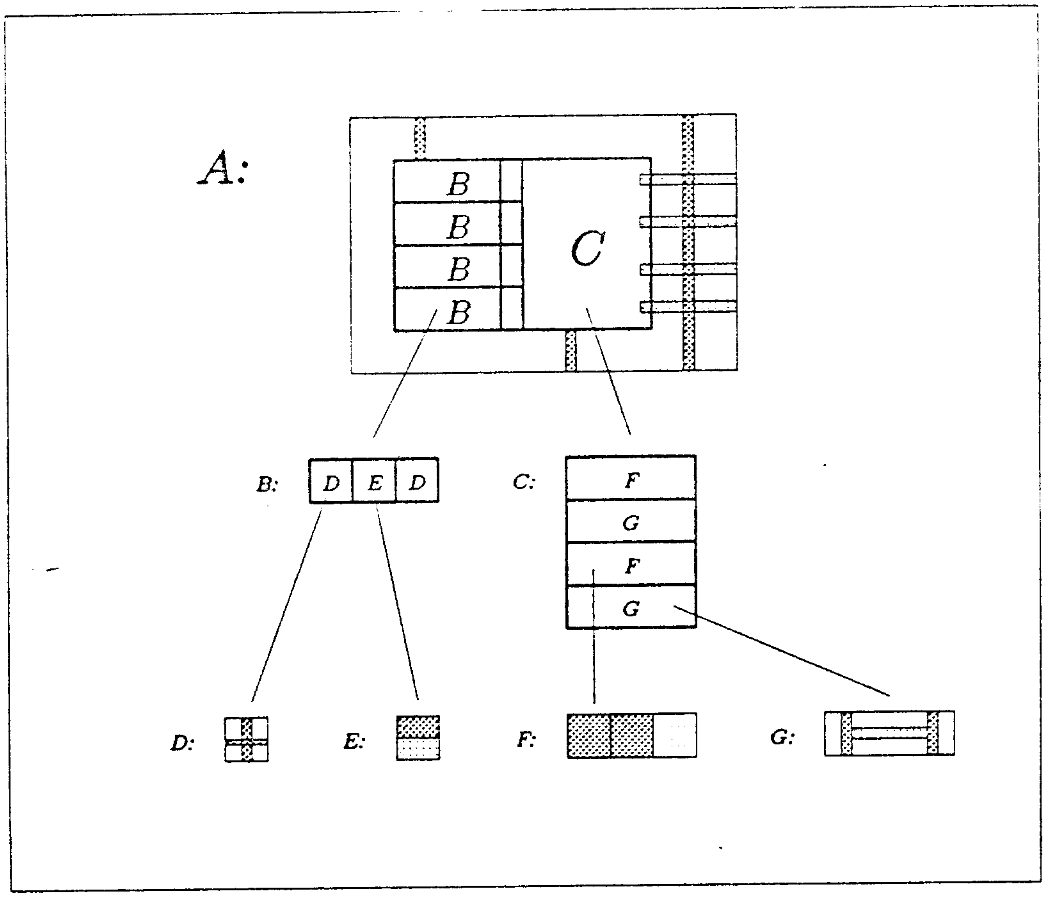


Figure 8.1. - Hierarchical Representation of Designs. A hierarchical design is divided into *cells*. Cells can contain both mask features and instances of other cells (subcells). A three-level hierarchy is illustrated above. The top-level cell, A, contains four instances of a cell B, and one instance of the cell C. The cells B and C in turn contain instances of D, E, F and G. These last four cells are *leaf* cells: they have no subcells.

The idea of hierarchical processing is to process each cell just once, regardless of how many instances of it occur in a design. The interfaces between cells must also be checked. For example a hierarchical design rule checker might check each cell once, and then check regions near instance boundaries for violations involving interactions between cells.

Hierarchical processing enhances performance by eliminating the need to recheck each instance of a cell. In addition the time and space penalities of creating and working with the bulky flat representations of designs are avoided. Hierarchical checking also provides a more convenient interface to the designer. Design rule violations are reported directly in terms of the cells the designer is working with, rather than in terms of global coordinates he must ultimately translate back to cell coordinates. Also violations in cells are reported only once regardless of how many times the cells are repeated. Anyone who has sorted through reams of output generated by a traditional DRC, searching for a few distinct violations, will recognize the importance of this feature.

The key problem in hierarcical checking is the handling of cell interactions. Even if each cell is correct when considered in isolation from its parents or children, there may be design rule violations that occur because of interactions between features in neighboring cells. All such interaction must be checked. In practice, the overhead of checking cell interactions limits the effectiveness of hierarchical checking. For example, a hierarchical check of the Delay design by Leo45 is actually slightly *slower* than a flat check, despite the fact that on the average their are nearly 20 instances of each rectangle! The reason is that the interaction checks sums to over half the area of the design. (See Chapter 9 for more, albeit less extreme, examples.) Different methods of hierarchical checking, differ chiefly in how they handle cell interactions.

The following subsections discuss various methods for hierarchical checking that have been proposed, and in most cases implemented.

8.2.1. Whitney's Filter

Telle Whitney developed a hierarchical filter, [Whitney 1983], that creates a flattened version of a design with many of the redundant mask features removed. A traditional flat DRC is run on the output of the filter. Versions of the filter have been used at both Caltech and DEC. For one design the DRC time with the filter (including the time to run the filter) was 20% of the time required without the filter. This design was very regular: there were an average of 74 instances of each transistor specified by the user.

The filter works as follows. A representative instance of each cell is chosen and all the mask features contained directly in that cell are output as well as all mask features of other cells (including subcells) that are near enough to interact. Interaction checks between pairs of subcells are noted, and a given configuration is output only once. Thus the filter checks one instance of each cell and one instance of each cell/cell interaction. In outline, the algorithm is as follows:

```
CheckCell(c):
  if (c not already checked)
        write out all mask features in c for checking;

        for each (subcell, sc, in c)
           CheckCell(sc);
           write out all mask features in sc that interact with features in c;
           for each (other subcell, sco, in c)
                if (sc and sco interact and interaction not yet checked)
                   write out all pairs of interacting features, with
                   one element drawn from sc and the other from sco;
                endif
           endfor
        endfor

        mark c as checked;
  endif

  return;
```

The algorithm is not completely reliable: three way interactions between features in different cells can cause genuine violations to be missed and false ones to be reported; see Figure 8.2. This problem seems to be intrinsic to the filter approach. As long as only part of the design is written out for checking, erroneous checking can result from missing context near the edges of the parts that are written out.

8.2.2. Scheffer's Strict Hierarchy

Louis Scheffer developed the concept of *strict hierarchy* in his PhD thesis at Stanford [Scheffer 1983] and incorporated these ideas in systems at Hewlett Packard and Valid Logic Systems. A strict hierarchy allows no overlap between subcells or between a subcell and mask features in the parent; see Figure 8.3. In addition devices may not cross cell boundaries, and all connection points on a cell boundary must be explicitly labeled as ports. Cell boundaries

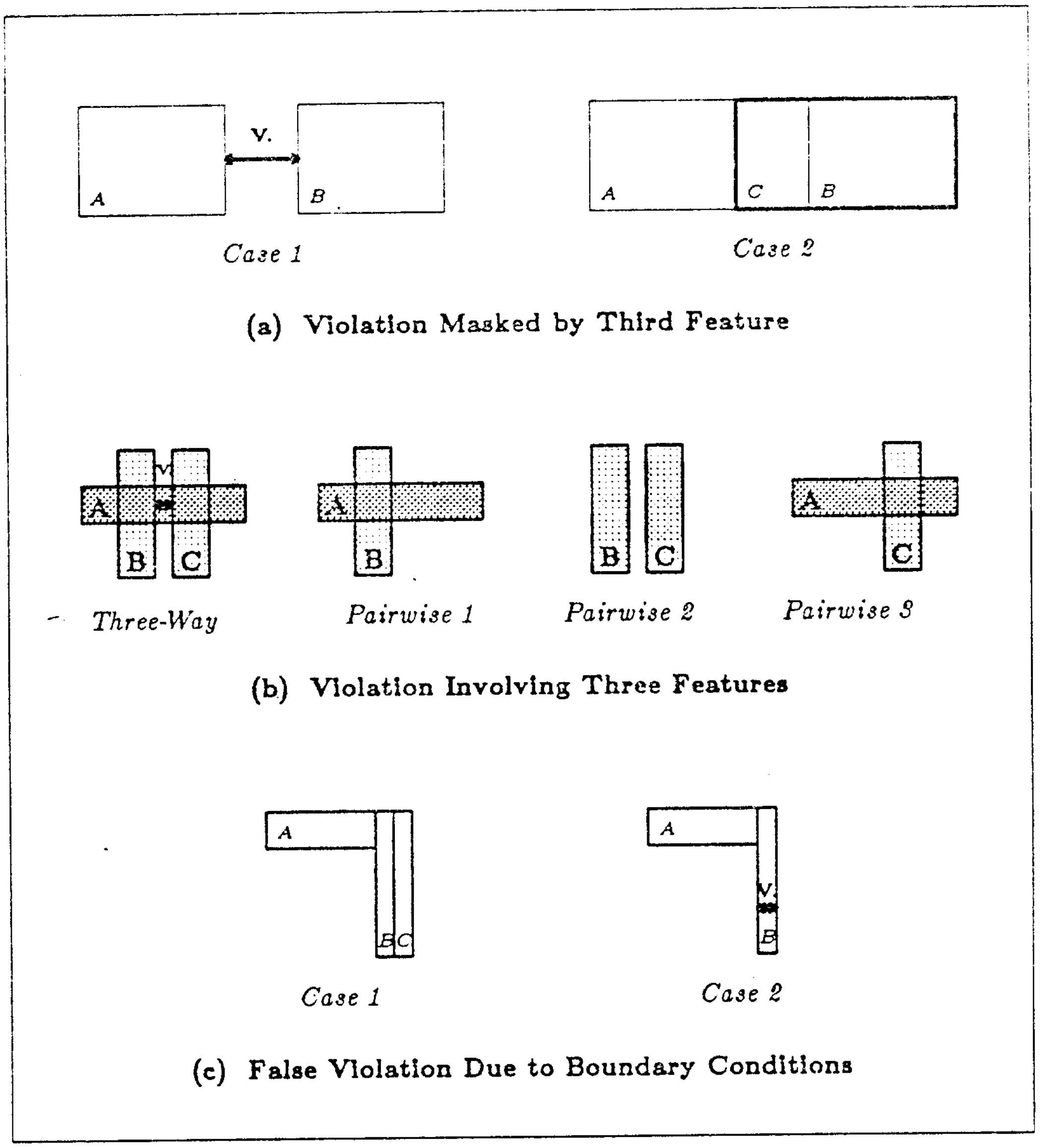

V.
A
B
Case 1
A
C
B
Case 2
(a) Violation Masked by Third Feature
V
A
B
C
Three-Way
A
B
Pairwise 1
B
C
Pairwise 2
A
C
Pairwise 3
(b) Violation Involving Three Features
A
B
C
Case 1
A
V
B
Case 2
(c) False Violation Due to Boundary Conditions

Figure 8.2. - Problems With the Hierarchical Filter. The pair-wise interaction paradigm of Telle Whitney's filter does not handle three-way interactions reliably. The examples involve three mask features assumed to belong to separate cells, A, B and C. In (a), if C is present over the instance of the interaction between A and B written out for checking, (case 2), but not over all instances, (case 1), violations present only in the absence of C will be missed. If all pair-wise interactions between A, B, and C are OK, as in (b), and present somewhere in the design, a violation involving a three-way interaction between the features may be missed. False violations reports can also result if two features are checked outside the context of a third nearby feature. For example, in (c), if A and B are written out without C, a false violation results. Problems like the above seem intrinsic to the filter approach.

are not restricted to rectangles; the user can specify a general manhattan polygon boundary. Of course all the mask features of a cell must be contained within its boundary.

Scheffer argues that the use of strict hierarchy is a good discipline for designers as well as being useful for hierarchical design rule checking and circuit extraction. Strict hierarchy incorporates the software engineering concepts of clean, explicit, interfaces between modules and nested scoping.

If strict hierarcy is employed, checking interactions between cells reduces to checking the parts of a cell near the edges with the parent(s) of the cell. More precisely, the portions of a cell within the largest design rule interaction distance (*I-radius*) of the edges, are checked

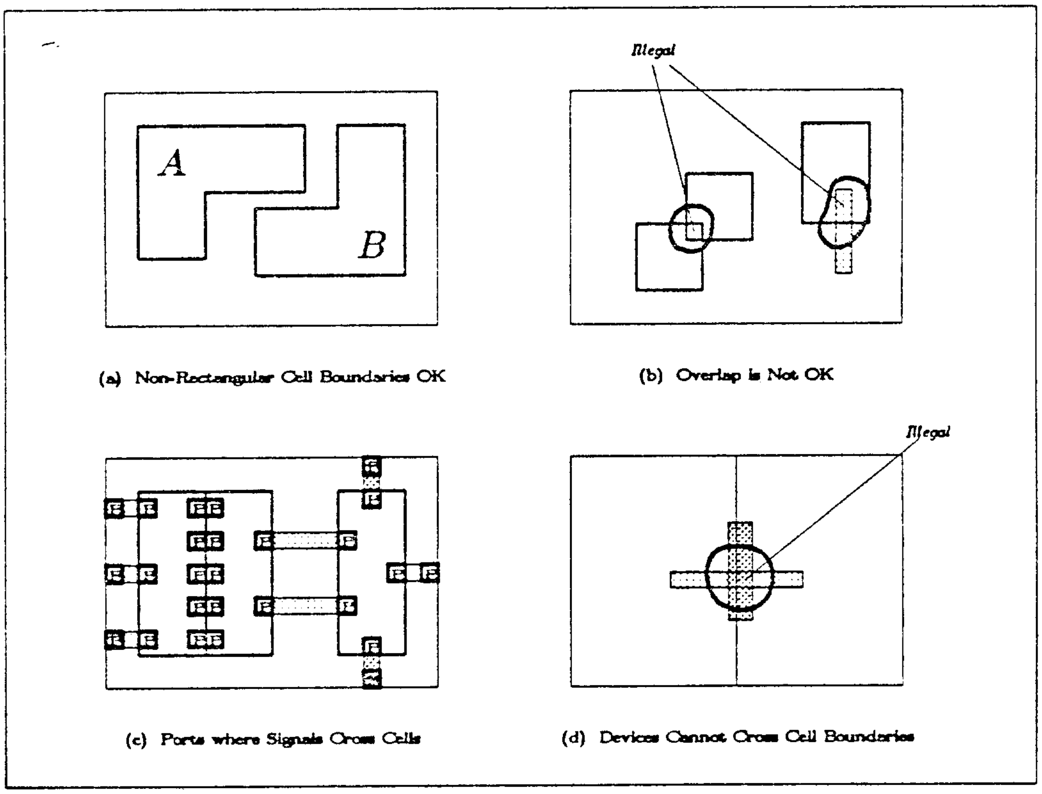

Illegal
A
B
(a) Non-Rectangular Cell Boundaries OK
(b) Overlap is Not OK
Illegal
(c) Ports where Signals Cross Cells
(d) Devices Cannot Cross Cell Boundaries

Figure 8.3. - Strict Hierarchy. Louis Scheffer suggests the use of strict hierarchy to simplify hierarchical processing of designs. Strict hierarchy allows nonrectangular cell boundaries as in (a), but constrains interaction between cells. Cells are not allowed to overlap with other cells or mask features as in (b), points where signals leave a cell must be explictly label as ports as in (c), and devices can not straddle cell boundaries as in (d).

with the the parent. This is illustrated in Figure 8.4.

Nonoverlapping cells do simplify hierarchical processing and avoid situations, involving large overlap between cells, that lead to anomalous behavior in other systems (see next section). But the price is high. Restrictions on design style reduce the designer's flexibility and limit the use of the checker to environments where the restrictions are honored. In situations where cell overlap is natural and convenient, such as shared busses, the designer must manually fragment the layout into non-overlapping cells. This can result in many variants of a given cell, each used in a different overlap situation. Strict hierarchy can also force global wiring to be done in an unnatural way, splitting a wire between several cells it happens to pass through, and defining ports at each of the cell interfaces. The added complexity of nonrectangular cell boundaries is necessary to keep nonoverlap of cells from becoming impractically restrictive.

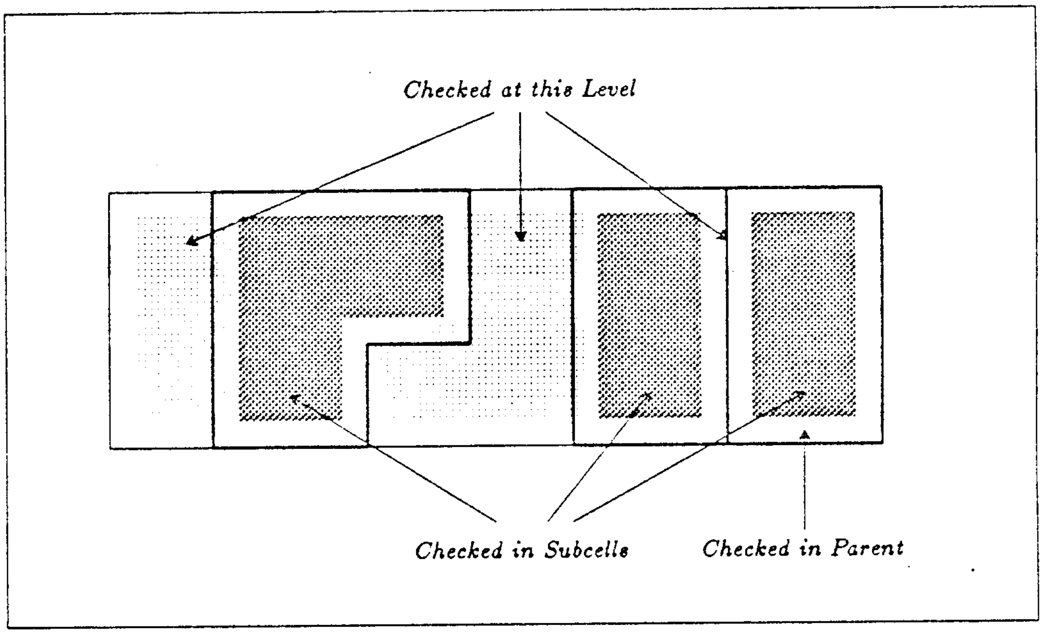


Figure 8.4. - Hierarchical Checking of Strict Hierarchies. If strict hierarchy is employed, each cell divides into an outside margin, to be checked in the parent cell, an interior (including the outside margins of subcells), to be checked at this levels, and the interiors of subcells, to be checked at the subcell level.

The value of forcing designers to adhere to a strict hierarchy is unclear. Requiring strict hierarchy for design rule checking reduces the flexibility of designers and limits the scope in which the program can be used. These disadvantages must be weighed against the possible advantages of clean cell interfaces. The derived disjoint hierarchy approach discussed below, and the unrestricted hierarchy approach given in the next section explore hierarchical checking of hierarchies that need not be strict.

8.2.3. Newell and Fitzpatrick's Derived Disjoint Hierarchy

Martin Newell and Daniel Fitzpatrick developed a circuit extractor that allows arbitrary cell overlap in the design hierarchy, but uses an automatically derived *disjoint* hierarchy internally [Newell & Martin 1983]. They define a disjoint hierarchy as a hierarchy in which subcells do not overlap with each other or with mask features from the parent cell, just as in Scheffer's strict hierarchy. However, disjoint hierarchy differs from strict hierarchy in that transistors may be split across cell boundaries.

Figure 8.5 illustrates the transformation to disjoint hierarchy. The disjoint hierarchy is derived by dividing the area of the design into regions uniformly covered by specific subcells or combinations of subcells, and then creating a new hierarchy with these regions as the cells. The disjointing process is continued recursively with each subcell, until leaf cells are reached.

In Newell's circuit extractor, the disjointing process is carried out using a scanline algorithm. Results show that derivation of the disjoint hierarchy is quite efficient. Once the disjoint hierarchy is obtained, checking can proceed just as with Scheffer's strict hierarchy.

This approach has the advantage of being general and not burdening designers with restrictions. Its major disadvantage is that a new intermediate internal representation must be derived and used. Violations in derived cells must be converted back to violations in the original hierarchy, and it is not trivial to determine which original cell a violation belongs in. The use of polygonal boundaries (at least internally) also adds complexity.

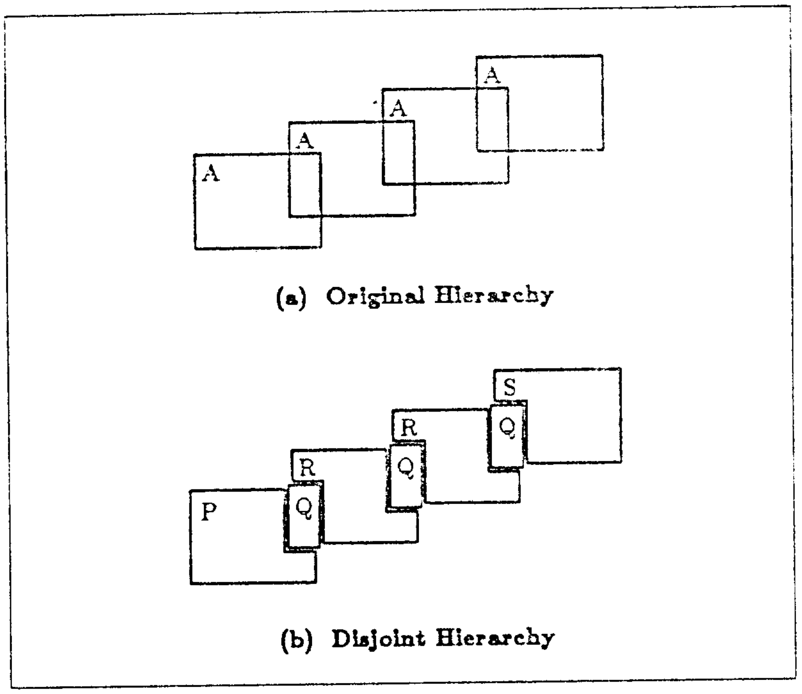


Figure 8.5. - Transformation to Disjoint Hierarchy. A disjoint hiearchy is automatically created by turning regions of cell overlap into independent subcells. Notice that the derived hierarchy can contain nonrectangular cell boundaries, even if the original hierarchy contains only rectangular cells.

8.3. Direct Processing of Unrestricted Hierarchy

This section presents an alternative method of hierarchical checking I have developed that allows unrestricted overlap of cells and works directly with the design hierarchy. This method has been implemented in Lyra, Leo and Leo45. George Taylor has implemented a similar method in Magic.

In this approach, interaction regions are identified and checked; see Figure 8.6. The maximum design rule interaction distance for a ruleset, I-Radius guides the process. The vicinity of each subcell is searched for other subcells, or mask features. Any cell or mask feature within I-Radius of the cell boundary is considered to interact with it. Check regions for interacting objects are computed by expanding their bounding boxes by I-Radius in each direction and then intersecting them. The resulting regions are checked in a context extending out I-radius in each direction.

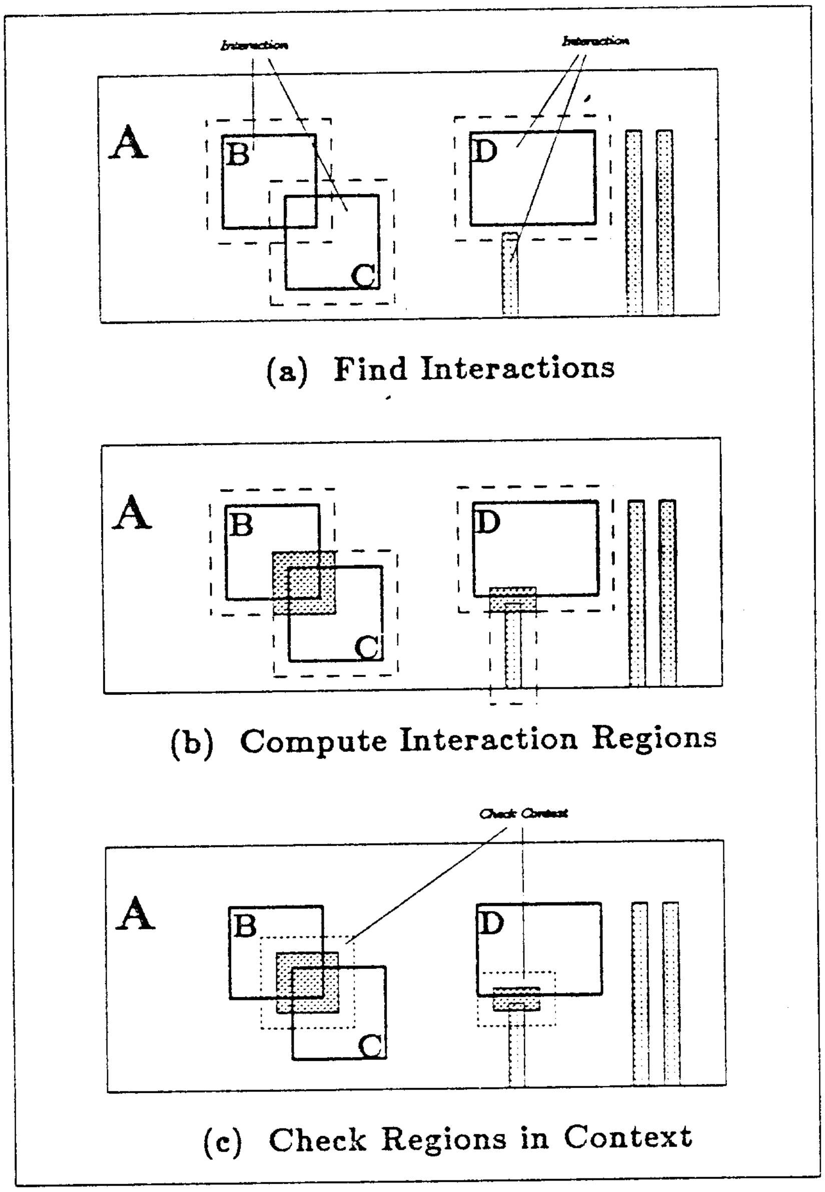


Figure 8.6. - Checking Interactions. Interactions are identified by searching for cells or features within I-radius of each subcell boundary (a). A check region is computed for each pair of interacting objects, by growing their bounding boxes by I-radius and then intersecting them (b). Check regions are checked in the context of a slightly larger region that extends beyond the check region by I-radius in each direction (c).

Checking a cell divides into three phases:

i. Check mask features contained directly in cell (pretending subcells don't exist).

ii. Check subcells (recursively).

iii. Compute and recheck interaction areas, as described above.

Ignoring subcells in i., can introduce false violations. These will be removed in step iii. when the regions involved are rechecked.

Performance is further enhanced by handling arrays specially. Instead of checking all interactions between cells of an array, only representative interactions are checked; see Figure 8.7. Since arrays account for much of the regularity of VLSI designs, and such special handling greatly reduces the overhead required for checking arrays, this method improves performance of the checker dramatically. For example the overhead (total interaction area / area of chip) for RiscI, which contains large explicit arrays, is only 11% and hierarchical processing speeds up checking by almost a factor of six. In contrast the overhead for the Delay chip, which contains no explicit arrays, is over 50%, and hierarchical checking is actually slightly slower than flat checking. This is despite the fact that Delay is essentially a single large array (though not explicitly specified as an array in the design file) and is more regular than RiscI; see Chapter 9.

Merging adjacent and overlapping interaction regions before checking them, improves performance still further. Merging interaction regions reduces both the number of regions to be checked and their total area. Leo45 uses the same merging algorithm employed in the Caesar layout editor [Ousterhout 1984] to merge interaction regions into maximum horizontal strips before checking them. This reduces check time by about 35%.

Though the above method of hierarchical checking does not restrict overlap between cells, it does restricts designers in another way: each cell must be design rule correct in

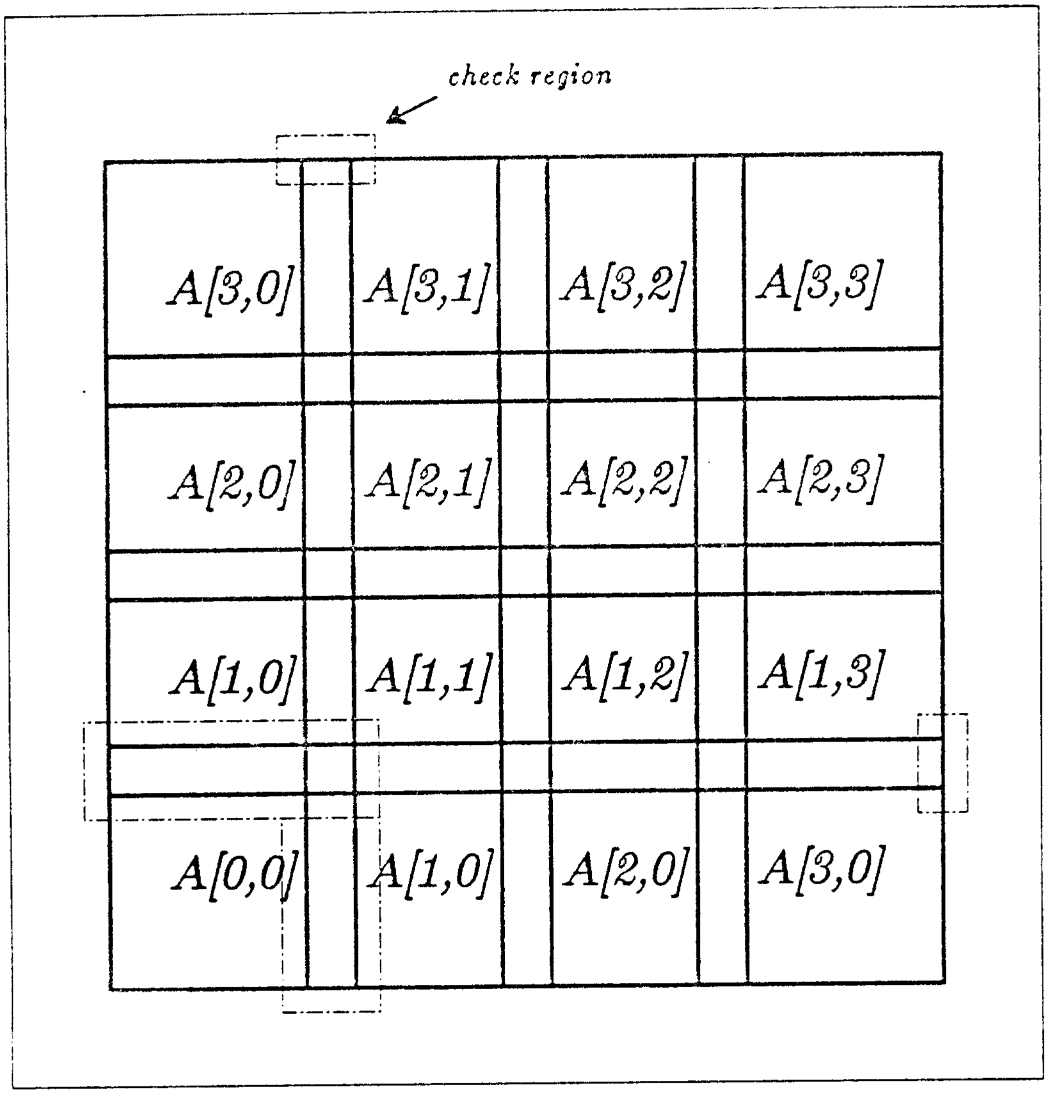


Figure 8.7. - Special Handling of Arrays. Interactions between component cells in an array are repeated over and over. Performance is greatly improved by checking only representative regions (the dashed regions above). All interactions in the array are identical to interactions in these regions.

isolation: a cell may not contain half a bus, half a contact or half a transistor. This seems to be a reasonable restriction, and no designer has complained about it. Hierarchical processing is of course most effective for designs involving little overlap, and degrades steadily as the amount of overlap increases. In this methodology, designers are free to exercise their own judgement, but must pay for their sins.

The main problem with my hierarchical algorithm is that big sinners are punished too severely. Designs containing large amounts of overlap can take several times longer to check hierarchically than flat! This is because the mask features inside overlap regions can end up being checked multiple times, i.e., first while checking the cells involved in the overlap, and then again when interactions between cells are checked.

For example, pads and global wiring are frequently placed in one subcell, while the rest of the design is placed in another. A hierarchical check of such a design checks the design hierarchically, and then once again flat. The flat check is done because the entire design is one large interaction region between the pad and wiring cell and the cell containing the rest of the design. This particular problem can be avoided by making the body of the design a subcell of the pad and wiring cell, rather than making them both subcells of a common top-level cell.

8.4. Incremental Checking

Incremental checking involves only checking the parts of a design that have been modified since the last DRC check. Incremental checking greatly reduces the time required for individual DRC runs, particularly near the end of the layout process. Benchmarks on the RiscI chip show that an incremental check of the chip after modification of the toplevel cell (which contains the global wiring) takes 40% of the time of a full hierarchical check, and the incremental check time after modification of a leaf cell is 1% of the time for a full check.

By reducing the computing resources required for design rule check runs, incremental checking permits frequent checks, creating a much more interactive environment. Early detection of design rule violations can save much effort later, since, for example, a spacing violation can be fixed before it is boxed in by other parts of the design. Incremental design rule checking also encourages greater experimentation and refinement at the end of the design process, since each change does not require another full DRC run on the design. Finally, incremental checking provides automatic checkpointing. If a long DRC run is interrupted because of computer failure, the next incremental run will automatically resume where the last one left off.

I know of only two incremental design rule checkers: Leo45 (and its predecessor Leo), developed as part of this research, and the recently developed Magic design rule checker. These two systems are discussed below.

8.4.1. Leo45

Leo45 is incremental by cell. It rechecks only the cells that have been modified since the last check, and interactions involving those cells. Since each cell is stored as a separate Unix file, the date of last modification is available from the operating system. The time of the last design rule check is stored in each cell. Thus *modified* cells can be detected by comparing the modification date for the file containing the cell with the last-checked-date recorded in that file. In addition to the notion of a *modified* cell, the incremental algorithm makes use of the concept of an *impacted* cell. A cell is impacted if one of its descendent cells (e.g. subcell or subcell of a subcell, and so on) is modified. Cell interactions involving impacted cells must be rechecked. Incremental checking in Leo45 is achieved by adapting the hierarchical algorithm presented in the last section to distinguish between modified, impacted, and unaffected cells. In outline the algorithm goes as follows:

```
checkCell(c):
  if (c not already CHECKED)

      for each subcell sc
        checkCell(sc);
        if (sc IMPACTED or MODIFIED)
            mark c IMPACTED;
        endif
      endfor

      if (c MODIFIED)
        check mask features in c;
        for each (subcell, sc)
            check interactions involving sc;
        endfor

      else if (c IMPACTED)
        for each subcell sc
            if (sc MODIFIED or IMPACTED)
              check interactions involving sc;
              if (bounding box of sc changed)
                  recheck region of previous bounding box;
              endif
            endif
        endfor
      endif

      mark c CHECKED;
  endif

  return;
```

Figure 8.8 illustrates an incremental check by Leo45. Note that modified cells are entirely rechecked, while all interactions involving impacted cells are rechecked. Modifying a cell can change its bounding box. When this occurs, violations may be introduced or removed in the region formerly occupied by the bounding box as well as in the region of the new bounding box. Thus old bounding boxes of subcells are stored with each cell, and the relevant regions are checked when a subcell is modified.

Incremental checking in Leo45 has proven very effective in reducing the time required for DRC runs, and has facilitated frequent use of the DRC during the design process. See Chapter 9 for numerical examples.

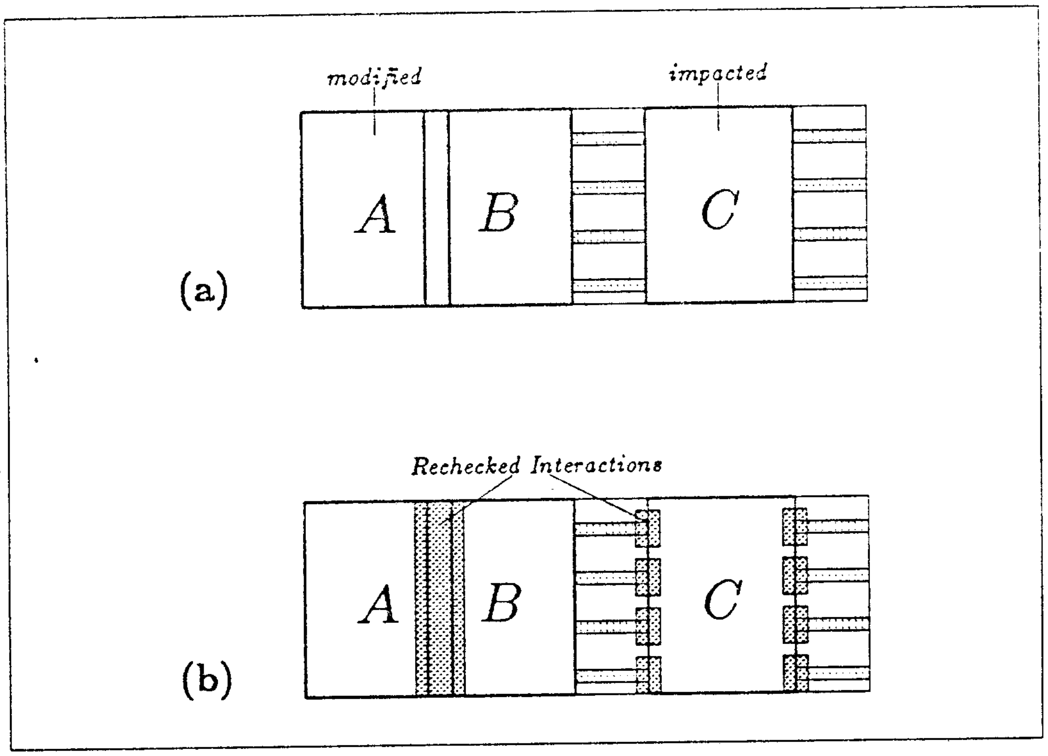


Figure 8.8. Incremental Checking in Leo & Leo45. For purposes of incremental checking, subcells are classifed into those modified since the last DRC, those impacted (i.e. having a descendent that has been modifed), and those that are neither modified nor impacted, (a). All interactions between parent cells and modifed or impacted children are rechecked, (b). In addition all mask features and subcell interactions in modifed cells are rechecked (not shown).

8.4.2. Magic

The Magic design rule checker runs as a background process, incrementally checking each modification to the design and giving, in most cases, instantaneous feedback on design rule violations. The Magic editor marks regions that have been modified with *to-be-checked* rectangles on a special layer. As the design rule checker checks these regions, it removes the *to-be-checked* rectangles. The use of *to-be-checked* rectangles, made possible by the close coupling between the layout editor and the DRC, eliminates the need to completely recheck modified cells: incremental checking proceeds on a much finer grain.

Because Magic is hierarchical as well as incremental, the impact of changes must be checked both up and down the hierarchy. Backpointers to parent cells are kept internally in the Magic system, so that the impact of changes can be traced up the hierarchy.

Continuous design rule checking in Magic works very well. The concept of design rule checking as a long batch run to be performed at the end of the design cycle is completely absent in Magic.

8.5. Summary

Most design rule checkers check an entire design on every invocation. This is wasteful in two ways. First, repeated structures are rechecked at every occurrence. Second, parts of the design that have not been modified since the last DRC run are rechecked. This chapter has presented two techniques for reducing such wasteful redundant checking, hierarchical checking to avoid rechecking repeated instances of the same structure, and incremental checking to avoid rechecking parts of designs that have not changed since the last DRC run. Both techniques have proven effective. A hierarchical check of RiscI by Leo45 takes only 17% of the time of a flat check, and an incremental check after minor modifications to a design takes only about 1% of the time for full check.

The basic idea of hierarchical checking is to check each cell only once, regardless of the number of instances of the cell in the design. However this is complicated by the need to check interactions between cells. Several methods have been proposed to solve this problem. Telle Whitney invented a hierarchical-filter based on pairwise interactions. The idea is that geometries are output only for the first instance of each pairwise interaction between cells, and then a traditional DRC is run on the output of the filter. Unfortunately, this method is not completely reliable. A second approach, championed by Louis Scheffer, simplifies interactions between cells by disallowing overlaps between cells. Yet another approach, proposed by Martin Newell and Dan Fitzpatrick, permits arbitrary overlap between cells, but generates a modified disjoint (or nonoverlapping) hierarchy prior to checking.

My approach to hierarchical checking, implemented in Lyra, Leo, and Leo45, permits arbitrary overlap between cells, and works directly on the hierarchy specified by the user.

Cell interaction regions are computed on the fly and checked flat. Arrays are handled specially, to minimize interaction check overhead. This approach allows fast checking of well structured designs, and degrades steadily as the amount of cell overlap increases. (Poorly structured designs can take longer to check hierarchically than flat, due to a huge amount of interaction checking.) Designers are permitted flexibility in handling shared buses, edges of arrays and global wiring.

There are only a couple of incremental design rule checkers I am aware of: The Leo45 (and Leo) systems, designed as part of this research, and the Magic design rule checker recently developed by George Taylor, and John Ousterhout. In Leo45 only cells that have been modified since the last check, and interactions involving those cells are rechecked. This is achieved by modifying the hierarchical algorithm to take into account modification- and check-dates on cells. The Magic system goes one step further, by taggin each each design modification as it is entered and checking them independently, keeping users continually up to date. Both systems provide much more interactive design rule checking than traditional systems, giving users more timely information on design rule violations, allowing errors to be more easily fixed, and encouraging more experimentation and refinement at the late stage of the design cycle.

8.6. References

[Newell & Martin 1983]

M.E. Newell & D.T. Fitzpatrick, "Exploiting Structure in Integrated Circuit Design Analysis," *Proc. Conference on Advanced Research in VLSI*, MIT, 1982, pp84-92.

[Ousterhout 1984]

J.K. Ousterhout, "The User Interface and Implementation of an IC Layout Editor," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-3, No. 3, July 1984, pp. 242-249.

[Scheffer 1983]

L.K Scheffer, *The Use of Strict Hierarchy for Verification of Intergrated Circuits,* PhD Thesis, Stanford University, May 1983.

[Taylor & Ousterhout 1984]

G.S. Taylor & J.K. Ousterhout, "Magic's Incremental Design-Rule Checker," *Proc. 21st Design Automation Conference,* June, 1984, pp. 160-165.

[Whitney 1983]
T. Whitney, Personal Communication, California Institute of Technology, Pasadena, California, 1983.

CHAPTER 9

Measurements

9.1. Introduction

This chapter presents performance figures for a number of design rule checkers. Measurements are also presented that relate to various topics discussed in the previous chapters. These include numbers on hierarchical/incremental checking, data organization, layer expression evaluation, constraint processing and rule indexing.

Most of the measurements were done on three corner-based checkers: Lyra, Leo, and Leo45. Recall that Lyra, the original corner-based design rule checker, is coded in Lisp and uses a two-dimensional bin structure to organize data. Leo and Leo45, belonging to Metheus Corporation, are coded in C and use a scanline organization. Lyra and Leo are strictly manhattan. Leo45 can handle 45-degree angles as well. All three programs are capable of hierarchical checking. Leo and Leo45 also check incrementally.

Performance figures for a number of other DRC's are also given. These include the NCA and ECAD commercial region-based systems, Mart, a corner-based DRC written by Mark Shand at CSIRO in Australia, the Magic DRC, which uses an edge-based algorithm similar to the corner-based approach, and Baker's pixel-based DRC. More details on these systems are provided in Chapters 4 and 7.

Most benchmarks were done on VAX-11/780's running Berkeley 4.2 BSD Unix. Some benchmarks were done on the Metheus λ, a 68000 based workstation running a port of 4.1c BSD Unix. The Delay, 32plus and Ioc chips, provided by Mark Shand, and the RiscI processor chip from Berkeley were used for the benchmarks. These designs range in size from 484 transistors (Delay) to 44,000 transistors (RiscI). All of them have a significant amount of hierarchical structure but only the RiscI chip contains explicit arrays. All use the Mead-

Conway nMOS design rules. The channel width is 5 microns, which corresponds to $\lambda = 2.5$ microns, and each design is resolved by a $\frac{1}{2}\lambda$ grid, i.e., by pixels of dimension $\frac{1}{2}\lambda$. Plots, and detailed statistics on the benchmark designs are given in the appendix.

9.2. Raw Performance

This section gives timings and check rates for nonhierarchical, nonincremental checks of Mead-Conway nMOS designs. Figure 9.1 gives timings for Shand's examples. The numbers were obtained from actual runs on lightly-loaded VAX-11/780's, except that the Magic time for Ioc was extrapolated from the smaller examples. Magic could not check Ioc (nonhierarchically) because of memory limitations.

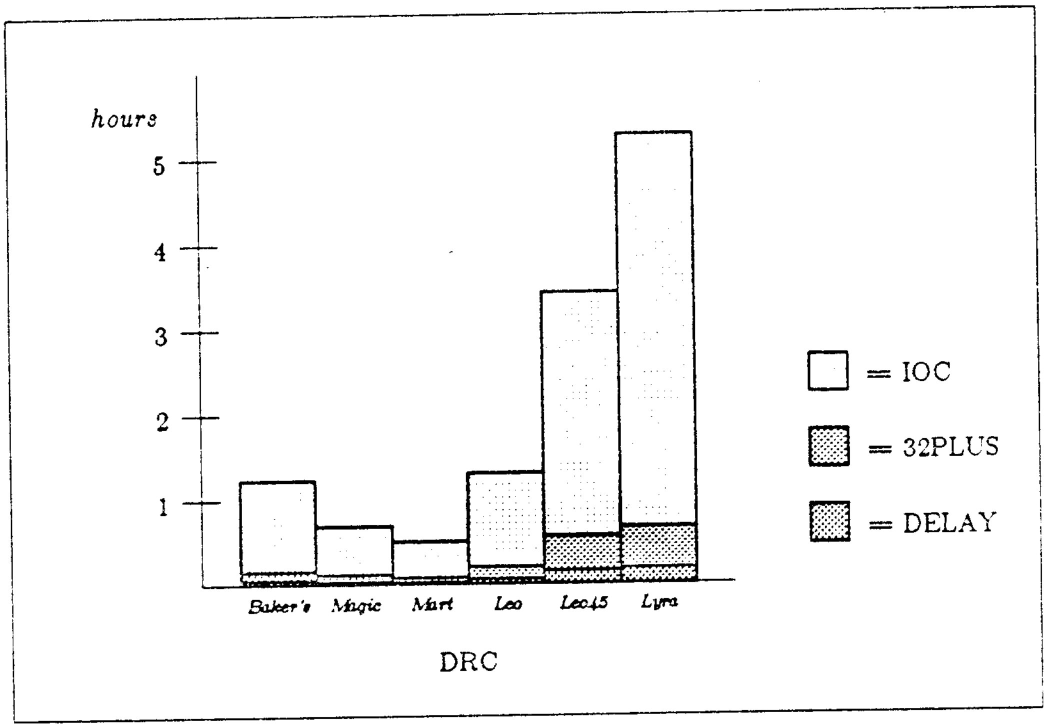


Figure 9.1. - Raw Performance on Shand's Examples. The chart shows run times for a number of systems checking Shand's three examples on VAX-11/780's. All checks were nonhierarchical and nonincremental. Magic could not check Ioc nonhierarchically because of memory limitations; the time given on the chart was extrapolated from smaller benchmarks.

Figure 9.2 gives check-rates for a number of systems. The numbers are for Mead-Conway designs on VAX-11/780's. The benchmarks these numbers were based on were on a number of machines. VAX-11/780 rates were computed based on the relative speed of the machines. The NCA and ECAD numbers correspond to about 6/84. Their current systems are probably faster.

The numbers in this section give a broad perspective on the raw performance of DRC's, but it is dangerous to give them too much weight. There are several problems with comparing DRC's in this way. First, the checkers are not all doing the same job. Baker's DRC, for example, does not check all the rules (e.g. the implant rules are not checked), while Mart, NCA, and ECAD go the exra mile and do the node-extraction necessary for accurate checking of internode spacing rules. (The DRC part of Mart also serves as the first pass for circuit extraction.) The NCA and ECAD systems do accurate diagonal checks and have

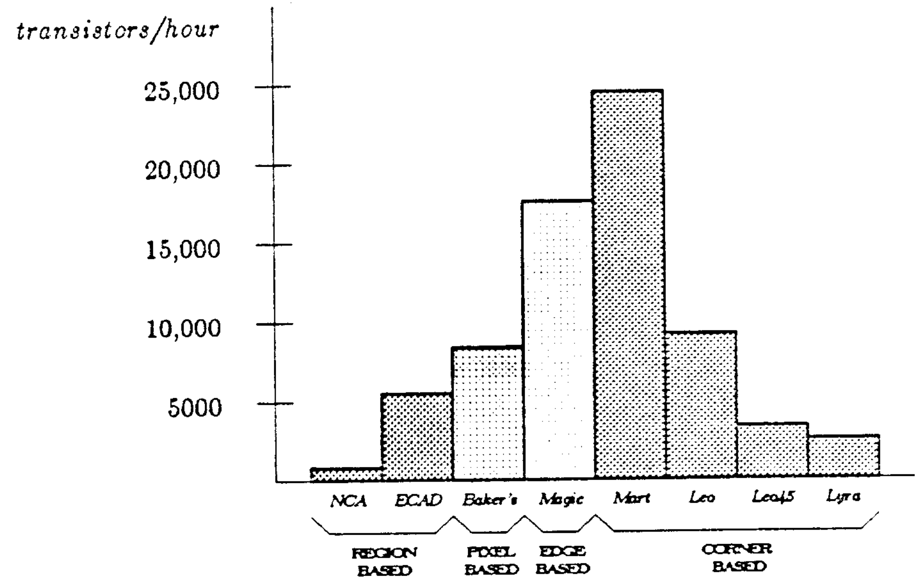


Figure 9.2. - Check Rate. This chart compares check rates (transistors/hour) for a number of systems. The numbers are for nonhierarchical nonincremental checks of Mead-Conway nMOS designs on a VAX-11/780. Some of the numbers estimates based on benchmarks on different machines and/or different rulesets.

provisions to handle all-angle data. Excluding NCA and ECAD, Leo45 checks diagonal distances much more accurately than the other systems. Second, those DRC's with hierarchical and incremental checking capability (Lyra, Leo, Leo45 and Magic) perform much better in practice than suggested by these *flat* numbers.

The above data is particularly misleading with respect to Magic. In Magic, contacts are normally explicitly specified by the user and stored in the database. Similarly transistors are identified and explicitly stored in the database at user entry time. In the Magic benchmarks reported above, 85% of the time was devoted to reading in the maskdata (CIF) and performing the logical operations necessary to convert contacts and transistors to internal form; checking designs already in Magic format is much faster. The Magic DRC runs in the background throughout an edit session, and generally appears instantaneous to the user.

9.3. Hierarchical and Incremental Checking

As discussed in Chapter 8, hierarchical processing, particularly when done incrementally, can greatly improve the effective performance of a DRC system. This section gives statistics on hierarchical and incremental checks done by Leo45 on the Metheus λ workstation. Hierarchical checking is done by verifying each cell separately and then rechecking all regions where cells interact. Incremental checking in Leo45 is done on a per cell basis: only those cells changed since the last check, and the interaction regions involving those cells, are rechecked.

Figure 9.3 gives timings for the four example designs. The first two columns give the times required for flat, and complete hierarchical checks, respectively. The third column gives the time required for an incremental check after modification of the top-level cell, and the fourth column gives the time required for an incremental check after a leaf cell has been modified. This chart dramatically illustrates the advantages of incremental checking. The only exception is the small Delay example, where a change to the top-level cell required the majority of the design to be rechecked. The chart also shows how the advantages of hierarchical checking alone are variable: hierarchically checking Delay was actually slower, while hierarchically checking RiscI resulted in an almost sixfold speedup.

Additional statistics on the hierarchical checking of these designs are given in Table 9.1. The regularity factor, large in all four of the example designs, gives the speedup that would be obtained from ideal hierarchical checking involving no overhead from cell interactions. The regularity factor is defined as the ratio of the number of rectangles in the fully instantiated design to the number in the hierarchical representation. Overhead is measured as the ratio of

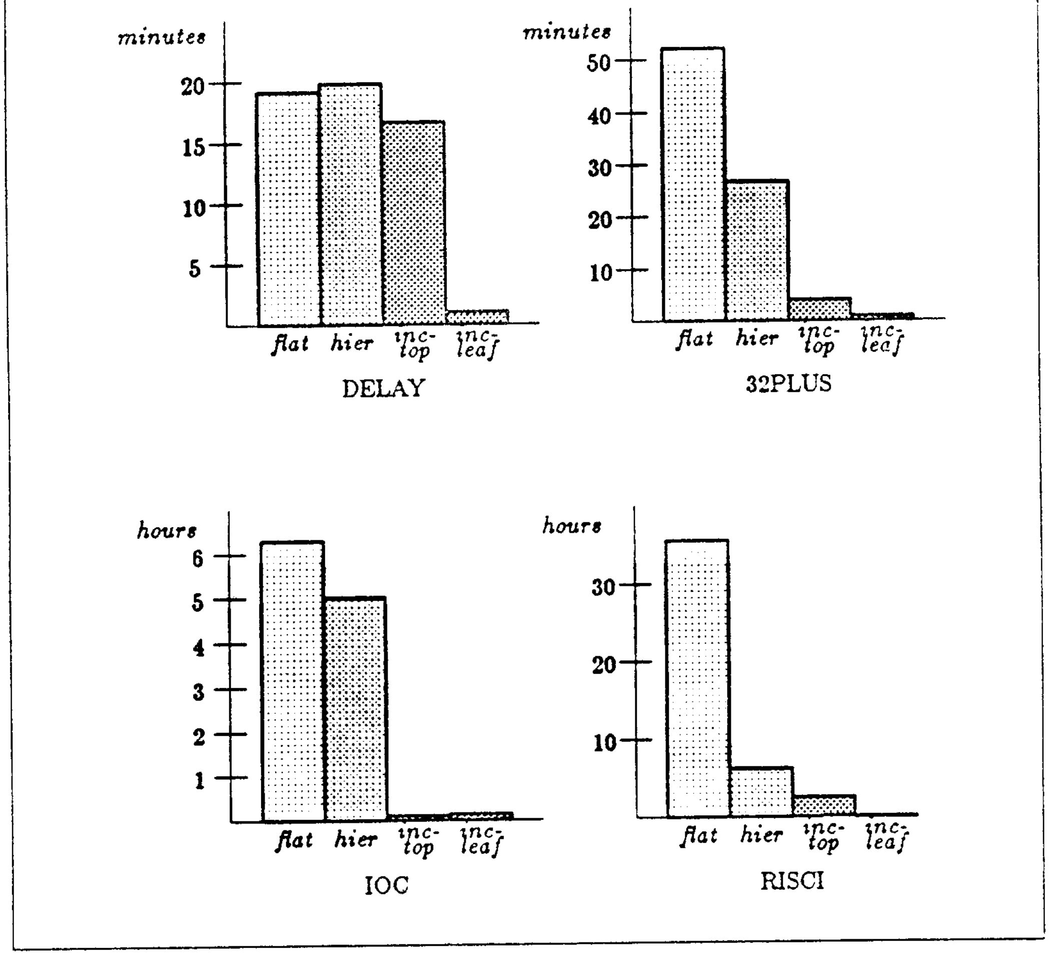


Figure 9.3. - Hierarchical & Incremental Performance. This chart compares the flat, hierarchical, and incremental performance of Leo45. Two incremental times are given, one for a change to the top-level cell, and one for a change to a leaf cell. The benchmarks were run on a Metheus λ.

the total area of interaction checks to the area of the design. The numbers in the table show that overhead is often large in practice, and that it critically effects hierarchical performance.

The efficiency of hierarchical checking is also affected by how fragmented the interaction regions are: because of overhead in starting up a check, one large check can be done more quickly than several smaller checks summing to the same area. The average size of interaction regions given in Table 9.1 provides a measure of fragmentation. Note that performance does appear to be related to this figure.

Fragmentation of interaction regions is minimized in Leo45 by merging adjacent or overlapping check regions before doing the checks. The effectiveness of this process is illustrated in Figure 9.4. Notice that merging not only reduces the number of interaction regions, but also their total area. This is because there is significant overlap between the original check regions.

9.4. Sensitivity of Check Rate to Design Density

In pixel-based systems the processing rate per pixel element is approximately constant. Thus for a given resolution (pixels/area) the processing rate (time/area) is approximately constant. In corner-based systems, in contrast, the processing rate is approximately

Statistics on Hierarchical Checking					
design	speedup	overhead	average interaction area	regularity	arrays
Delay	.97	52%	253	19.4	NO
32plus	1.9	36%	461	12.1	NO
Ioc	1.3	33%	251	8.3	NO
RiscI	5.8	11%	709	16.6	YES

Table 9.1 These statistics were collected for Leo45 running on a VAX-11/780. The columns give, respectively, the speedup factor for hierarchical checking over flat, the area of interaction checks as a percentage of the total chip area, the average area of interaction regions, the ratio of rectangles in the fully instantiated design to rectanges in the hierarchical representation, and whether the design contains arrays.

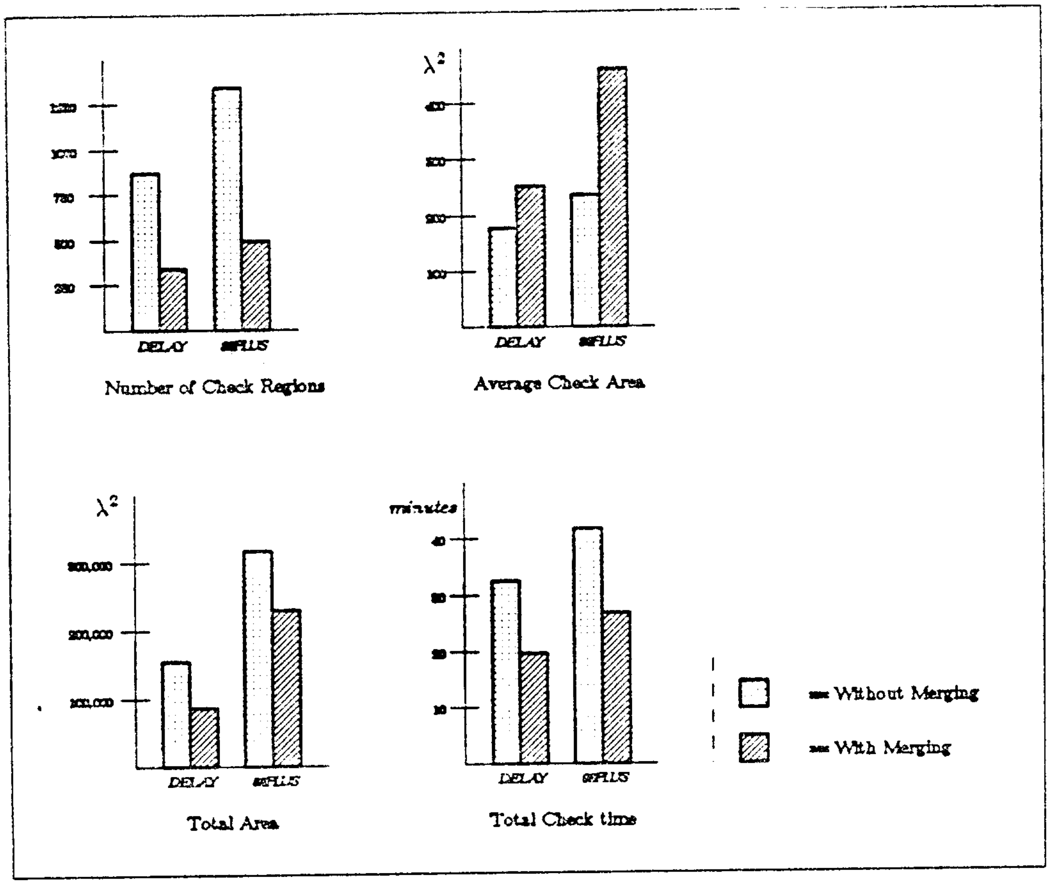


Figure 9.4. - Importance of Merging Check Regions Merging adjacent and/or overlapping check regions prior to checking reduces the total number of regions to check, increases the average size of check regions, and decreases the total area to check. The combination of less setup overhead for checks (snce the average check region is larger) and less total area to check, results in considerable savings in total check time.

proportional to the density of the layout and independent of resolution.

These relationships are illustrated in Figure 9.5. The first graph shows that the processing rate of Baker's pixel-based system is approximately constant across Shand's three examples. The similarity of the second and third graphs show that the processing rate of Leo is approximately proportional to layout density.

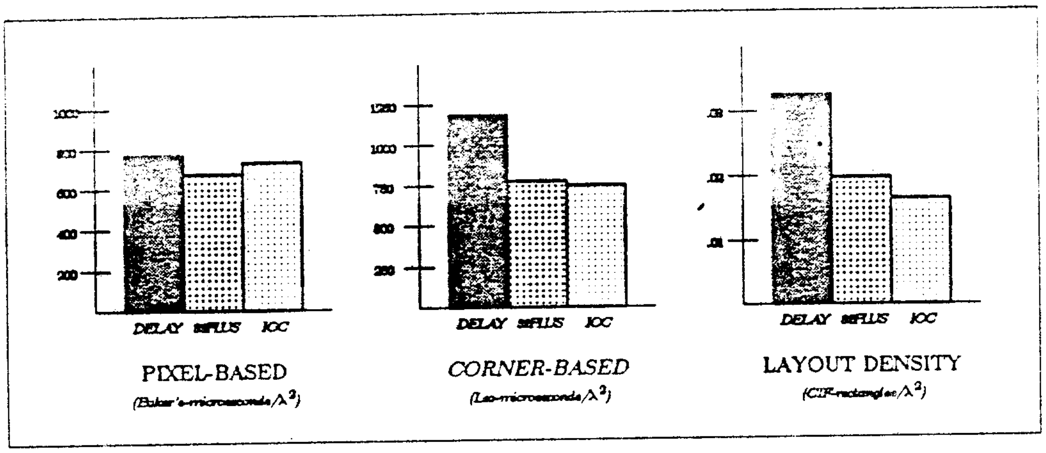


Figure 9.5. - Processing Rates and Layout Density. The above charts illustrate how the processing rate per unit area of pixel-based systems is approximately constant, while the check rate of corner-based systems is correlated with the layout denisity. The data is from benchmarks on VAX-11/780's.

9.5. Sensitivity to Data Organization

The efficiency of corner-based checking depends on the ability to quickly reference all features geometrically near a given point. There are many ways to organize data to allow quick local reference, and I do not know which is best, but there is no doubt that performance is sensitive to the details of data organization. This is illustrated in Figure 9.6 showing sensitivity to bin-size in Lyra.

9.6. Layer Expression Evaluation

This section presents statistics on layer expression evaluation for Lyra, Leo and Leo45. The statistics are based on runs on a VAX-11/780, the Delay example, and Mead-Conway rules. The statistics demonstrate the importance of efficient expression evaluation, and document the value of the bitmapped DNF method. Differences in the layer expression usage patterns of Lyra, Leo and Leo45 are also analyzed.

Efficient layer expression evaluation is important because of the sheer number of evaluations that must be performed during corner-based checking. Figure 9.7 gives numbers for Lyra, Leo and Leo45 on Shand's Delay example. Note that even for this small 484

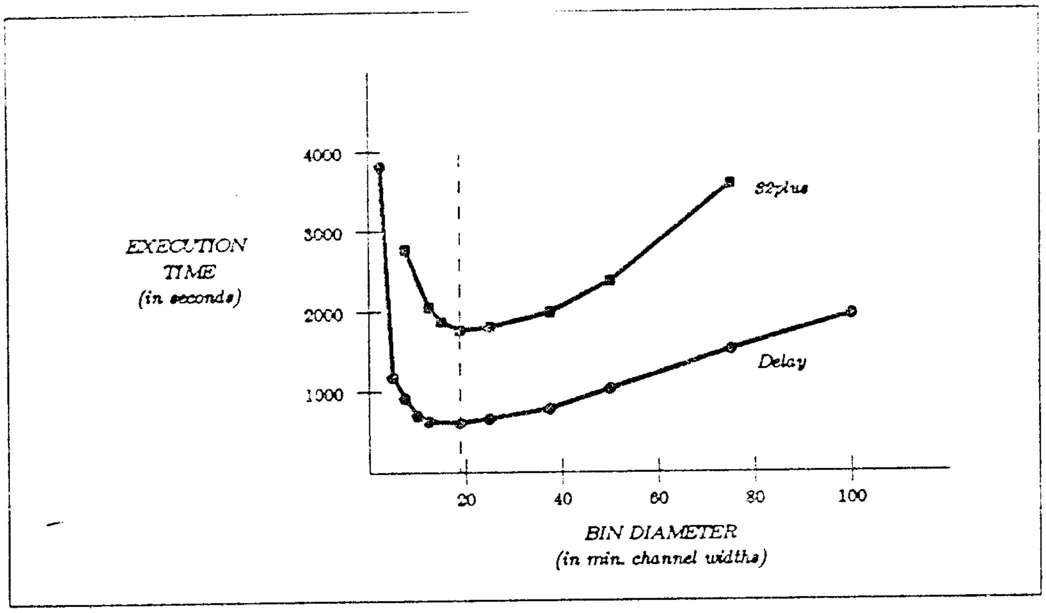


Figure 9.8. - Bins in Lyra. If too small a bin-size is used bin handling overhead becomes excessive. If too large a bin-size is used too much data must be searched at each corner. Thus the performance of Lyra is very sensitive to bin-size. In general the performance of a DRC is likely to be very sensitive to the details of the data organization employed. The above data is from benchmarks of Lyra on a VAX-11/780.

transistor design, millions of expression evaluations are required. Table 9.2 translates these numbers into rates.

Given the amazing number of expression evaluations required in corner-based checking, it is natural to ask what purposes they serve. The layer expression evaluations in Figure 9.7 are subdivided into those required for corner classification, (i.e., for determining which rules apply at the corners), for checking immediate conditions, and for checking regional conditions.

Corner classification in Leo and Leo45 require about three times as many expression evaluations as in Lyra. This is due partly to the different indexing methods used by Lyra and the Metheus systems. The edge-crossing method of Leo and Leo45 result in about 1.5 times as many layer candidates as the layers-present method of Lyra. This is exacerbated in Leo because false edges are not filtered out prior to indexing. Leo45 filters out false edges, but requires twice as many layer expression evaluations per layer candidate (one for each octant, rather than one per quadrant).

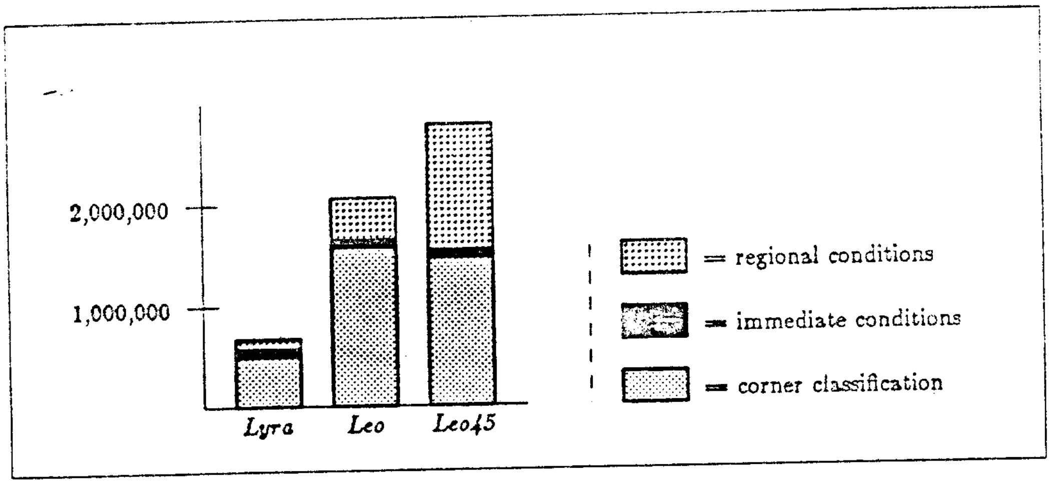


Figure 9.7. - Number of Layer Expression Evaluations. The chart shows the number of layer expressions required by three corner-based programs to check the 484-transistor Delay design. Evaluations are divided into those used for corner classification, immediate-condition evaluation, and region-conditon evaluation.

Layer Expression Rates				
program	exprs./λ^2	exprs./corner	exprs./rectangle	exprs./transistor
Lyra	4.07	20	150	1408
Leo	12.3	61	455	4267
Leo45	16.7	82	614	5761

Table 9.2 These expression evaluation rates were computed from a benchmark of Delay on a VAX-11/780.

The number of layer expression evaluations required for checking immediate conditions remains small and approximately constant across the three programs.

The number of evaluations employed during regional condition checking is small in Lyra, but grows by a factor of four for Leo, and by another factor of three for Leo45. The number for Lyra is small because layer expressions are evaluated only when mask features on layers *involved* in the expressions impinge on the condition regions. In most cases there will be either no or one impinging feature: none in spacing rules and one in width rules. The number of evaluations in Leo is greater because the scanline method used in Leo requires layer expression evaluations for *all* impinging features, regardless of layer. Leo45 requires still more layer expression evaluations because the more complex region shapes employed in this 45-

degree system are checked in pieces, and because extra layer expression evaluations are required to support inclusive sector edges.

Fortunately layer expression evaluation can be done quickly. In Lyra layer expression evaluation times range from 69.8 microseconds for the simplest layer expression to 85.5 microseconds for the most complex. They average 75.9 microseconds. Each layer expression is compiled as a separate function, so evaluation time is dominated by function call overhead. The bitmapped DNF method, recommended in Chapter 5, and employed by Leo and Leo45 is even faster. Evaluation times range from 20 microseconds to 32 microseconds, averaging 22 microseconds.

Figure 9.8 gives the time required for layer expression evaluation in the context of total processing time. The hypothetical performance of each system, using the alternate layer expression evaluation method is also shown. The chart shows that the compiled function method would consume over 50% of the processing time in Leo. The DNF method requires 23% of processing time in Leo, 12% in Leo45 and would require only 2% in Lyra. Thus it is apparent that the bitmapped DNF method of layer expression evaluation is fast enough not to dominate processing time in corner-based systems, but that significantly less efficient methods of layer expression evaluation would be too slow.

9.7. Region Condition Processing

Benchmarks on the Delay example show that corner-based systems spend a large part of their time processing constraints: 30% in Lyra, 56% in Leo and 75% in Leo45. This is shown graphically in Figure 9.9. Additional statistics are provided in Table 9.3.

Region condition processing is particularly slow in Leo45, because of the added complexity of the region shapes, and because of extra processing to support inclusive edges. (Notice that the support for inclusive edges reduces the total number of condition regions).

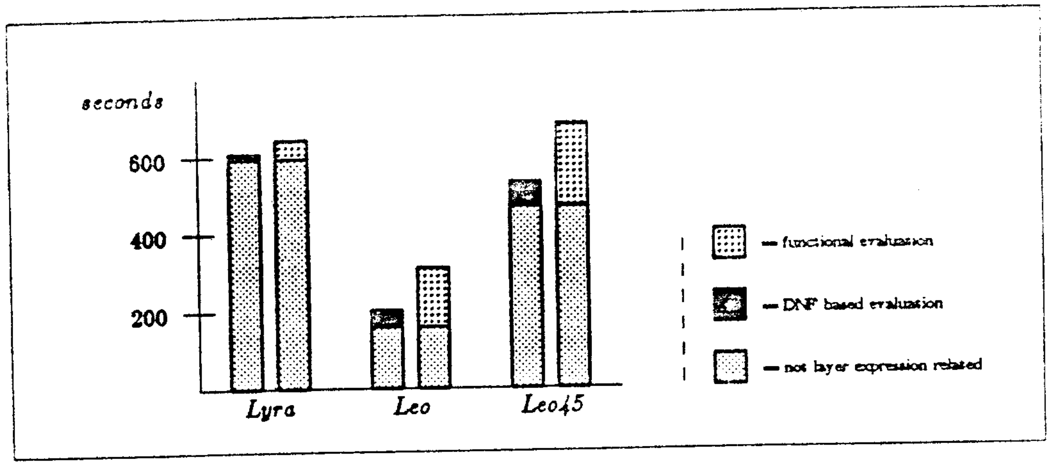


Figure 9.8. - Time Required for Layer Expression Evaluation. The chart contrasts the time required for layer expression evaluation in three programs with the time required for all other purposes. For each program two evaluation schemes are considered: function-based evaluation as used in Lyra, and DNF-based evaluation, as implemented in Leo and Leo45. The numbers come from benchmarks of Delay on a VAX-11/780. The time that would be required by each program for the alternate evaluation method was computed from a trace of layer expression evaluations for that program.

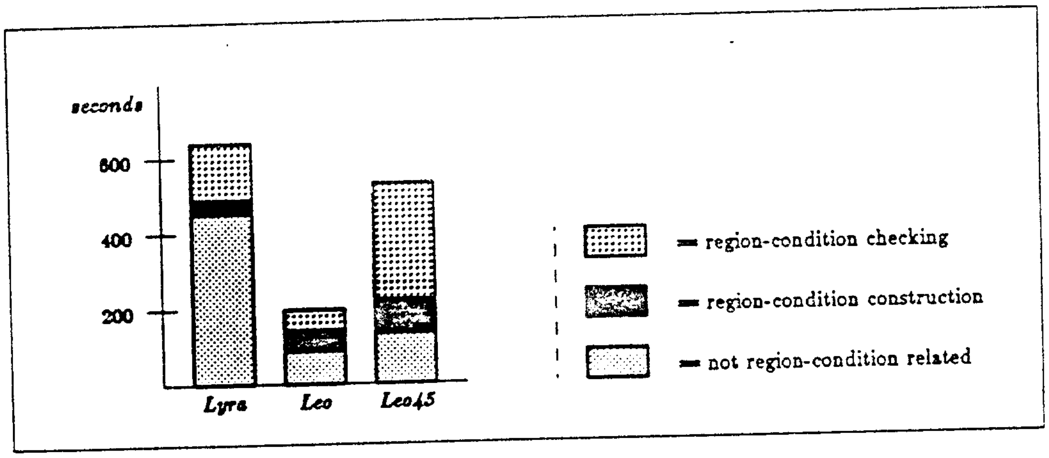


Figure 9.9. - Time Required for Region Condition Processing. The chart divides total check time for Delay on a VAX-11/780, into the time required for region-condition checking, the time required fo region-condition construction, and all other check time. Region conditon processing clearly dominates processing time in the 45-degree program, Leo45.

Statistics on Region-Condition Processing				
program	# conditions	conditions/sec.	# layer expressions	exprs./condition
Lyra	86,990	462	100,159	1.15
Leo	86,990	787	410,520	4.72
Leo45	50,272	127	1,239,369	24.65

Table 9.3 These statistics are for Delay on a VAX-11/780. The columns give the total number of region conditions checked, the average number of conditions checked per second, the total number of layer-expression evaluations required to process the conditions, and the average number of layer-expression evaluations per condition, respectively.

9.8. Rule Indexing

In corner-based systems, rules are indexed so that those applying to a given corner can be found quickly. Indexing is important because rulesets contain many rules, only a few of which apply at any given corner. This is illustrated by the numbers in Table 9.4. These numbers are based on checks of Delay by Lyra and Leo45. Note that, for Lyra, less than 5 of the 44 rules in the Mead-Conway ruleset are relevant at the average corner. The number of relevant rules is higher in Leo45, because rules for convex and concave corners are not separate. The number of corner-points varies slightly between the two programs, because different methods are used to filter out false corners.

The basic indexing strategy in corner-based systems is to group rules by the layer they apply to. Then given a corner-point, layer candidates are determined by some method and then tested to see if corners on those layers are present. The rules corresponding to the layers for which corners are actually present are then applied to those corners. Lyra generates

Statistics on Rule Application				
program	# of Corner Points	# of Rules	Maximum Rules (/Corner Point)	Average Rules (/Corner Point)
Lyra	20591	44	17	4.78
Leo45	19088	43	20	7.66

Table 9.4 These statisitics apply to checks of Delay. The columns give the total number of corner points processed, the total number of rules in the ruleset, the maximum number of rules applying at a single corner point, and the average number of rules applying at a corner point, respectively.

corner-points by the method suggested in Chapter 6, based on the mask-layers present at a corner-point. Leo45 uses a method based on edge intersections. Statistics for the two methods are presented in Table 9.5. The average number of layer-candidates for the edge-intersection scheme is about 1.5 times greater than in the layers-present method. But the number of layers actually applying is also higher.

Two measures of indexing effectiveness are the ratio of layers for which corners are actually present to the number of layer candidates, and the average number of false candidates per corner. These statistics are shown in Table 9.6. It is clear from these statistics that the layers-present method of Lyra is more effective than the edge-intersection method of Leo45. Both methods are significantly more effective than sequential testing of rules or sequential testing of all layers indexing rules. Rule indexing becomes more important as ruleset size and complexity increases.

Statistics on Indexing Methods			
program	Total Index Layers	Avg # of Candidates	Avg # that Apply
Lyra	15	6.12	1.87
Leo45	15	9.64	2.36

Table 9.5 Again, these statistics were obtained for Delay. The columns give the number of distinct index layers used to index the rules, the average number of index layers considered at each corner point, and the average number of index layers for which corners are actually present at a corner point, respectively.

Indexing Efficiency		
program	# that apply / candidate	#false layers / corner
Lyra	31%	4.25
Leo45	24%	7.28

Table 9.6 These statistics, obtained for Delay, are measures of the efficiency of indexing. The first column gives the percentage of index-layer candidates that actually apply. The second column gives the average number of layer candidates that do not apply at a corner point.

CHAPTER 10

Summary

This chapter briefly summarizes the major points of the thesis.

10.1. Previous Approaches to Design Rule Checking

Design rules specify minimum tolerances for topological relationships on and between mask regions. These include minimum width, spacing, enclosure and extension tolerances. Rules can apply to regions formed by combinations of mask layers (such as transistor gates) as well as to regions on the individual mask layers. Some rules are unconditional, that is they apply equally to all regions on the specified layers (or layer combinations). Others are conditional. Conditional rules give tolerances that apply only to selected regions on the given layers. Conditional rules can depend on the presence of nearby mask features, the length of conducting lines, connectivity information, and even the intended function of the features involved.

The traditional, almost universally used, approach to design rule checking use sequences of region operations. Each region operation takes one or two layers as input and generates an (intermediate) output layer. Each design rule is coded as a sequence of *selection* operations, isolating the regions to which the tolerance applies, followed by a tolerance check operation. The region operations in a typical system include several tolerance checks, boolean operations, sizing operations (to grow or shrink regions) and topological operations that for example select all regions partially overlapped by a specified layer. Other operations allow selections based on connectivity, or node labels. The main problem with this approach is that it involves a very large amount of data manipulation. Each design rule involves several operations, and the more complex ones can involve dozens. In checking VLSI designs, each region-operation in the sequence involves hundreds of thousands of input and output edges. Relatively little design rule checking is done per I/O operation, so the overall result is a slow I/O bound system.

Some experimentation has been done with pixel-based systems employing special purpose hardware. These systems represent a design in terms of a fine grid of pixels where each pixel is tagged with the mask layers present in it. Such a representation involves an even larger amount of data, and hence has the same problem that the region-operation approach has: relatively simple processing must be done on a very large amount of data. Of course the special purpose hardware might be built with sufficient bandwidth to handle such data quickly. However the ballooning of the design into such a large amount of data seems to be inherently unbalanced. The problem of I/O to and from the special purpose hardware remains, and the hardware itself is likely to be expensive. No system of this type has yet been completed.

10.2. The Corner-Based Approach

The corner-based approach solves the I/O bottleneck problem plaguing most design rule check systems. Corner-based checking uses pattern-directed rule application. Patterns at each corner in a design determine which rules apply, and hence which tolerances are to be checked. The rule patterns are indexed so that the rules applying to a given corner can be quickly determined (i.e. much faster than a linear search through the rules). Just *one pass* is made through the design data, and *no intermediate layers* are generated.

Corner-based checking can be used in general settings. Though current corner-based implementation are restricted to manhattan or 45-degree mask data, their is nothing intrinsic in the approach that limits it to these settings. Corner-based rules directly replace the tolerance check and boolean operations of the traditional operation-based system. In addition multiple condition rules can often (but not always) establish context that would otherwise require sizing operations. Corner-based checking alone can not replace nonlocal region operations, such as topological operations, or operations for deriving connectivity. If rules requiring such operations are to be checked, a hybrid system is required, where region-operations are used to establish nonlocal context prior to the corner-based checking.

Corner-based systems are quite flexible. Variants of rules, that would require the coding of a new operation in an operation-based system, can often be captured by simply modifying the corner-based rule specification. Another advantage of the corner-based approach is that it can easily handle anisotropic rules that are difficult or impossible to handle in region-operation based systems. This is because conditions in corner-based rules can be combined to establish directional context.

Tolerance checks in corner-based systems are done differently from tolerance checks in region-operation systems: they involve distances between corner-points and the boundaries of regions, rather than distances between pairs of edges. Such point/edge tolerance checking naturally identifies violations with points in the design. This makes corner-based checking particularly well suited for hierarchical and incremental systems, where piece-wise checking is required.

10.3. Hierarchical and Incremental Checking

Hierarchical checking avoids redundant checking by checking each cell only once regardless of how many times it is repeated in a design. In addition to checking cells, interactions between cells must be checked. The hierarchical algorithm I used in Lyra and Leo has three steps:

i. Check subcells recursively.

ii. Check all mask features in current cell

iii. Find and recheck regions where cells interact.

No special restrictions on cell interaction are required. Arrays of cells are checked specially: instead of checking all cell interactions in an array, only representative interactions are checked. The algorithm is effective for checking hierarchical designs with moderate overlap between cells. The special treatment of arrays makes for particularly efficient checking, since most of the regularity in VLSI designs comes in the form of arrays. However the algorithm handles designs with large amounts of overlap poorly. Such designs can take several times longer to check hierarchically than flat. This is because mask features in regions of overlap are checked multiply, first as part of the overlapping cells, and then again when the interaction resulting from the overlap is checked.

The Leo and Leo45 programs were also made incremental. Incremental checking avoids redundancy by only checking cells that have been modified since the last design rule check. It was surprisingly easy to adapt the hierarchical algorithm to be incremental as well. Hierarchical/incremental checking after small changes to a large design typically takes from 5% to less than 1% of the time for a full flat check. This makes it possible to run design rule checks frequently during the design process, and hence to catch violations early when they are still easy to fix.

10.4. Implementations

The viability of corner-based checking, and related ideas developed in this thesis, have been demonstrated by a number of systems. Lyra, my initial, manhattan, corner-based system, has been used on a number of university and industrial projects involving several, Mead-Conway style, nMOS and CMOS rulesets. Lyra was also the first hierarchical design rule checker, and demonstrated that hierarchical checking can be effective without special restrictions on cell interactions. Leo, a second implementation I developed in conjunction with Metheus corporation, showed that corner-based checking can be fast. Leo also demonstrated the feasibility and usefulness of incremental checking. Mart, a corner-based design rule checker, developed by Bruce Nelson & Mark Shand of CSIRO, handles somewhat more complex rules than Lyra, and, in terms of raw speed, is probably the world's fastest design rule checker.

Two other systems, employ ideas presented in this thesis. Intel recently developed a region-based design rule checker that uses corner-based style (i.e. point/edge) tolerance checks. This style of tolerance checking was chosen to facilitate piecewise processing. Finally the, edge-based, Magic design rule checker, recently developed by John Ousterhout and George Taylor at Berkeley, very effectively uses pattern-directed rule application. The Magic design rule checker runs in the background throughout edit sessions, and provides virtually instantaneous feedback on design rule violations.

APPENDIX A

Benchmark Designs

This appendix contains statistics and a complete set of plots for the benchmark designs used for most of the measurements presented in Chapter 9.

A.1 Statistics

Four benchmarks are used. The Delay, 32plus and Ioc designs, provided by Mark Shand, and the RiscI processor design from Berkeley. These designs range in size from 484 transistors (Delay) to 44,000 transistors (RiscI). All of them have a significant amount of hierarchical structure but only the RiscI chip contains explicit arrays. All use the Mead-Conway nMOS design rules. The channel width is 5 microns, which corresponds to $\lambda = 2.5$ microns, and each design is resolved by a $\frac{1}{2}\lambda$ grid, i.e., by pixels of dimension $\frac{1}{2}\lambda$. Tables A.1, A.2 and A.3 give detailed size, density, and hierarchical statistics on the four designs.

Size of Designs

design	area (λ^2)	area (mil^2)	hierarchical ($CIF-kbytes$)	flat	transistors	rectangles drawn	rectangles
Delay	167,433	67,513	13.6	155	484	282	5475
32plus	637,304	256,977	34.6	347	1,369	1,038	12,540
Ioc	5,290,000	2,133,060	367	2,501	7,286	10,584	87,395
RiscI	20,002,750	8,065,608	610	13,991	44,000	32,158	532,941

Table A.1

Density of Designs		
design	rectangles/λ^2	transistors/λ^2
Delay	.033	.0023
32plus	.020	.0021
Ioc	.016	.0013
RiscI	.027	.0022

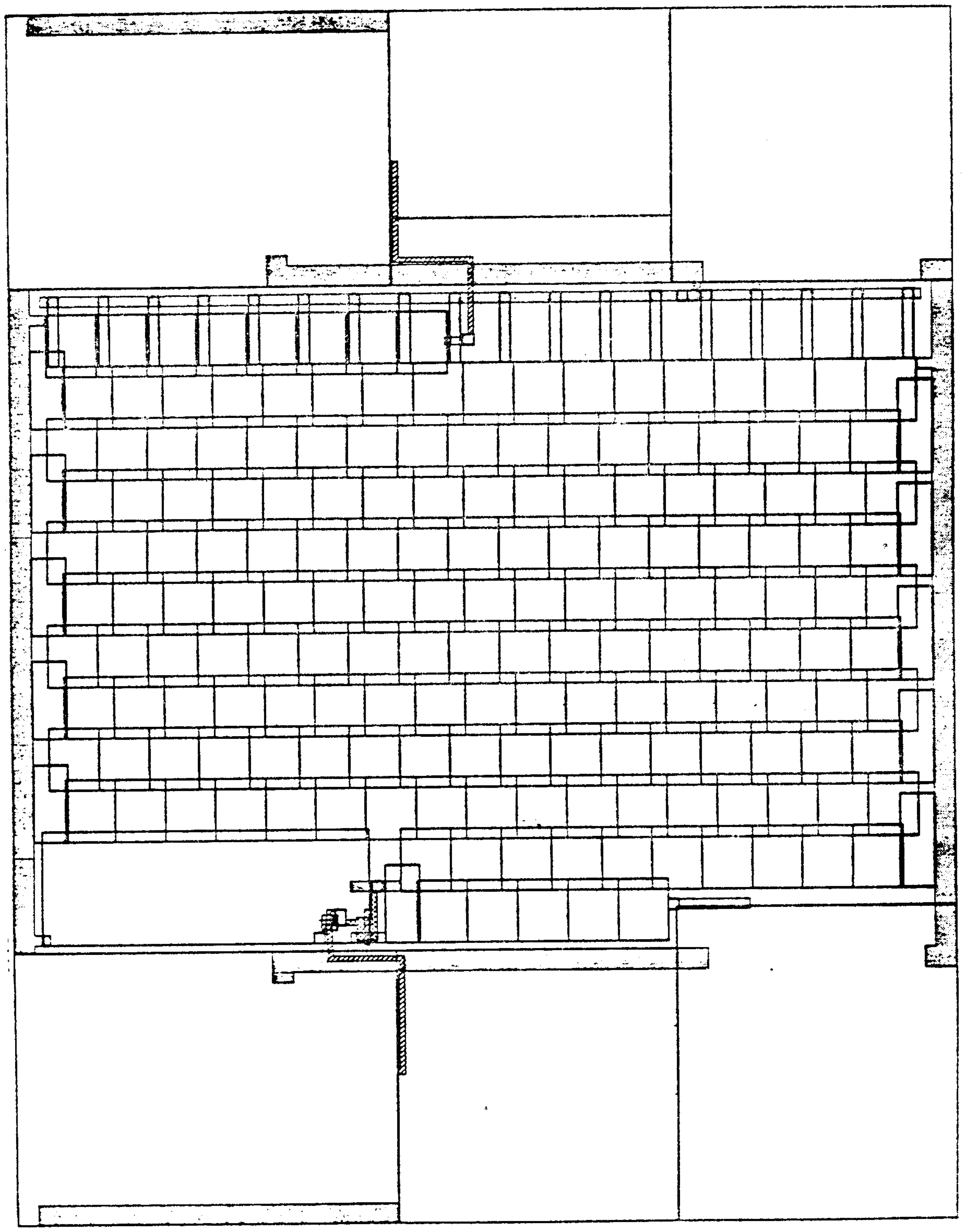
Table A.2

Statistics on Hierarchy				
design	% of data in top cell	regularity (rects/drawn-rect)	cells	hierarchy depth
Delay	42	19.4	16	4
32plus	1.6	12.1	44	6
Ioc	.05	8.3	346	11
RiscI	26	16.6	202	7

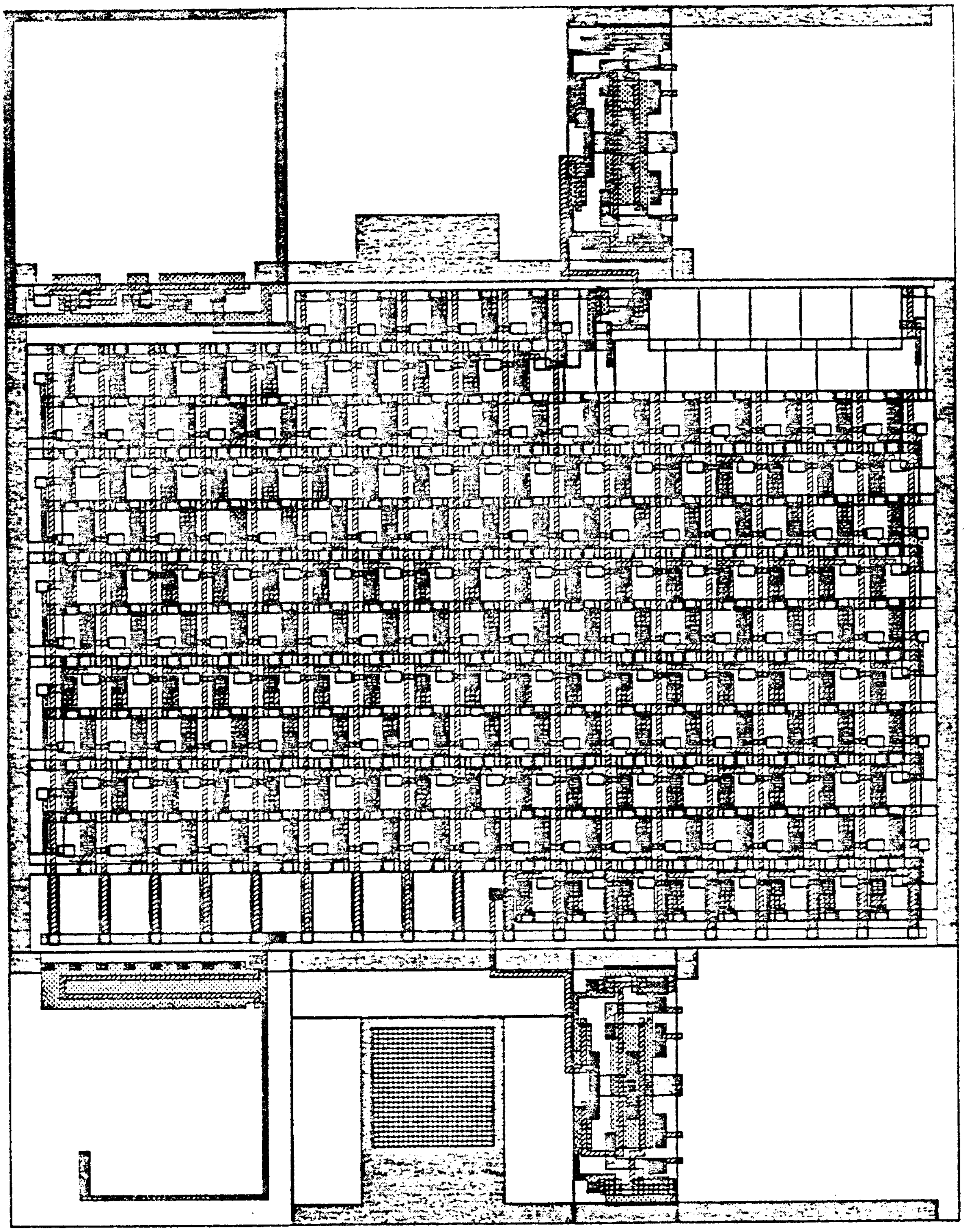
Table A.3

A.2 Plots

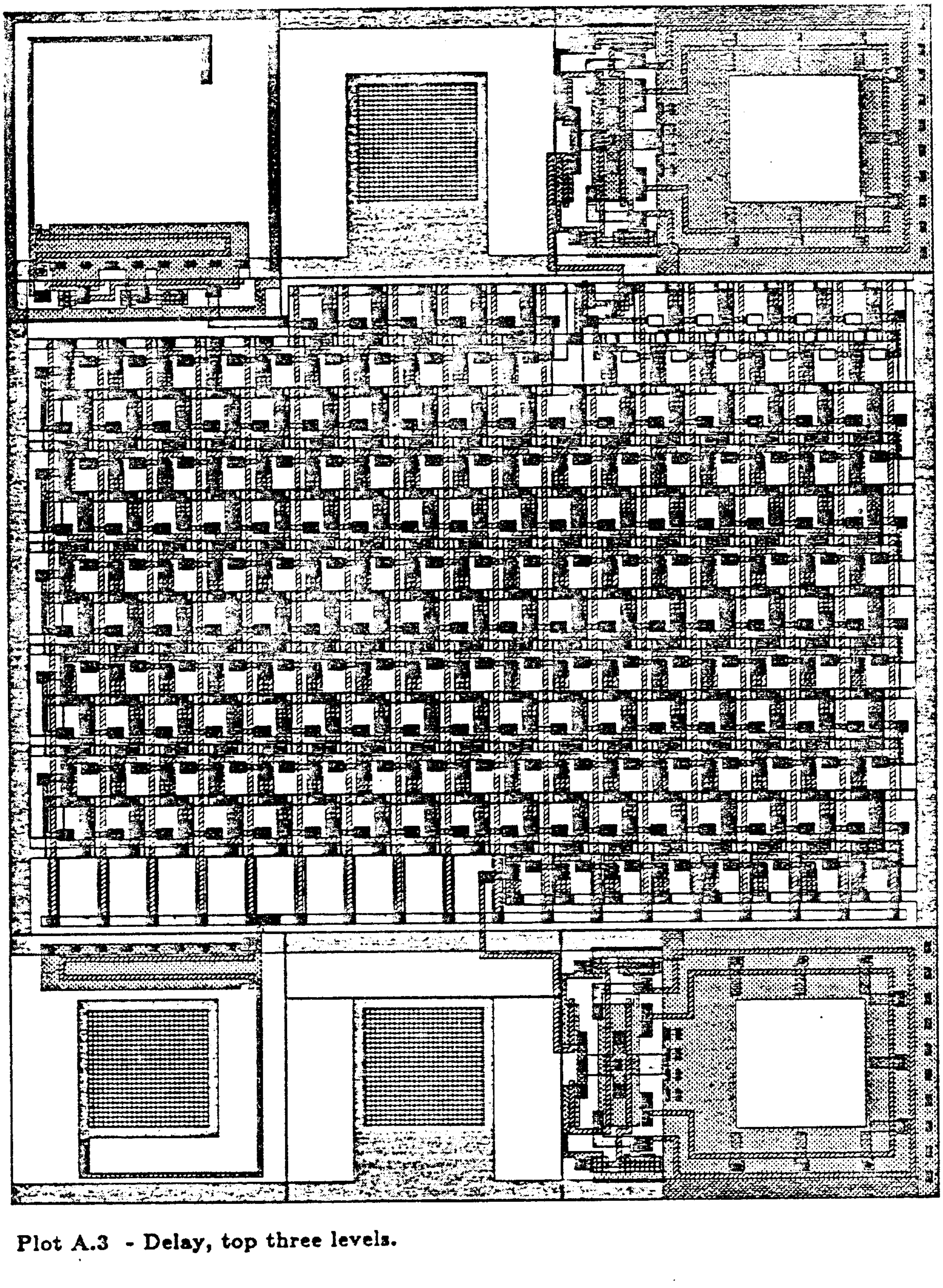
Plots for the four designs are given below. Each design is plotted to successively deeper levels of its cell hierarchy (once for each level).



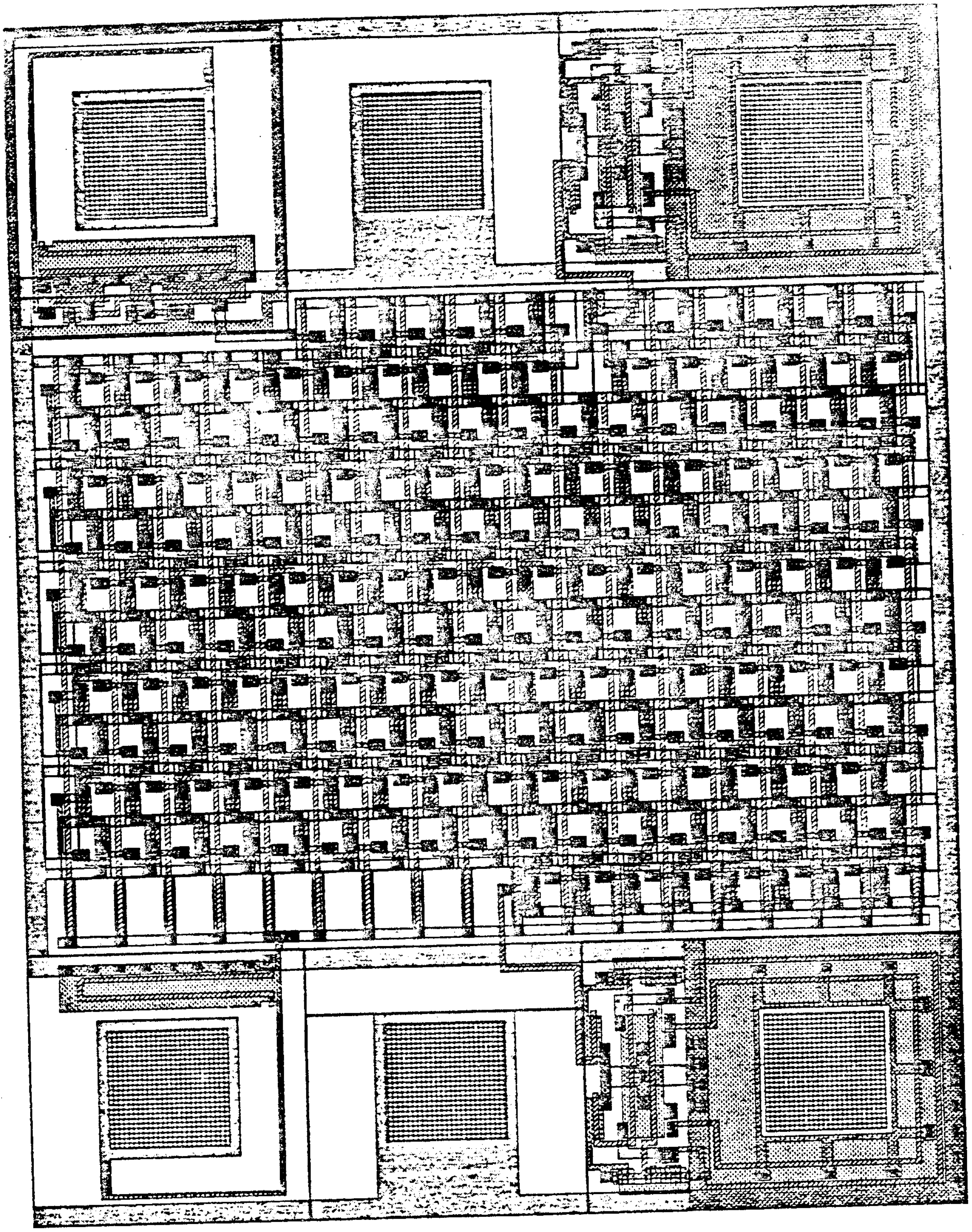
Plot A.1 - Delay, top level.



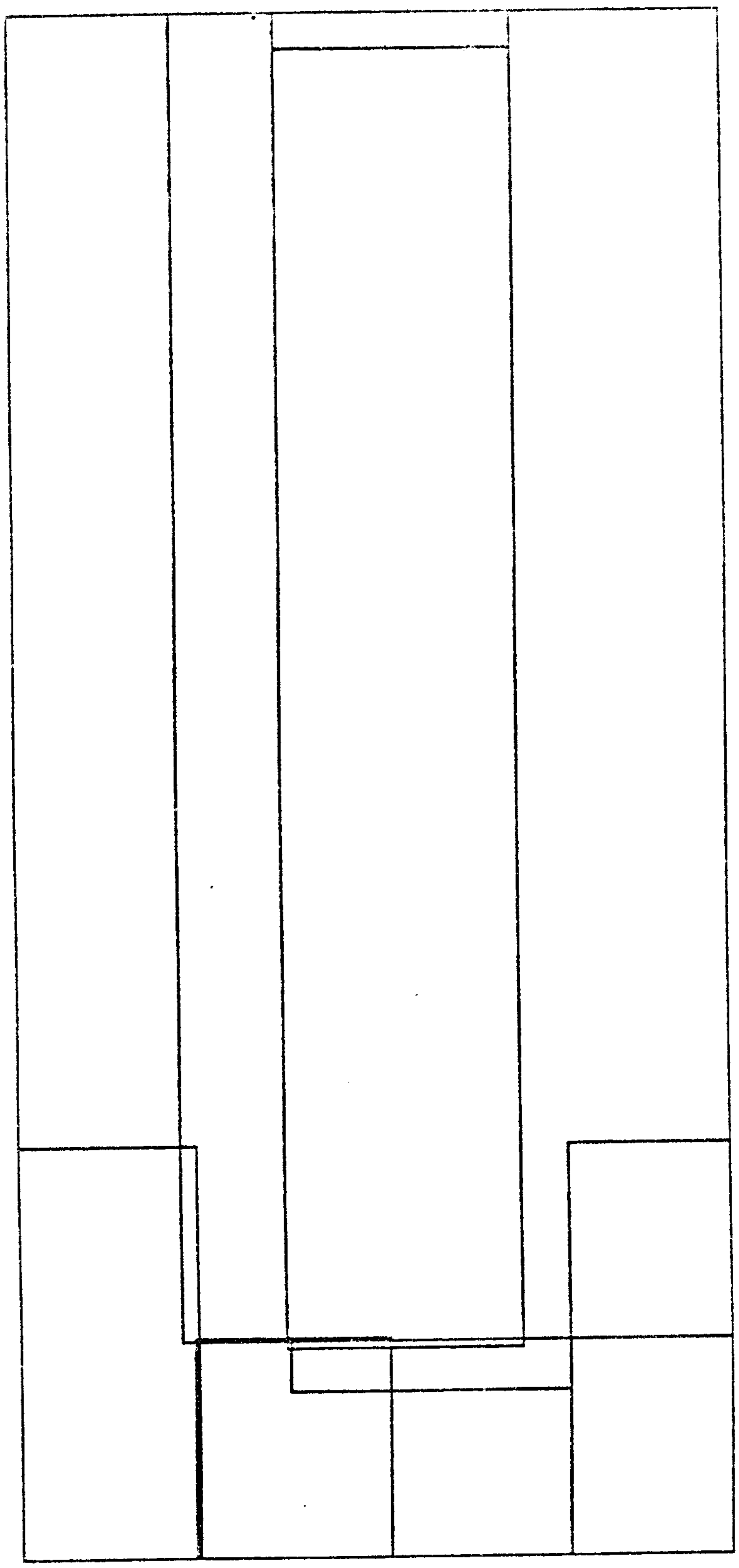
Plot A.2 - Delay, top two levels.



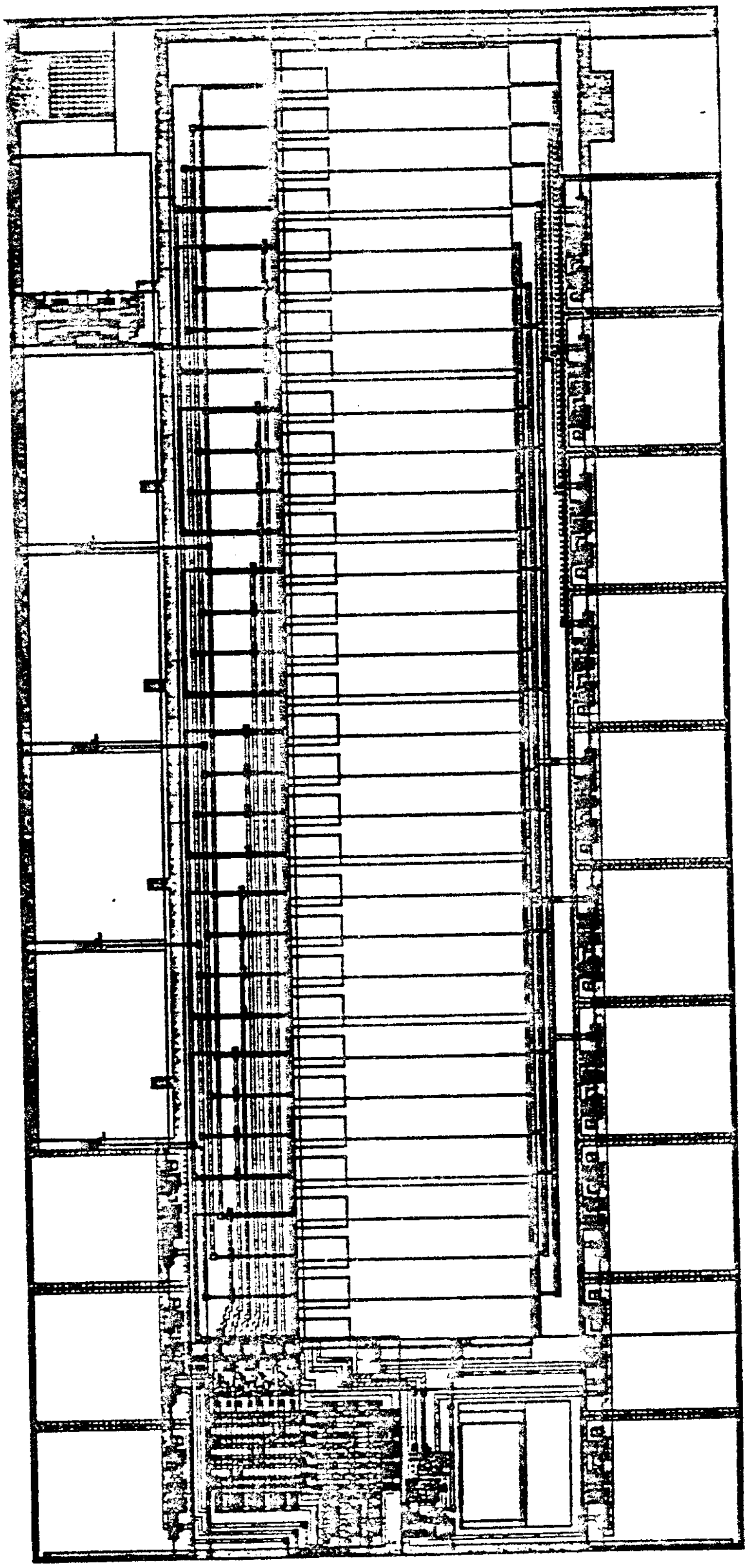
Plot A.3 - Delay, top three levels.



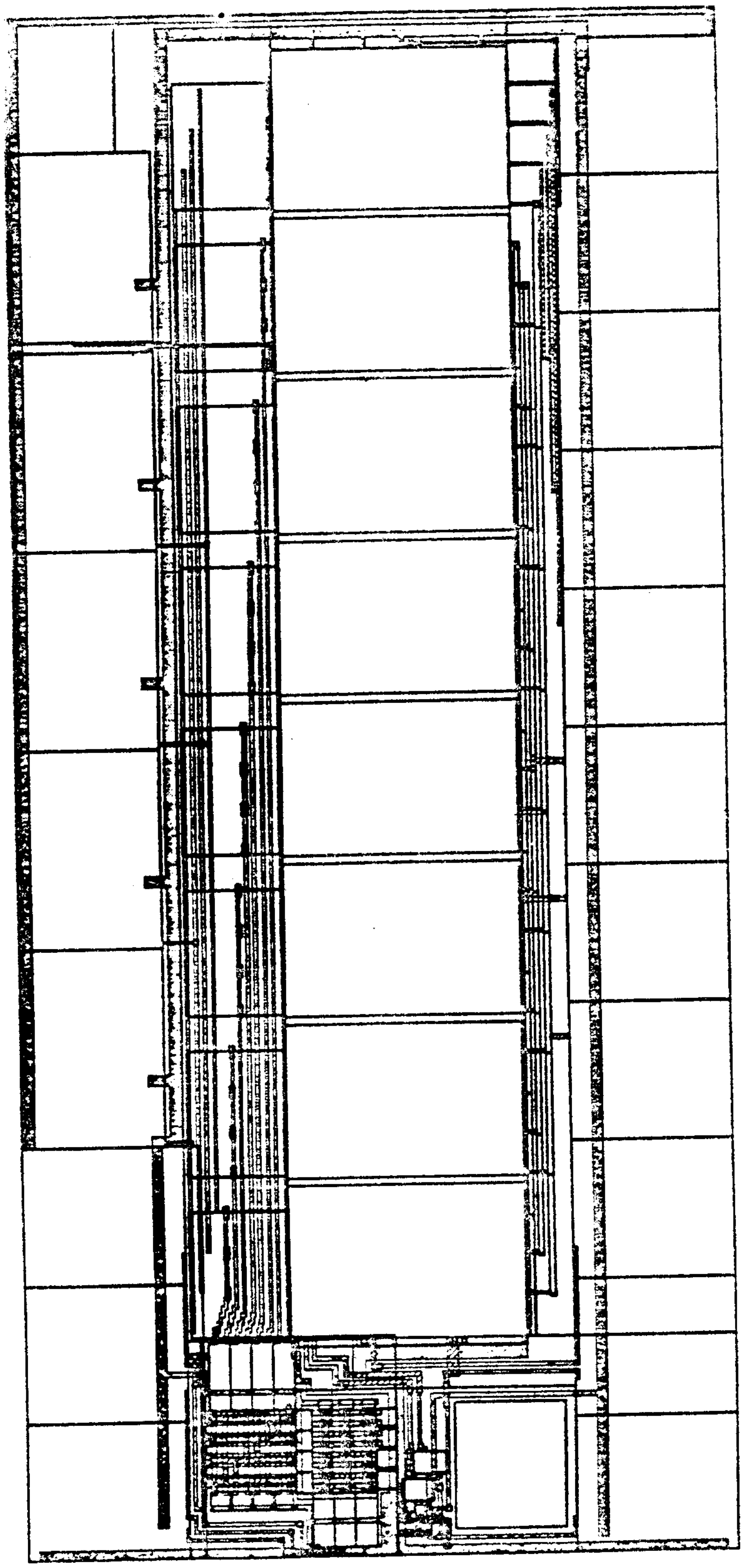
Plot A.4 - Delay, top four levels (complete).



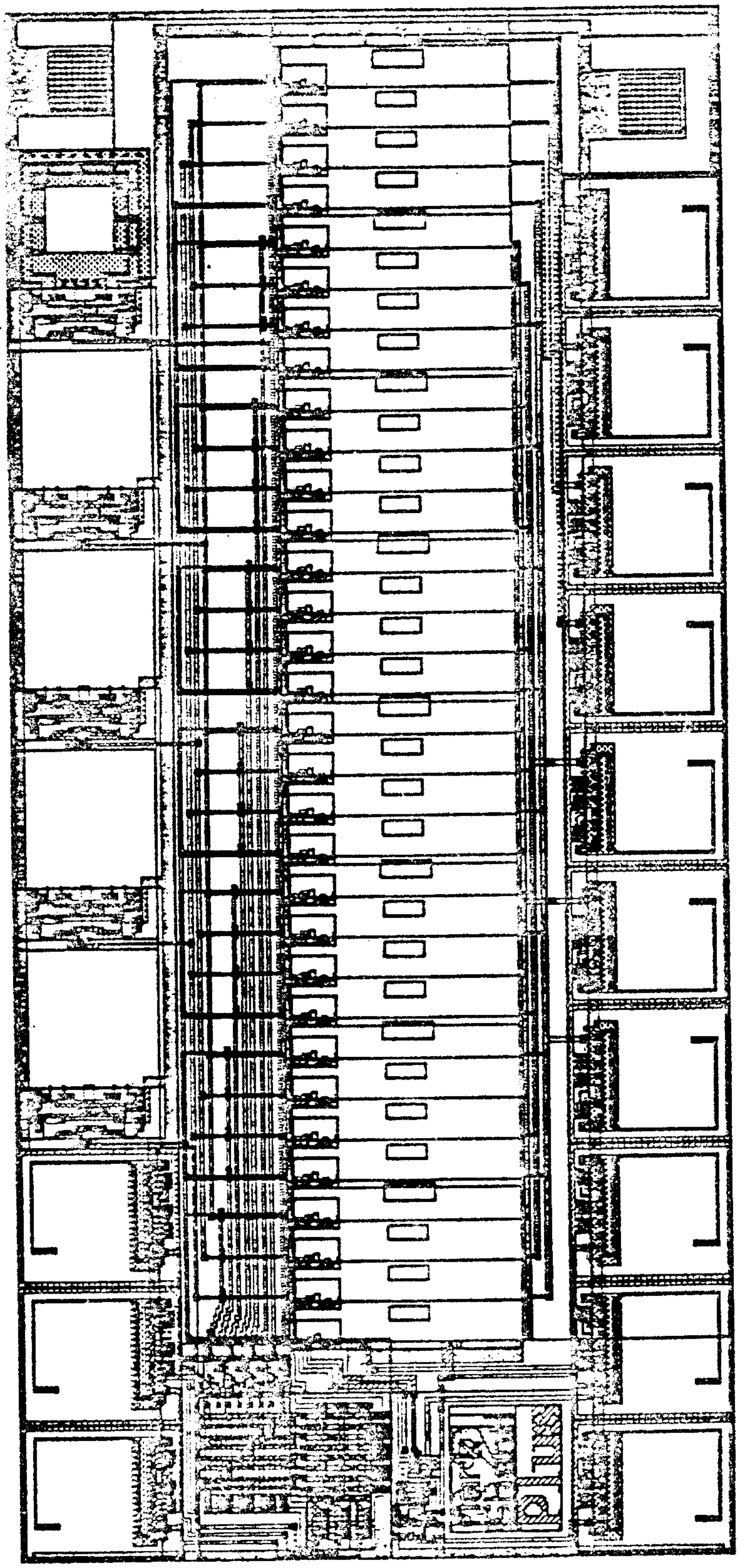
Plot A.5 - 32plus, top level.



Plot A.6 - 32plus, top two levels.



Plot A.7 - 32plus, top three levels.



Plot A.8 - 32plus, top four levels.