AN INTERACTIVE DESIGN RULE CHECKER FOR INTEGRATED CIRCUIT LAYOUT

by

Kwanghyun Kim

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Joseph G. Tront, Chairman

James R. Armstrong

Charles E. Nunnally

August, 1985

Blacksburg, Virginia

AN INTERACTIVE DESIGN RULE CHECKER FOR INTEGRATED CIRCUIT LAYOUT

by

Kwanghyun Kim

Joseph G. Tront, Chairman

Electrical Engineering

(ABSTRACT)

An implementation of an interactive design rule checker is described in this thesis. Corner-based design rule checking algorithm is used for the implementation. Due to the locality of checking mechanism of the corner-based algorithm, it is suitable for hierarchical and interactive local design rule checking. It also allows the various design rules to be specified very easily.

Interactive operations are devised so that the design rule checker can be invoked from inside the layout editor. All the information about the violation, such as position, type of violation, and symbol definition name are provided in an interactive manner. In order to give full freedom to the user to choose the scope of checking, three options, "Flattening", "Unflattening" and "User-defined window" are implemented in creating the database to be checked. The "User-defined window" option allows hierarchical design rule checking on a design which contains global rectangles. Using

these three options, very efficient hierarchical checking can be performed.

ACKNOWLEDGEMENTS

I sincerely wish to thank Dr. Joseph G. Tront for his encouragement, suggestions and for editing this thesis. I also would like to acknowledge my fellow graduate students and for their helpful advices for my work. Special thanks are directed to my family in and my wife.

TABLE OF CONTENTS

1.0 INTRODUCTION 1

2.0 LAMBDA-BASED NMOS DESIGN RULES 5

3.0 EXISTING DESIGN RULE CHECKING ALGORITHMS 19

3.1 Polygon Algebra Algorithm 19

3.2 Raster-Scan Algorithm 21

4.0 CORNER-BASED DESIGN RULE CHECKING ALGORITHM . . . 25

4.1 An Overview of the Corner-based Algorithm 25

4.2 Specification of NMOS Design Rules 32

4.2.1 Context Pattern 32

4.2.2 Spacing Rule Specification 36

4.2.2.1 Single Layer Spacing Rule Specification . 36

4.2.2.2 Interlayer Spacing Rule Specification . . 43

4.2.3 Width Rule Specification 57

4.2.4 Transistor Overhang Rule Specification . . . 61

4.2.5 Ion-implant Overhang Rule Specification . . . 75

4.2.6 Contact Cut Surrounding Rule Specification . 79

4.2.7 Butting Contact Rule Specification 89

4.2.8 Buried Contact Rule Specification 97

4.3 Corner Detection and Bit-Map Generation 105

4.4 A Limitation of the Corner-based DRC Algorithm 115

5.0 IMPLEMENTATION OF CORNER-BASED DRC ALGORITHM . 117
5.1 Design Rule Checker 117
5.1.1 Corner Detection and Bit-Map Generation . . 117
5.1.2 Corner Processing 124
5.2 Interfacing DRC with ICICLE 132
5.2.1 Database Creation for Hierarchical Design Rule Checking 132
5.2.2 Reporting of Design Rule Violations 136

6.0 RESULT 139

7.0 CONCLUSION 140

8.0 REFERENCES 142

VITA 144

LIST OF ILLUSTRATIONS

Figure 1. Width and spacing rules for diffusion, poly and metal. 9

Figure 2. Transistor overhang rule. 10

Figure 3. Spacing and ion-implant overhang rule for depletion mode transistor. 12

Figure 4. Contact surrounding rule. 13

Figure 5. Spacing rule between transistor gate and contact cut. 14

Figure 6. Butting contact. 16

Figure 7. Buried contacts. 17

Figure 8. Spacing rule violation of metal in the 4λ X 4λ window. 22

Figure 9. Corners in the design. 26

Figure 10. Spacing requirement on a metal box. 27

Figure 11. Spacing rule specification for four corners of the metal box. 29

Figure 12. Spacing rule violation between two metal boxes. 30

Figure 13. Four quadrants around a corner. 33

Figure 14. Five types of the context patterns. 34

Figure 15. Context pattern of a convex corner and the constraint for the single layer spacing rule. 37

Figure 16. Spacing rule violation detected by two small "NOT M" constraints. 38

Figure 17. Context pattern of a concave corner and the constraint for the single layer spacing rule. 39

Figure 18. Spacing rule violation detected by the constraint of a concave corner. 40

Figure 19. Spacing rule violation at a one-side corner. 41

Figure 20. A convex corner of diffusion and the corresponding constraint for poly for the interlayer spacing rule. 45

Figure 21. Specification of the interlayer spacing rule between poly and diffusion for the context patterns of poly. 47

Figure 22. Specification of the interlayer spacing rule between poly and buried cut for the context patterns of poly. 48

Figure 23. Specification of the interlayer spacing rule between poly and diffusion for the context patterns of diffusion. 49

Figure 24. Specification of the interlayer spacing rule between diffusion and buried cut for the context patterns of diffusion. 50

Figure 25. Specification of the interlayer spacing rule for the context patterns of buried cut. . . 51

Figure 26. Specification of the interlayer spacing rule between transistor gate and ion-implantation for the context patterns of ion-implantation. 52

Figure 27. Specification of the interlayer spacing rule between transistor gate and ion-implantation for the context patterns of transistor gate. 53

Figure 28. Specification of the interlayer spacing rule between transistor gate and contact cut to diffusion for the combined context patterns of diffusion and contact cut. 54

Figure 29. Specification of the interlayer spacing rule between transistor gate and contact cut to diffusion for the context patterns of transistor gate. 55

Figure 30. (a) Specification of the width rule for a convex corner.
(b) Specification of the width rule for a concave corner. 58

Figure 31. Width rule violation at convex, concave and one-side corners. 59

Figure 32. Corner in a transistor. 62

Figure 33. Combined context patterns of diffusion and poly for a convex type context pattern of diffusion. 64

Figure 34. Specification of the transistor overhang rule for the combined context patterns of group (C) in Figure 33. 66

Figure 35. Combined context patterns of diffusion and poly for a one-side context pattern of diffusion. 67

Figure 36. Specification of the transistor overhang rule for the combined context patterns in Figure 36. 68

Figure 37. Combined context patterns of diffusion and poly for a concave type context pattern of diffusion. 70

Figure 38. Specification of the transistor overhang rule for the combined context patterns in Figure 37. 71

Figure 39. Combined context patterns of diffusion and poly for the all-filled type context pattern of diffusion. 72

Figure 40. Specification of the transistor overhang rule for the combined context patterns in Figure 39. 73

Figure 41. Specification of the ion-implant overhang rule for the context patterns of the depletion mode transistor. 76

Figure 42. Specification of the ion-implant overhang rule for the context patterns of the depletion mode transistor. 77

Figure 43. Contact cut. 80

Figure 44. One-side and all-filled corners in the contact cut containing more than one rectangle. . . 81

Figure 45. A contact cut containing a concave corner. 82

Figure 46. Specification of the contact cut surrounding rule. 83

Figure 47. Specification of the contact cut surrounding rule. 84

Figure 48. Specification of the contact cut surrounding rule. 85

Figure 49. Specification of the contact cut surrounding rule. 86

Figure 50. Contact cut surrounding rule violations detected at one-side corners. 87

Figure 51. (a) Region which is specified by the contact cut surrounding rule in the butting contact.
(b) Butting contact. 90

Figure 52. (a) Checking procedure for the poly-diffusion overlap.
(b) Procedure for the butting contact rule checking. 92

Figure 53. Combined context patterns of cut, diffusion and poly at the corners of contact cut in the butting contact. 94

Figure 54. Constraints for the context patterns in Figure 54 for the butting contact. 95

Figure 55. Three configurations of the butting contact obtained by rotation of the configuration in Figure 51. 96

Figure 56. Buried contacts. 98

Figure 57. Procedure of the buried contact rule checking. 99

Figure 58. Specification of the buried contact rule for the corner of buried cut with poly. 101

Figure 59. Specification of the buried contact rule for the corners of only buried cut when a corner of poly exists at left. 102

Figure 60. Specification of the buried contact rule for the corners of only buried cut when a corner of diffusion exists at left. 103

Figure 61. Specification of the buried contact rule for the corners of only buried cut when no layer exists at left except buried cut. 104

Figure 62. Two corners of a rectangle. 107

Figure 63. (a) Two rectangles in the layout.
(b) Database for two rectangles.
(c) Two sort arrays in order of bottom and top edge of two rectangles. 111

Figure 64. Final bit-map and detected corners. . . . 114

Figure 65. A false violation detected in the spacing rule checks. 116

Figure 66. Grid on the scanning window. 119

Figure 67. (a) A concave corner of metal.
(b) Constraints for the width and spacing rules in the scanning window. 121

Figure 68. Overall flow of the design rule checker using the corner-based algorithm. 123

Figure 69. (a) Four numbers assigned to the four quadrants around a corner.
(b) Sixteen pattern numbers for the 16 context patterns. 125

Figure 70. Numbers for the expansion direction used in the routine COUNT_PIXEL. 127

Figure 71. Function COUNT_PIXEL. 128

Figure 72. Spacing and width checks at a concave corner of the poly layer using the routine COUNT_PIXEL. 130

Figure 73. A linked list for the design rule error reporting 138

1.0 INTRODUCTION

Recent developments in Very Large Scale Integrated (VLSI) circuit technology allow the manufacturing of complex VLSI circuits containing 100,000 transistors, and it is projected that the technology will support fabrication of a million transistors in the near future. One of the key steps in the VLSI circuit design procedure is to create a layout of different layers that comprise the circuit. The layout consists of various geometric configurations to be patterned as layers of different materials on the chip. The fabrication process uses this layout as the description of the circuit.

On the layout, various geometric restrictions such as minimum width, spacing and overlap between layers are placed. These restrictions result in a set of geometric design rules, specific to a particular fabrication process. Since these design rules originate directly from the nature of the fabrication process, it is extremely important to obey them strictly so that (1) the design remains within the limitation of the fabrication process and (2) the circuit description is preserved in the fabrication process.

Although significant advances are being made in the developments of CAD tools to create layouts automatically, Integrated Circuit (IC) design still heavily depends on manual

operations, especially in custom chip design. When the circuit complexity of a chip increases to the level of complexity of a VLSI circuit, the probability of violating design rules becomes significant. Additionally, it is almost impossible to check a circuit for the design rule errors manually. It is for this reason that a software to carry out design rule checking, commonly referred to as a Design Rule Checker (DRC), must be developed.

A DRC accepts a geometrical description of layout as an input, typically in the form of a database, and produces an indication of the design rule violation in the layout as an output. In addition to this basic function of detecting violations, a DRC should have the following features to be an efficient and effective tool in VLSI design.

1. Speed: Due to the complexity of a VLSI circuit, it takes a fairly long time to carry out design rule checking on even moderate size design. Therefore, the execution speed of a DRC should be made as fast as possible.

2. Interactivity: Since, in general, a layout is created interactively using interactive graphic editors, design rule checking also should be accomplished in an interactive manner, i.e, all the information about a design rule

violation should be displayed over the layout so that the designer can correct any design rule violations easily.

3. Hierarchical nature: Due to the hierarchical nature of the database created by many layout editors, design rule checking becomes very efficient if it is carried out hierarchically.

One drawback of the previous design rule checking algorithms are their long execution time [12-15]. As a solution to this problem, Arnold [1] proposed a new design rule checking algorithm, called the "Corner-based" algorithm, for "Manhattan circuit". A Manhattan type circuit is a layout which consists of only rectangles. In the corner-based algorithm, each design rule is specified by the geometric patterns of the corner and corresponding constraints which should be kept at all corners having the same geometric pattern. Here "corner" means four corners of a rectangle as well as intersections between rectangles. Since the design rule checking is performed by processing only corners of the layout, this algorithm can save much execution time compared to previous design rule checking algorithms.

In this thesis, the corner-based algorithm and its implementation are presented. Also the interfacing of the DRC

with the integrated circuit layout editor, called ICICLE [17-18], is described.

Chapter 2 describes the λ-based NMOS design rule given by [3] and [4]. Chapter 3 reviews some existing design rule checking algorithms. Chapters 4 and 5 describe the corner-based algorithm and its implementation.

2.0 LAMBDA-BASED NMOS DESIGN RULES

As stated in chapter 1, design rules are certain geometrical constraints given to the designer so that the geometrical pattern of the design is preserved in the fabrication process. These constraints are given in terms of minimum requirements for line widths, spacings, extensions and overlaps in various IC mask layers.

Lambda-based NMOS design rules were developed by Mead & Conway [3]. In this set of design rules, all the rules are specified in terms of a length parameter called "λ". Before λ-based design rules were developed, the design rules were specified in terms of more than one parameter and absolute physical dimensions were used for specifying the parameters according to the target fabrication process. The permissible minimum feature size of the fabrication process is becoming smaller with advances in fabrication technology. This decreasing trend of the minimum feature size has caused some problems in using the traditional design rules. For example, if the minimum feature size of the target fabrication process is changed during the design period of a chip, that design becomes obsolete. Moreover, since each process has its own specific characteristics, an IC designed for particular process may not be fabricable by other processes.

Lambda-based design rules can solve these kinds of problems since only one dimensionless parameter "λ" is used for specifying the all NMOS design rules. In other words, since the layout is designed on λ, it can be fabricated by any process although the physical value of λ varies depending on the process.

The parameter λ is defined as the minimum feature size which can be fabricated by the process. This definition of λ can be interpreted in two ways:

1. Bound on the deviation of the feature size on the wafer

2. Misalignment of the mask from its original position

With these two interpretations of λ, we can consider the following situation that may occur in the fabrication process. Suppose that an edge of a feature in one layer at a design has deviated by 1λ, and one edge in another layer is shifted by 1λ due to the misalignment of that layer from its original position. In this case, the original separation between two edges increases by 2λ. The fabricated chip cannot perform the intended function if the original separation is critical to its proper functioning. For this reason, any feature should be at least 2λ wide, and the separation between two features is also at least 2λ to maintain the re-

lationship between different features. From these basic premises, Mead & Conway specified all the NMOS design rules in terms of λ as follows.

For polysilicon (poly) regions, basic 2λ width and spacing rules are adopted directly since there is no other factor which has to be considered.

Diffusion regions need a minimum separation 1λ wider than the basic 2λ spacing requirement. Since the depletion region is formed outside the diffusion region, if two separated diffusion regions are placed too close, then their depletion regions will overlap and current can flow through the overlapped depletion region. The basic width requirement of 2λ is enough for proper functioning.

There is also a spacing requirement between poly and diffusion regions even though the two layers cannot be connected unintentionally. If unrelated poly and diffusion layers are overlapped accidentally, unwanted capacitance is formed and the diffusion region is reduced since diffusion region cannot be formed underneath a poly region. However, since these two factors are not fatal in the proper functioning of the design, only 1λ spacing requirement is given between two layers.

For metal regions, a 3λ width and 3λ spacing are required. Metal layer is patterned last among all the layers and covers the other layers which are already patterned. Therefore, this layer has much more non-uniform terrain to cover compared to the other layers. For reliable covering, one more λ is added to both the basic width and spacing requirements. The width and spacing rules for above three layers are shown in Figure 1.

When a diffusion region and a poly region are overlapped to form a MOS transistor, two unwanted geometrical configurations can be formed. If the diffusion region does not cross the poly region completely, the drain or source region of the transistor is not formed. If the poly region does not reach the end of the diffusion region, the drain and source of the transistor will be shorted. To prevent both cases, a requirement of the 2λ extension of poly and diffusion from the boundary of the transistor gate region is given as shown in Figure 2.

In the depletion mode transistor, if the channel between drain and source is not filled completely with ion-implant, a small part of the channel will operate as an enhancement mode transistor. To avoid this, a 1.5λ extension of ion-implant is required beyond the boundary of the gate region as shown in Figure 3. Moreover, to prevent other nearby en-

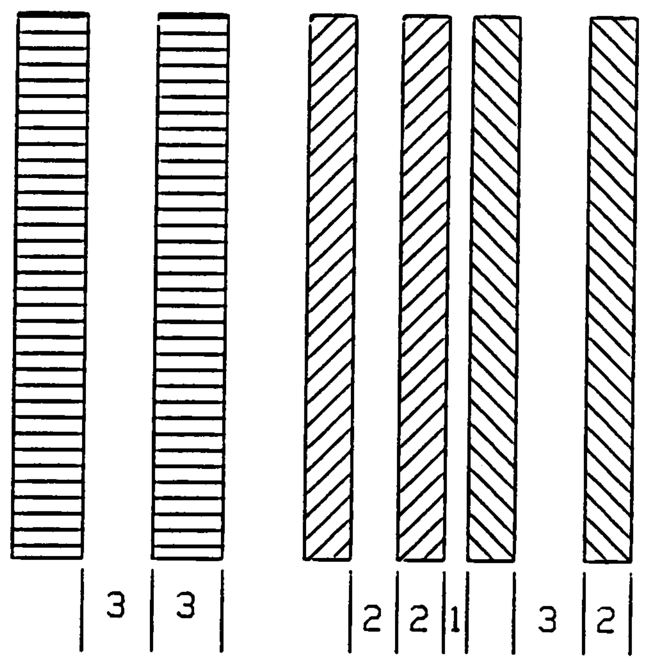


Figure 1. Width and spacing rules for diffusion, poly and metal.

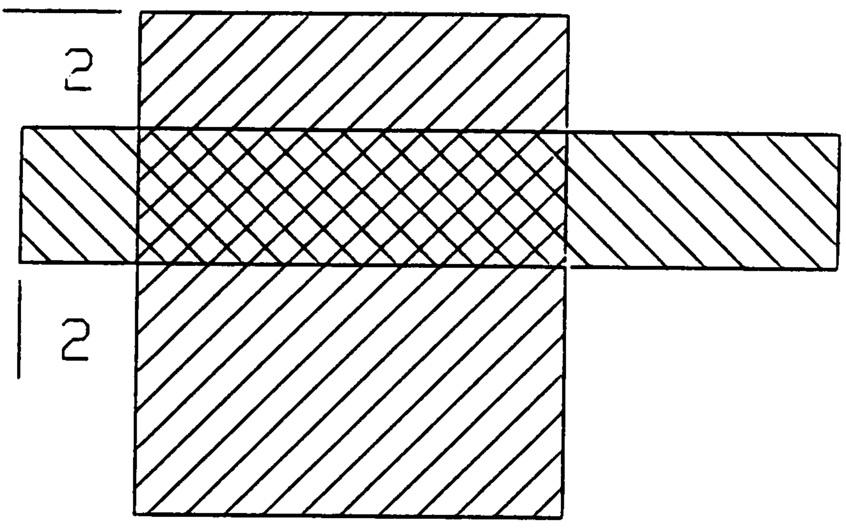


Figure 2. Transistor overhang rule.

hancement mode transistor from operating as a depletion mode transistor, a 1.5λ spacing requirement between ion-implant and unrelated transistor gate is given. This rule is also shown in Figure 3.

The purpose of a contact cut is to connect the metal with either poly or diffusion region. To form the proper connection, the two different regions ('metal & poly' or 'metal & diffusion') should be placed within the contact cut window. This allows the metal to reach the poly or diffusion region on the silicon wafer through the contact cut window. To ensure a proper connection, it is required that 1λ of metal AND (poly OR diffusion) surround the contact cut on all sides. It is also required that the basic 2λ width and spacing requirements be followed during placement of the contact cut window. These rules are shown in Figure 4.

When drain or source of a transistor is connected to a metal region using a contact cut, 2λ separation between transistor gate region and contact cut is required. If the contact cut window is placed very near the transistor gate region, drain or source of the transistor may be connected to the gate of the same transistor in the fabrication process. This spacing rule between transistor gate and contact cut is shown in Figure 5.

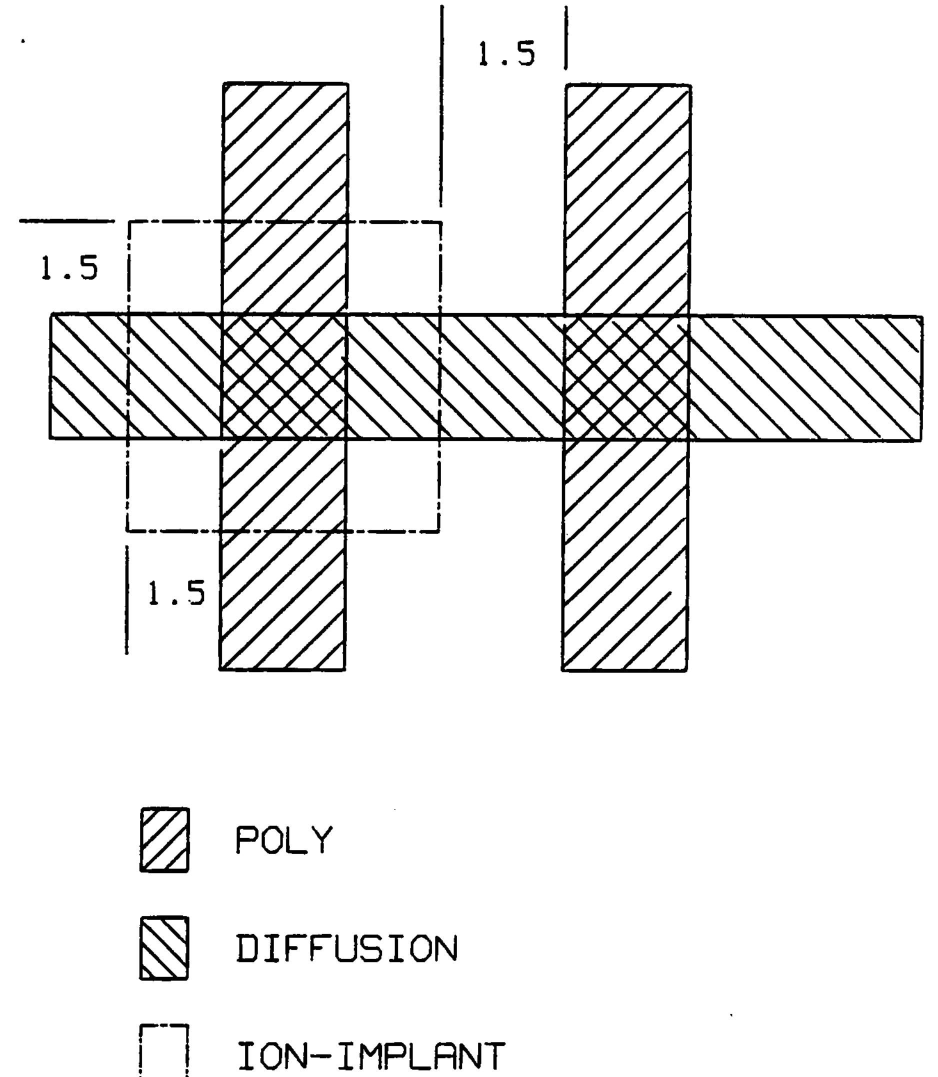


Figure 3. Spacing and ion-implant overhang rule for depletion mode transistor.

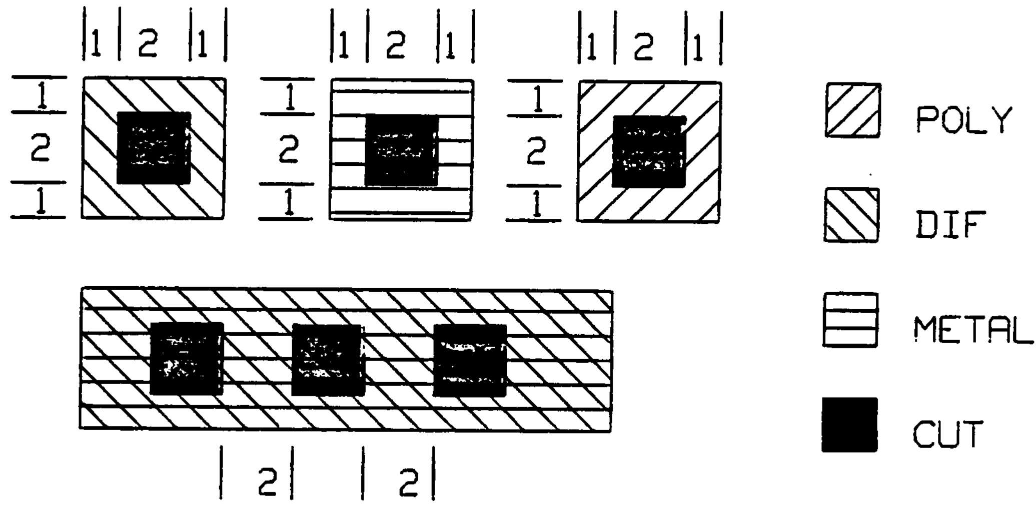


Figure 4. Contact cut surrounding rule.

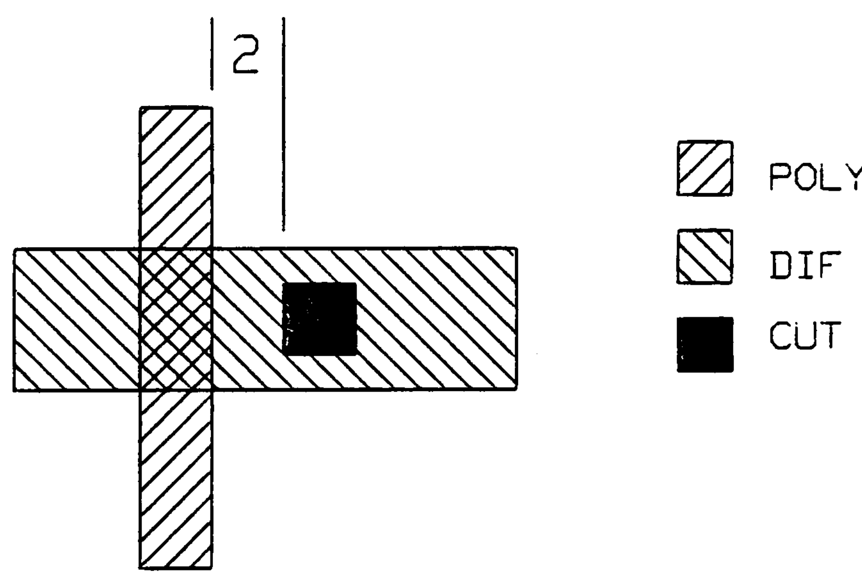


Figure 5. Spacing rule between transistor gate and contact cut.

There are two kinds of constructs used to connect the diffusion and poly regions. One is the "butting contact" and the other is the "buried contact". In a butting contact, the connection between the diffusion and the poly layer is made using an ordinary contact cut window. The same method as in the pure contact cut is used in this contact, and then the surroundings requirements of the contact window are also given to the butting contact. In addition to the surroundings requirements, to make sure of the connection between diffusion and poly region, 1λ overlap is required between poly and diffusion reigon. Figure 6 shows the configuration of the butting contact.

For a buried contact, the first step in making a connection is to remove the thin oxide layer between poly and diffusion layer. This necessitates an additional masking step to remove the thin oxide layer. This buried contact rule is not specified in the Mead & Conway NMOS design rules. Although the same configuration is used in general, the minimum distance requirement is slightly different depending on the fabrication process. Since the silicon broker used by VPI & SU and many others is MOSIS, we will describe the buried contact rules which have been prescribed by MOSIS [4].

There are three configurations for buried contacts as shown in Figure 7. In this figure, the configuration (a) and

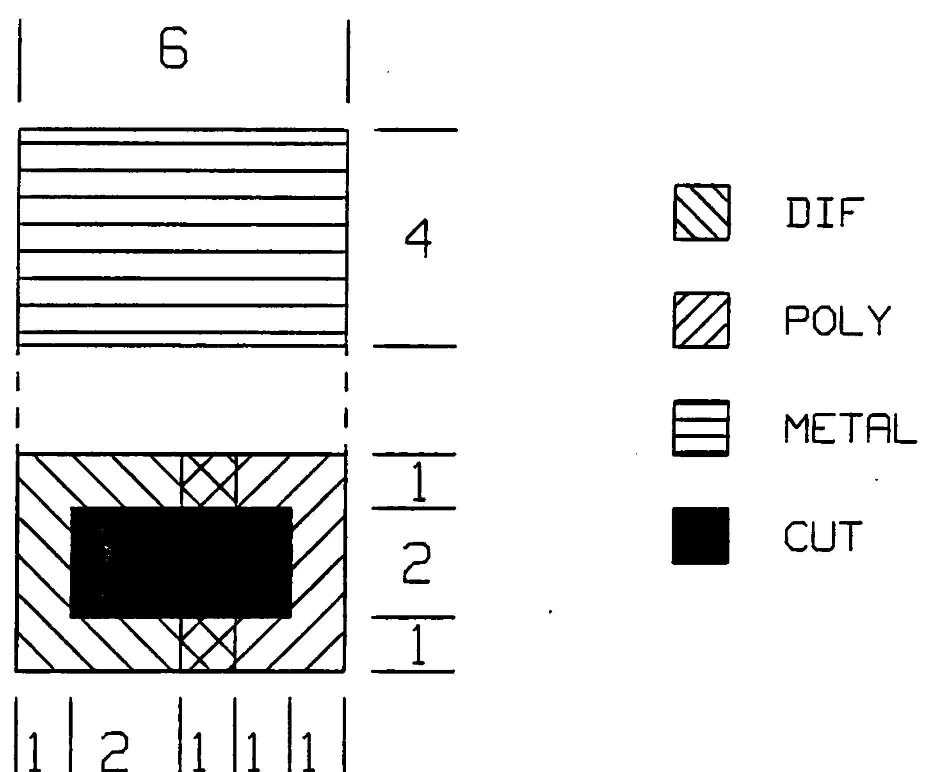


Figure 6. Butting contact.

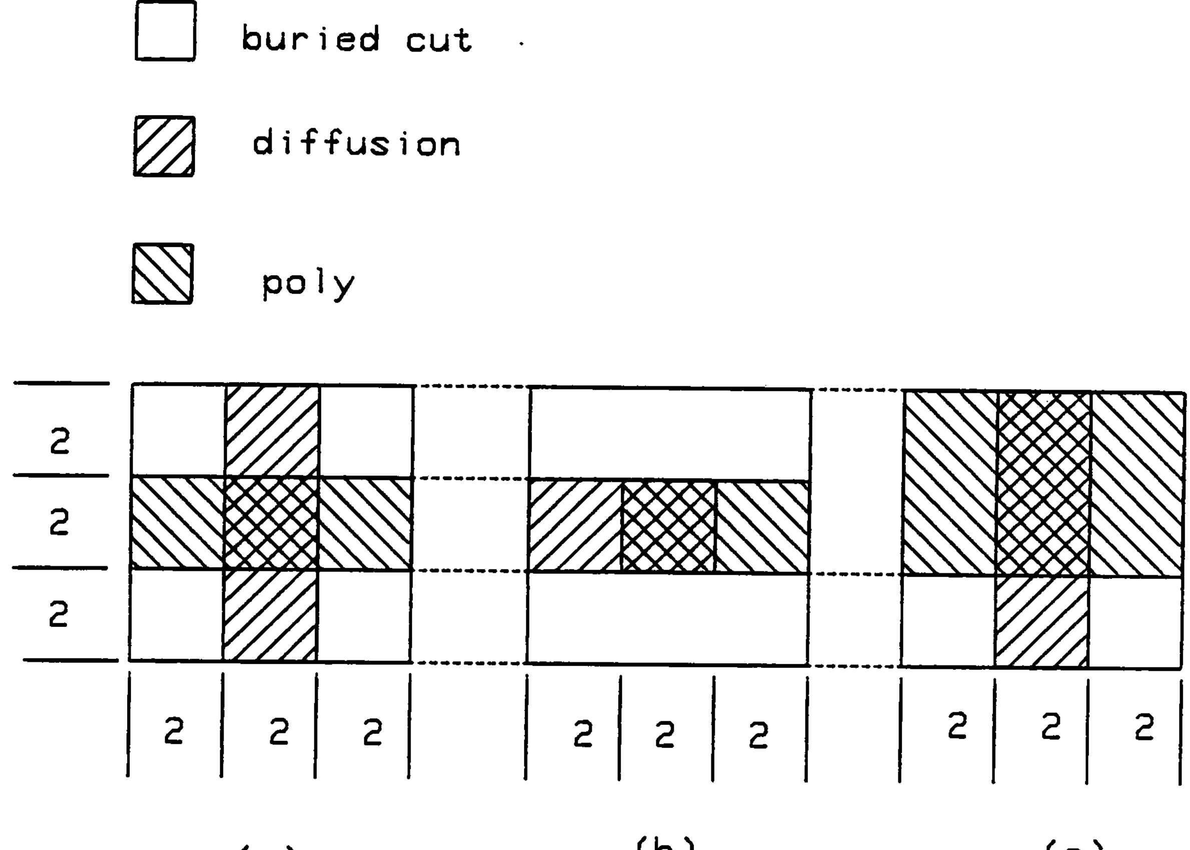


Figure 7. Buried contacts.

(b) are used for straightforward connections and (c) is used for pull-up transistor compaction. There are two requirements for a buried contact. First, the buried cut window should be 6λ wide and 6λ long to provide enough space for connecting diffusion and poly. Second, the poly-diffusion overlap should be at least 2λ X 2λ square and this square should be placed at the center of the buried contact window as shown in Figure 7.

Due to the potability of the λ-based design rules, the development of the DRC software has been accelerated. In the next chapter, we will introduce two most commonly used DRC algorithms.

3.0 EXISTING DESIGN RULE CHECKING ALGORITHMS

Among the existing design rule checking (DRC) algorithms, two algorithms have been used most commonly. One is Polygon Algebra algorithm and the other is Raster-scan algorithm. In this chapter, we will describe these two algorithms briefly.

3.1 POLYGON ALGEBRA ALGORITHM

Most of the running DRC in industry have been developed based on this polygon algebra algorithm [12-15]. In this algorithm, the smallest entity is a polygon and each mask layer is treated as a collection of polygons. An algebra is defined to manipulate and combine the layers. This algebra includes various operations such as growing and shrinking polygons on single layers, merging of all intersecting polygons, intersection, union and difference of layers and so on. Using this algebra, all kinds of design rules can be checked. For example, a typical way to perform a spacing rule check is to grow each polygon by half of the minimum spacing and then to look for intersecting polygons. In the worst case, the number of comparisons needed to determine all polygon intersections is proportional to the square of the number of polygons. For the rules in which the transistor

gate is involved, one additional layer "transistor" is derived using the polygon algebra:

transistors = poly AND diffusion.

Using the algebra outlined above, design rule violations are found by a sequence of operations on the original and derived layers.

A strong merit of this method is its use of algebraic operations. Using these algebraic operations, any design rule set can be specified very easily. Even complex rules may be described in a straightforward manner, although they may require long sequences of operations.

Polygon algebra algorithm has several drawbacks. First, the entire layout is treated as a single entity: all operations should be performed on the entire layout and cannot be applied to small areas of the design. Therefore, if the design is large, execution time will be very long. Since some operations need comparisons, if the size of the design is doubled, the execution time may increase four times in the worst case. Recent design rule checking algorithms reduce the number of comparisons by using sorting and windowing techniques [10]. Secondly, this algorithm requires a large amount of storage due to the creation of many intermediate

layers when multilayer checking is performed. Moreover, since this database is maintained on disk, frequent accesses to disk make the processing slow. If the design is large, this problem becomes very severe.

As we see in the above, polygon algebra algorithm has a drawback in its slow speed. Therefore, if we consider the trend of increasing chip complexity, this algorithm may not be applicable to the complex design of the future.

3.2 RASTER-SCAN ALGORITHM

The raster-scan algorithm is proposed by Baker [7] and can be used only for Manhattan type circuits. The basis for the algorithm is that all the basic design rules are local, in that they are only minimum widths and spacings. For this algorithm, the layout is represented on a λ grid. Each pixel, a 1λ X 1λ square on a grid, contains one bit which represents the existence of the layer at that position for each layer. Since the maximum width and spacing requirement is 3λ (for metal), a 4λ X 4λ window will view sufficient information so as to allow for detecting the width and spacing rule violations. Shown in Figure 8 is one example of a spacing rule violation on metal in the 4λ X 4λ window.

M	M	M	M
M	M	M	M

Figure 8. Spacing rule violation of metal in the 4λ X 4λ window.

Design rule checking is accomplished by moving the window over the layout, in the horizontal and vertical direction. Multilayer rule checking is performed in a similar manner. For example, if four pixels in the center of the window are all contact cut, then the whole 4λ X 4λ window should be filled with 'metal AND poly' or 'metal AND diffusion' to meet the minimum surroundings requirements specified in the design rules of Chapter 2.

In the raster-scan algorithm, design rules are specified in two ways: using tables and using special purpose code. For example, two tables are constructed for the width checks. One table is for the 3λ width check and the other is for the 2λ width check. There are two entries in the table. One entry is a number which represents the layer arrangement in the 4λ X 4λ window. The other entry is a bit indicating whether or not that arrangement of the layer violates the width rule. Since there are 2^{16} possible arrangements of a layer in the window, 8K bytes of memory are necessary to store one table if a memory address is used to image a layer arrangement in the window. In this table, 1 means that the layer arrangement of the window does not represents a design rule error and 0 means it does.

The special purpose code is constructed for multilayer checking. Because the number of possible windows is so large

if mutilayer interaction is considered, the table look-up technique requires large memory space. Special purpose coding eliminates this need for large memory.

The raster-scan algorithm has the following limitations. First, since the knowledge of design rules is embedded in the tables and special purpose code, when the design rules are changed, the tables and special purpose code must be rewritten. Moreover, if the new design rule set requires a window larger than 4λ X 4λ, it becomes very expensive to examine all possible windows. If the window size is increased to 6λ, it is almost impossible to apply the raster-scan algorithm. The second limitation is that the entire layout is scanned by the window. This causes execution time to be wasted if the layout area contains some empty regions.

Compared to the raster-scan algorithm, another technique of DRC called "Corner-based" algorithm has following advantages. First, the corner-based algorithm is much easier to specify the design rules, and it requires less memory space than the raster-scan algorithm. Second, since design rule checking is accomplished by checking the local geometric constraints, it is faster than the raster-scan algorithm which scans the entire layout. We will see the reasons for these advantages when the corner-based algorithm is discussed in Chapter 4.

4.0 CORNER-BASED DESIGN RULE CHECKING ALGORITHM

4.1 AN OVERVIEW OF THE CORNER-BASED ALGORITHM

The basis for the corner-based DRC algorithm comes from the fact that all geometrical design rules of the IC layout can be specified as constraints on the presence or absence of layers in the neighborhood of corners of a layout. It must be assumed that the entire layout consists only of rectangles. Here the corner refers to the four corners of a rectangle and intersections between other rectangles as shown in Figure 9. The specification of the design rules by constraints can be explained as follows.

Consider the metal box in Figure 10. To satisfy the 3λ spacing rule of Chapter 2, there should not be an unrelated metal box within the region bounded by dashed lines. We observe that the regions R1, R2, R3, R4 have the corners C1, C2, C3, C4 respectively, and the layer arrangement around each corner is distinct. Thus, the spacing requirements for the region R1, R2, R3, R4 can be specified as the constraints on the corners C1, C2, C3, C4 respectively, as shown in Figure 11. As we see in Figure 11, a spacing constraint is determined by a geometric layer arrangement around the corner. This geometric arrangement will be called a "context pattern"

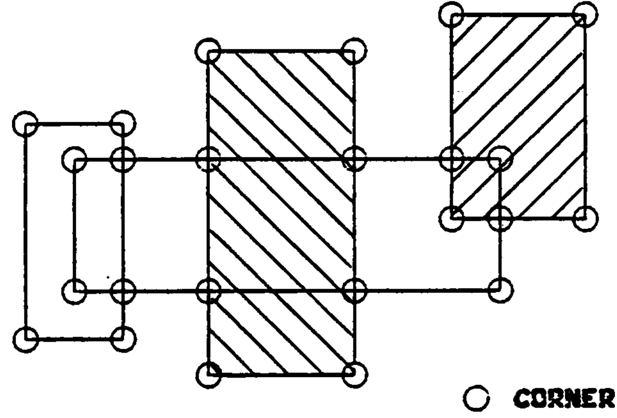


Figure 9. Corners in the design.

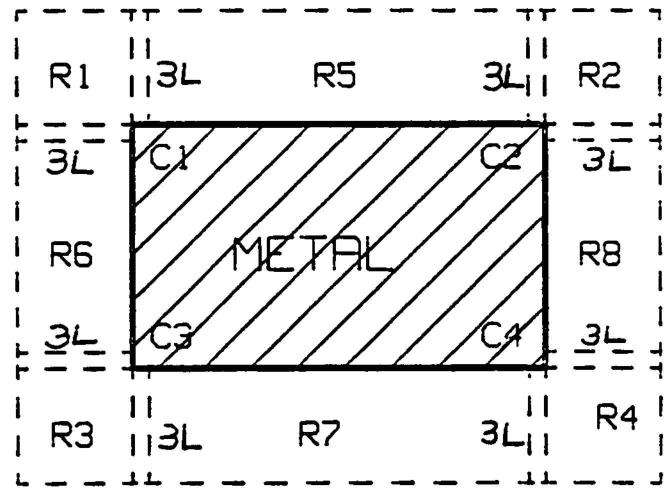


Figure 10. Spacing requirement on a metal box.

of the corner under consideration in the later chapters. Since the constraints in Figure 11 cannot check the space requirements for the region R5, R6, R7, R8, it might seem that more constraints are necessary. However, if there is a metal box in these regions, the violation can be detected by the constraint on the corner of the box which resides in these regions as shown in Figure 12. The constraints of the spacing rule for other context patterns will be explained in the next section.

In similar manners, all Mead & Conway design rules can be specified with context patterns and constraints. Thus, the actual design rule violations are detected by checking constraints given to the corner. In order to check the constraint, the position and context pattern of the corner must be known. Therefore, detecting a design rule violation is performed by the following three steps.

1. Corner detection

2. Context pattern identification

3. Constraint checking

We can now see why corner-based algorithm performs efficient design rule checks. Since the number of context pat-

CORNER	CONTEXT PATTERN (IF)	CONSTRAINT (THEN)
C1	NOT M NOT M NOT M M	3L D 3L NOT M NOT M D NOT M
C2	NOT M NOT M M NOT M	D 3L NOT M NOT M 3L NOT M D
C3	NOT M M NOT M NOT M	D NOT M 3L NOT M NOT M 3L D
C4	M NOT M NOT M NOT M	NOT M D NOT M NOT M 3L D 3L

D : 3.5L

Figure 11. Spacing rule specification for four corners of the metal box.

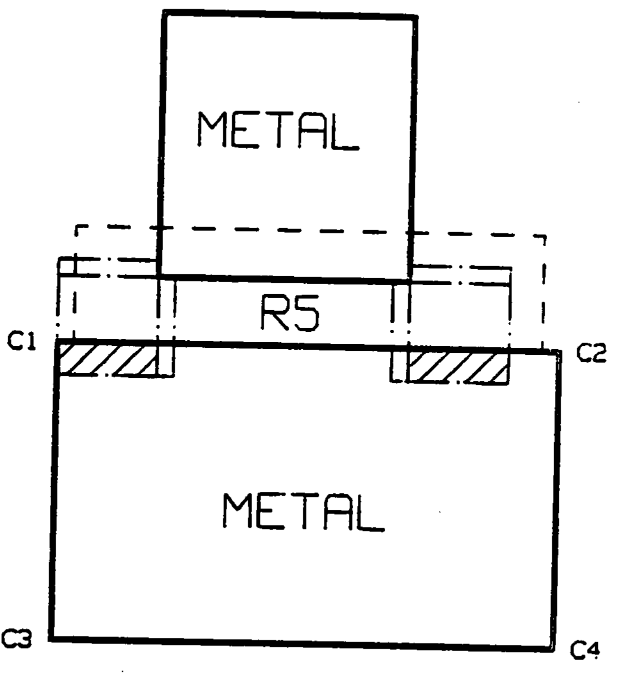


Figure 12. Spacing rule violation between two metal boxes.

terns is much less than the number of 4 X 4 matrices (2^{16}) , design rule specification is much simpler than that in the raster-scan method. Moreover, since the design rule checker already knows the positions of the corners, it can be intelligent enough not to perform the rule checking in empty regions, while the raster-scan method checks the entire layout.

4.2 SPECIFICATION OF NMOS DESIGN RULES

4.2.1 CONTEXT PATTERN

In the previous section, we defined "context pattern" as the geometric pattern of the layout around a "corner". Since it is assumed that the entire layout consists of only rectangles, there should be a vertical edge and a horizontal edge at every corner. Thus, layers around the corner can be considered as lying in a four quadrant plane with the origin at the corner as shown in Figure 13. This means that there are 15 possible context patterns for one layer. Hence, for a single layer rule specification, 15 constraints should be specified. Also, a rule for more than one layer should specify constraints for all possible combinations of the context patterns of each layer. These 15 context patterns can be divided into 5 groups based on the similarity of patterns as shown in Figure 14. This similarity can make it easy to specify the design rules because the constraints for one group of context patterns can be obtained by finding just one constraint and rotating it.

As an example, consider the metal spacing rule constraints in Figure 11. Notice that all four corners have convex type context patterns. The three constraints on corners C2, C3, C4 can be found by rotating the constraint of

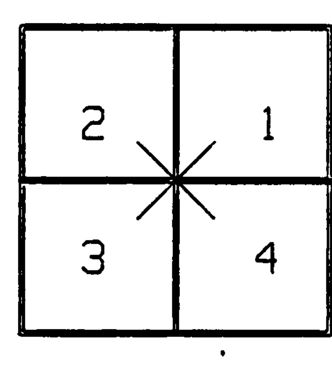


Figure 13. Four quadrants around a corner.

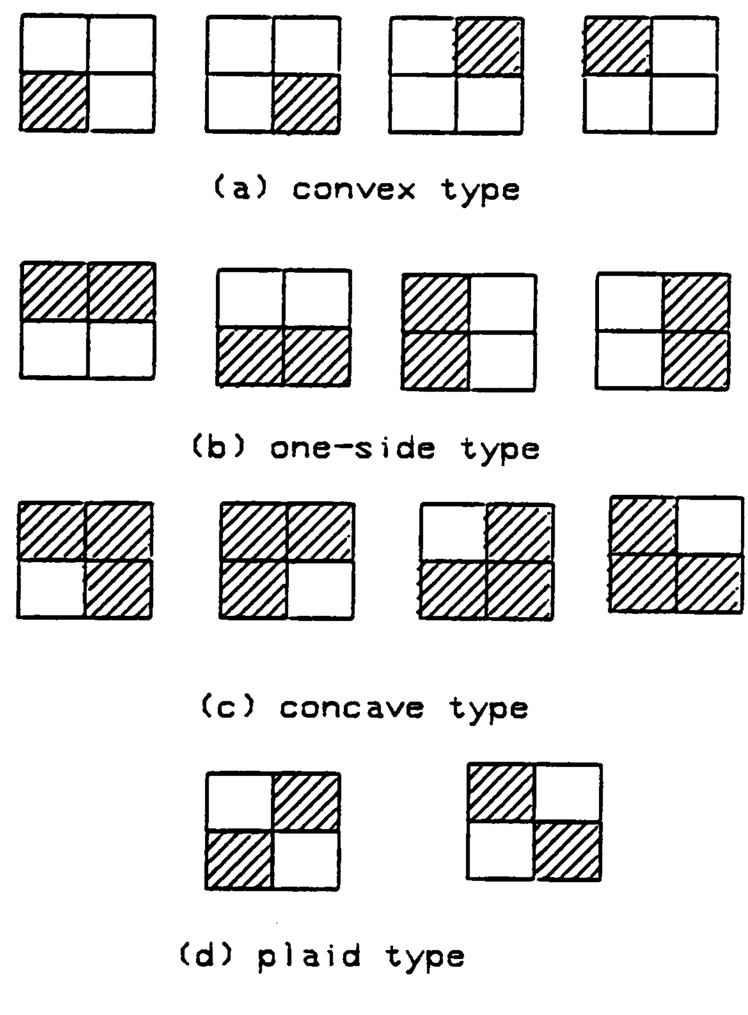


Figure 14. Five types of the context patterns.

C1 by 90, 180, 270 degrees, respectively. We will represent each group of context patterns in Figure 14 by a single context pattern and specify a single constraint for each group. The name given to each group will be used to represent all the context patterns in the group. The corner of any context pattern will be described by the type of that context pattern, e.g., the corner in a convex type context pattern will be called a convex corner.

The size of the context pattern window should be determined based on the basic unit of the layout. Since one-half λ is the basic unit of the layout editor being used, ICICLE, the size of the context pattern is determined as 1λ X 1λ square, i.e., two 0.5λ X 0.5λ boxes on each side of the context pattern.

4.2.2 SPACING RULE SPECIFICATION

4.2.2.1 SINGLE LAYER SPACING RULE SPECIFICATION

Single layer spacing rules indicate the required minimum distance between two separate regions on the same layer. In this section, we will specify the constraints of this rule for each type of context pattern.

The single layer spacing rule for a convex type corner, shown before in Figure 10, is repeated in Figure 15. Note the two small "NOT M" constraints. The reason for these constraints will be explained through following example. Consider three metal boxes in which some edges have the same x-coordinate or same y-coordinate shown in Figure 16. As marked in Figure 16, these three boxes are violating the single layer spacing rule. However, since the bottom edges of box A and B have the same y-coordinate and the left side edges of box B and C have same x-coordinate, these violations cannot be detected by the big "NOT M" constraint. To detect this kind of spacing rule violation, two small "NOT M" constraints are necessary.

Figure 17 represents the constraint due to the single layer spacing rule for a concave corner. This constraint is used to detect the spacing rule violation in Figure 18.

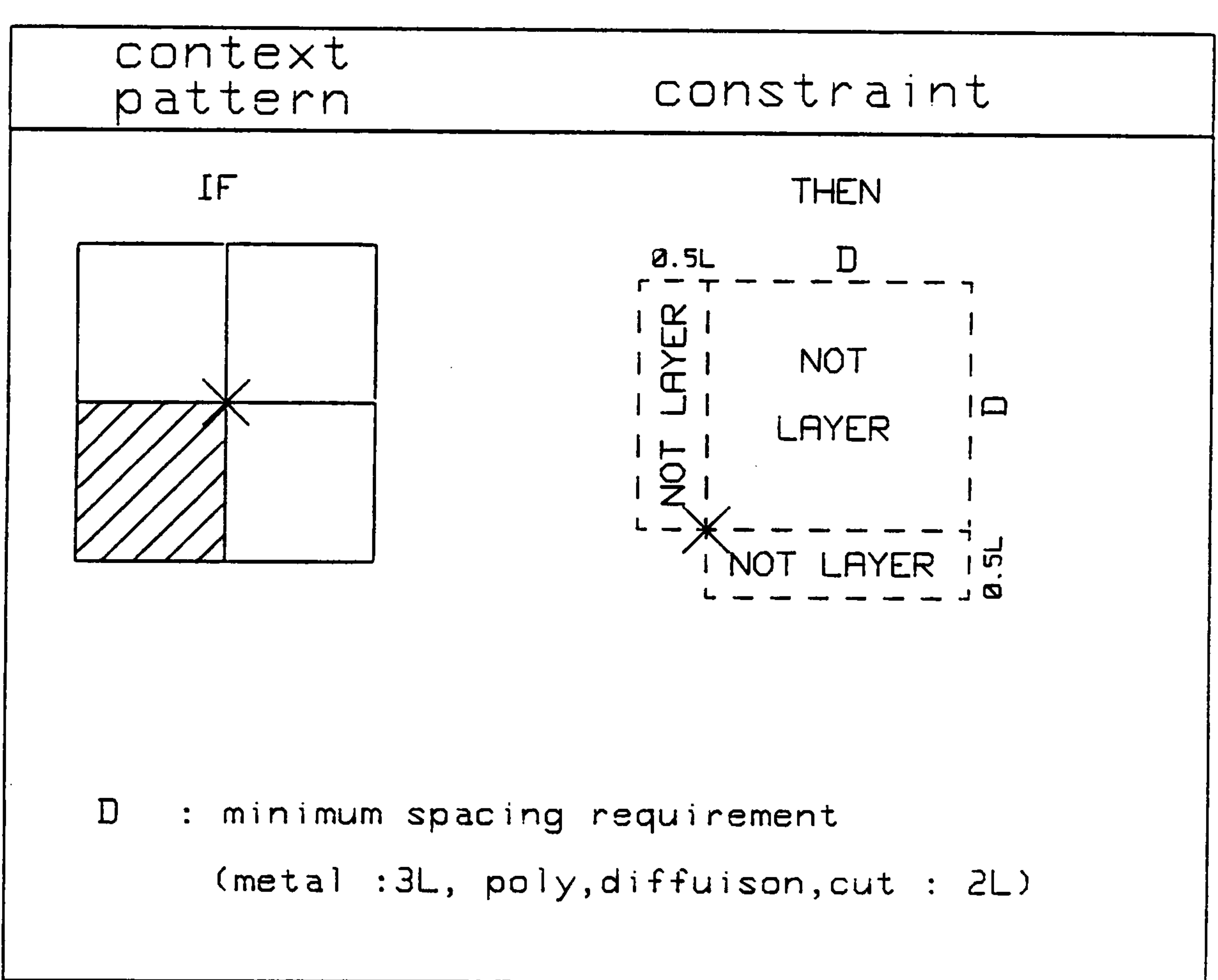


Figure 15. Context pattern of a convex corner and the constraint for the single layer spacing rule.

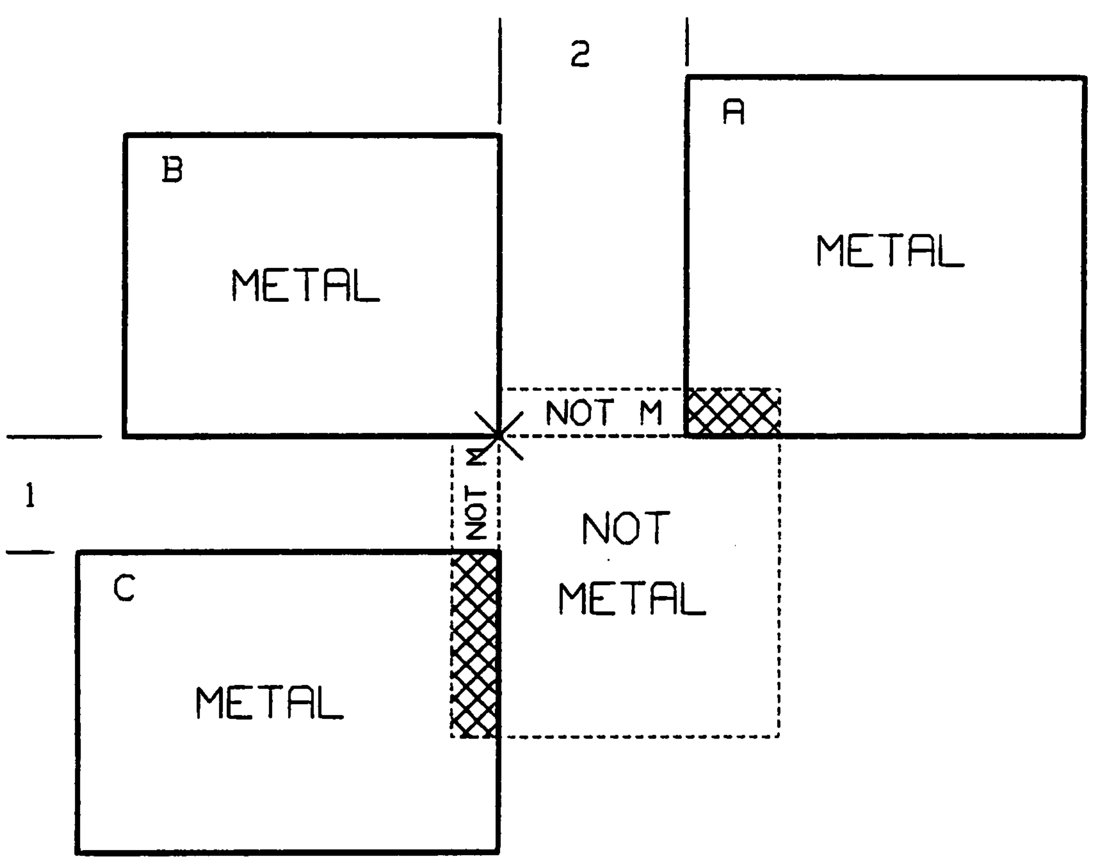


Figure 16. Spacing rule violations detected by two small "NOT M" constraints.

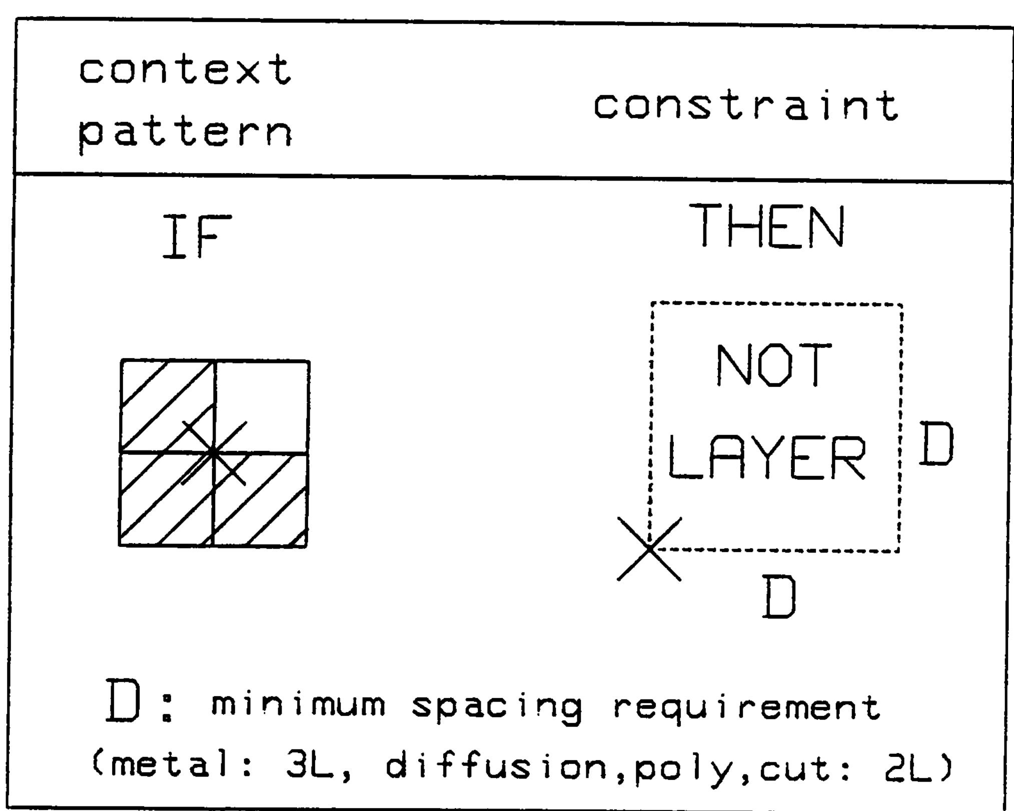


Figure 17. Context pattern of a concave corner and the constraint for the single layer spacing rule.

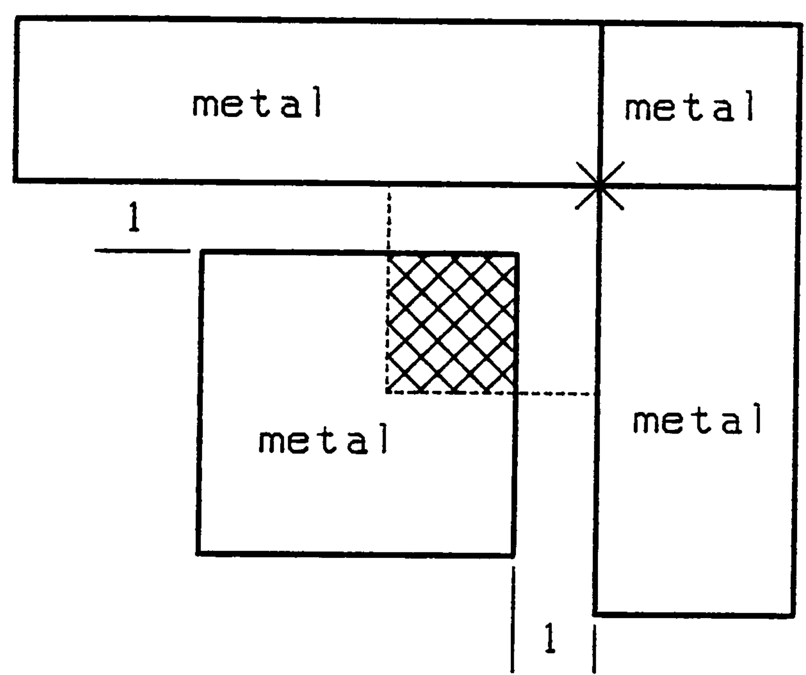


Figure 18. Spacing rule violation detected by the constraint of a concave corner.

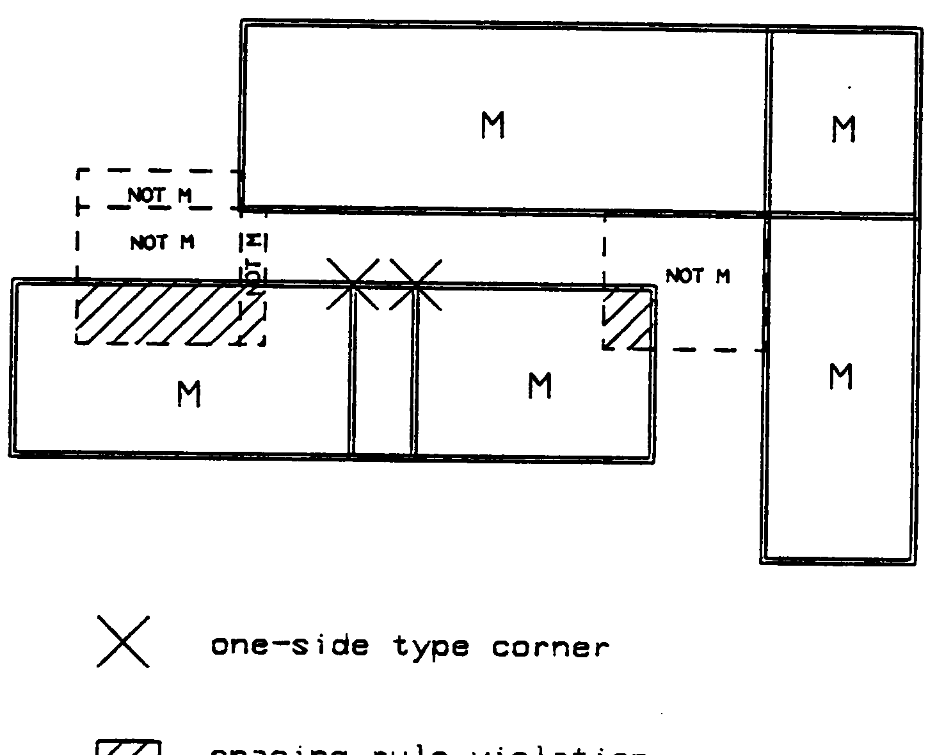


Figure 19. Spacing rule violation at a one-side corner.

A one-side corner is generated when two rectangles are connected in parallel as shown in Figure 19. If there is a spacing rule violation at this type of corner, this violation also will be detected at convex or concave corner as shown in Figure 19. This is due to the characteristics of Manhattan circuits. In the box configuration of the above example, the fact that the rectangles are rectilinear assures that if they are separated widely enough at the end points, then they do not violate the design rules at any point along their lengths. Therefore, the single layer spacing rule need not be checked at a one-side corner.

A violation can be reported without checking any constraint if a plaid corner is found since the very existence of such a corner is a spacing rule violation. For the context pattern in which all four quadrants are filled, we cannot identify any separation. Therefore, a constraint is not necessary.

As the foregoing analysis explains, the single layer spacing rule places constraints on only eight of the fifteen possible context patterns. It should be noted that this is much simpler than the spacing rule specification of the raster-scan method.

4.2.2.2 INTERLAYER SPACING RULE SPECIFICATION

Five interlayer spacing requirements of the Mead & Conway design rules are given in Table 1.

Table 1. Interlayer spacing rule.

LAYERS	MINIMUM SPACING REQUIREMENT
Poly & Diffusion	1λ
Buried-Cut & Diffusion	2λ
Buried-Cut & Poly	2λ
TR-gate & Ion-implant	1.5λ
TR-gate & Cut to diffusion	2λ

The only difference, with respect to the nature of the context patterns and constraints, between the single layer and interlayer spacing rule is that more than one layer is involved. Therefore, the same constraints as in the single layer spacing rules can be used for specifying interlayer spacing rules with proper changes of layer types and minimum

distances. For example, the spacing rule between the poly and diffusion for a convex corner of diffusion can be specified by the constraint on poly as shown in Figure 20. The example shows that this constraint is same as the constraint in Figure 15 except for the layer type and the minimum distance.

To check the interlayer spacing rule for transistor gates, we also need to give special attention to the MOS transistor gates besides different layer types and minimum distances. Since two layers are involved in the MOS transistor gate, the overlap between diffusion and poly should be considered as a single layer to specify the spacing rule for the transistor gate by similar constraints as in single layer spacing rule. In other words, if a corner with poly-diffusion overlap is found, the context pattern of the overlap should be identified first. Then, the corresponding constraint on the contact cut and ion-implant should be checked. If the constraint is specified for the poly-diffusion overlap, i.e., if a corner of cut to diffusion or an ion-implant corner is found, the poly-diffusion overlap should be searched in the constrained region.

Using the above ideas, all the interlayer spacing rules can be specified as shown in Figure 21 - Figure 29. Figure 21 and Figure 22 show the constraints on diffusion and

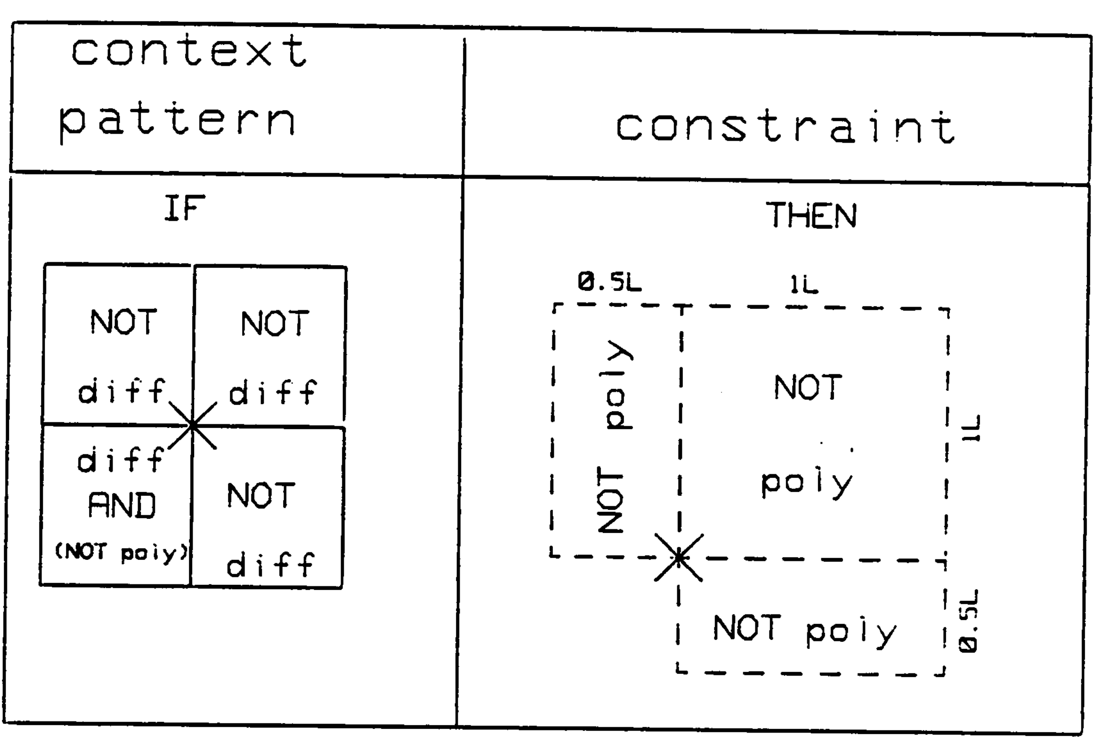


Figure 20. A convex corner of diffusion and the corresponding constraint of poly layer for the interlayer spacing rule.

buried cut for the poly context patterns. If a diffusion corner which is involved neither transistor nor buried cut is found, the interlayer spacing rule for poly and buried cut should be checked. Figure 23 and Figure 24 show the specification for this case.

For the buried cut corner, the constraints on diffusion and poly should be checked as shown in Figure 25. Note that the first quadrant of the top context pattern of buried cut is empty. This indicates the "don't care" condition, i.e., this context pattern represents either convex or plaid type context pattern.

As stated in Chapter 2, to prevent an enhancement mode transistor from working as a depletion mode transistor, there is a 1.5λ spacing rule between transistor gate region and ion-implant. Figure 26 and Figure 27 show the specification of this rule. The interlayer spacing rule between transistor gate and cut to diffusion can be specified as Figure 28 and Figure 29.

As we see in these figures, the constraints are specified for only convex and concave type of context patterns due to the same reason, i.e., rectilinearity of the Manhattan circuit and the spacing rule violations of plaid type corners, as in the single layer spacing rules. However, as

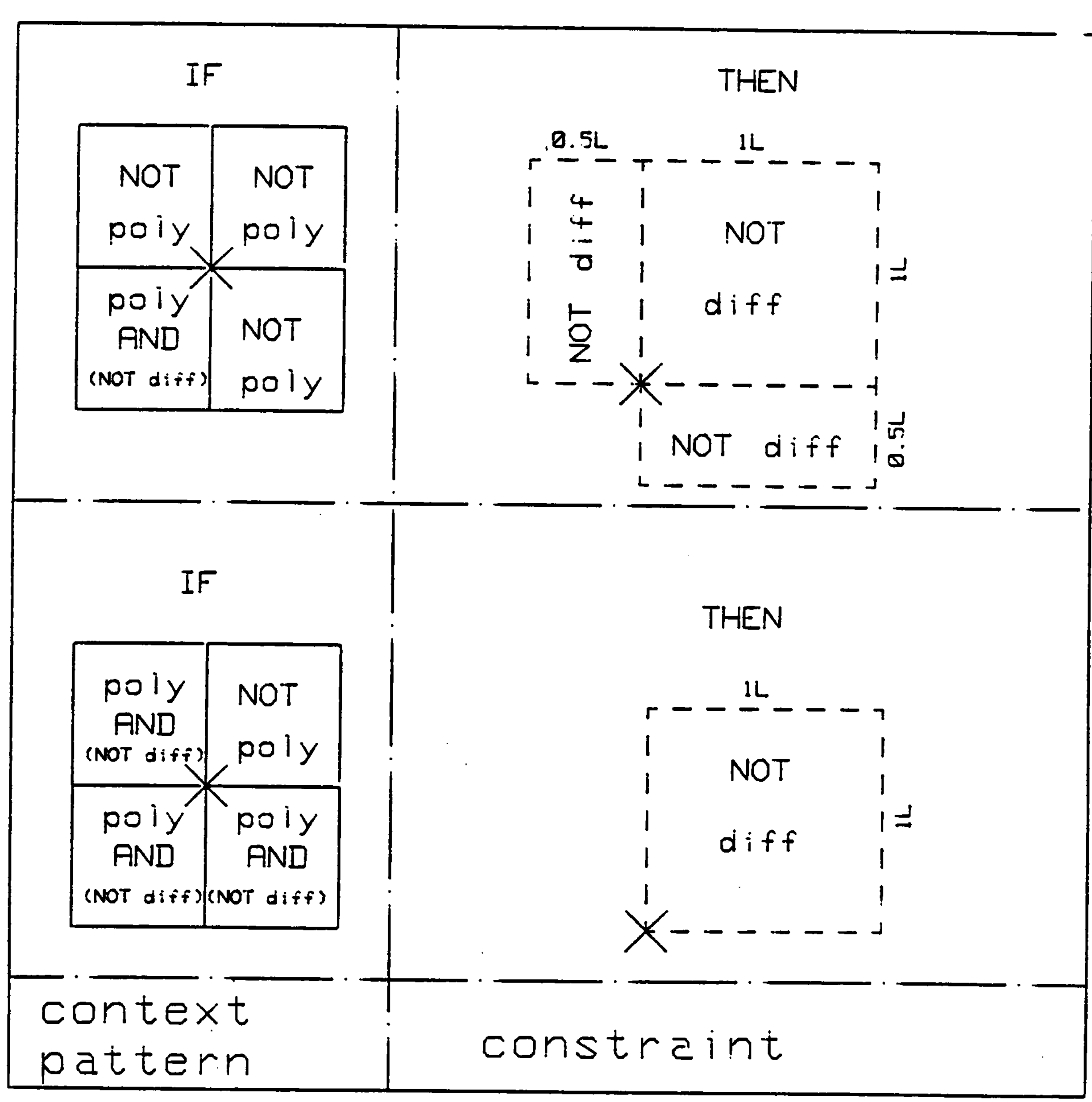


Figure 21. Specification of the interlayer spacing rule between poly and diffusion for the context patterns of poly.

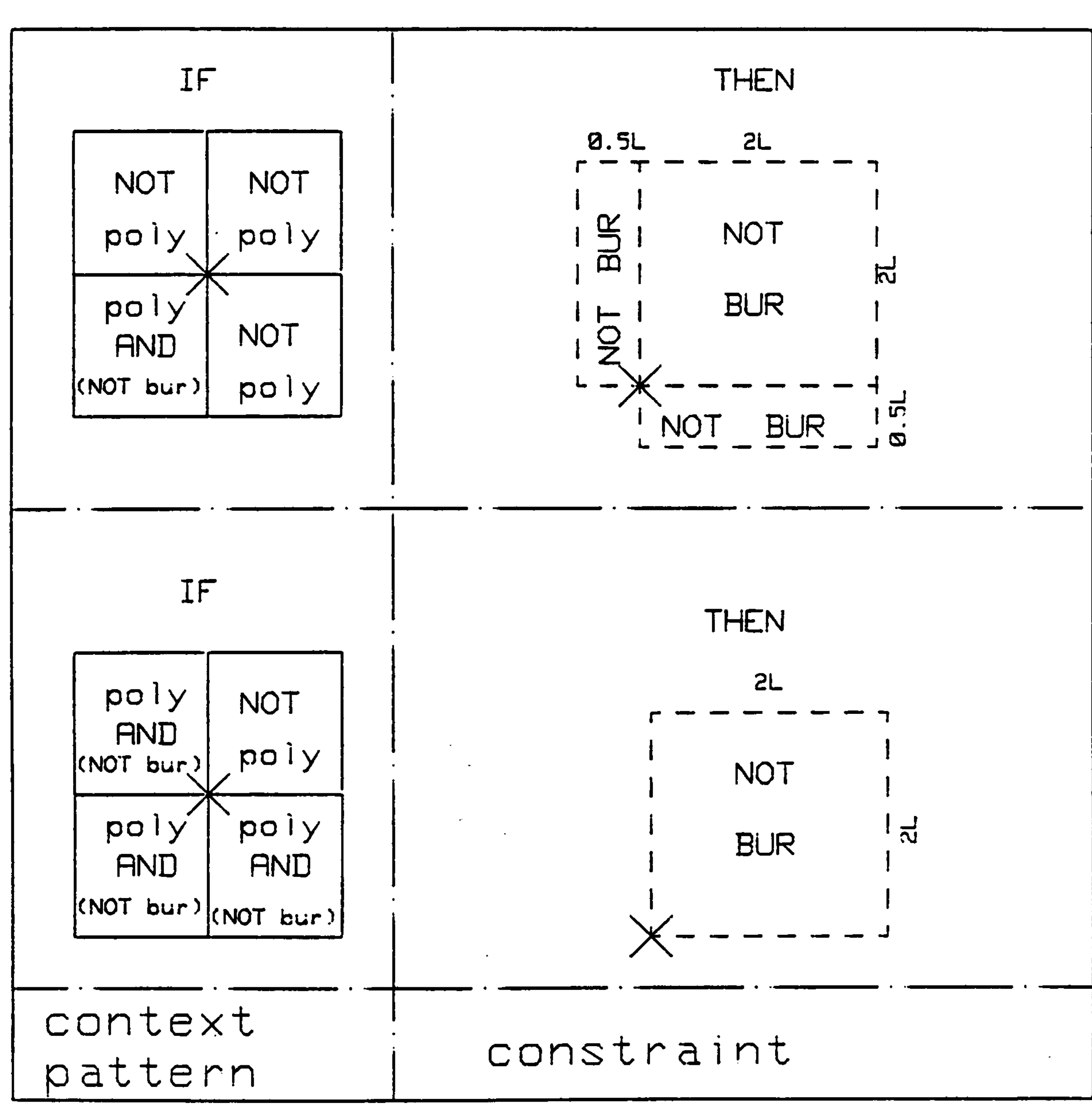


Figure 22. Specification of the interlayer spacing rule between poly and buried cut for the context patterns of poly.

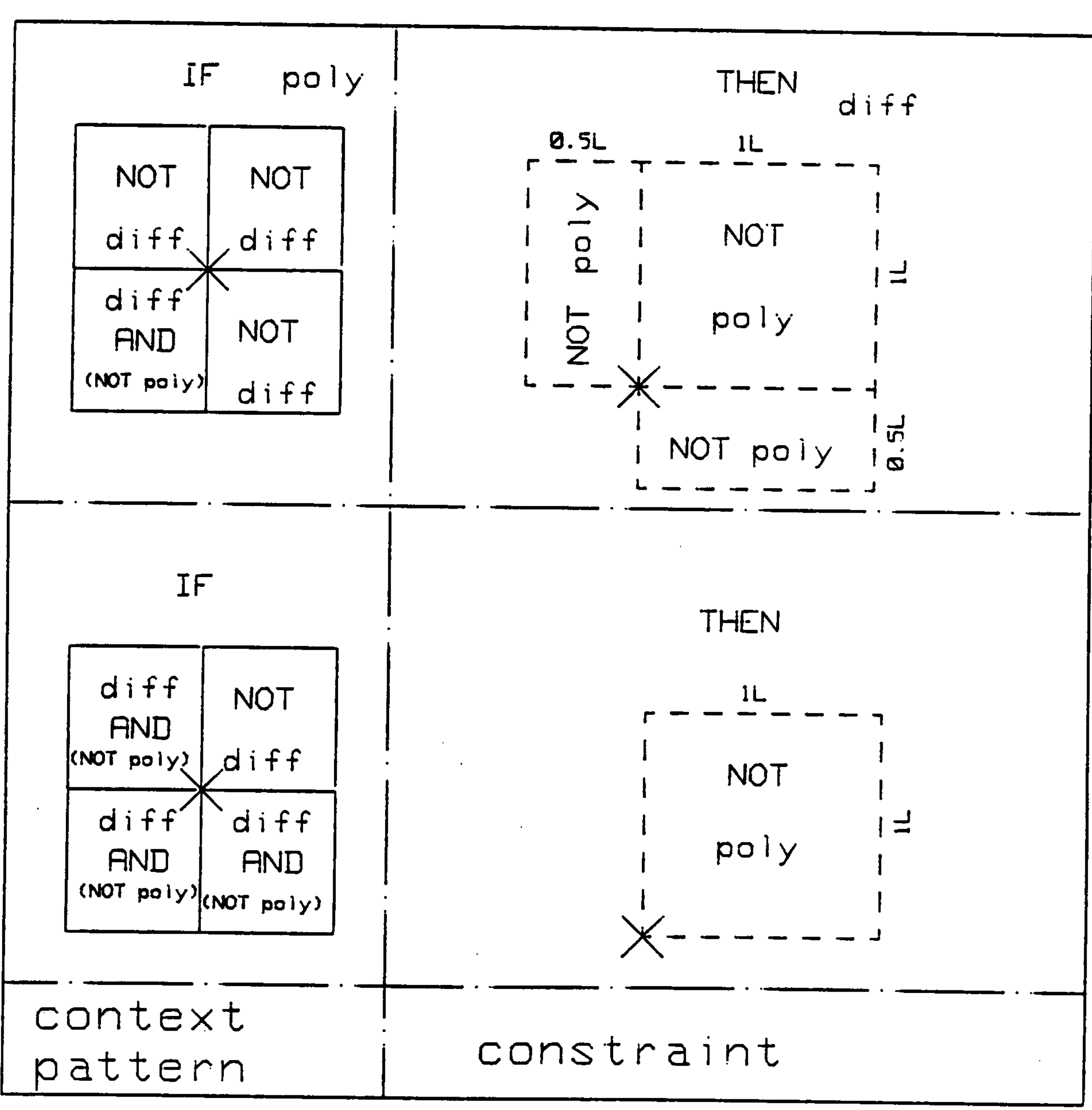


Figure 23. Specification of the interlayer spacing rule between poly and diffusion for the context patterns of diffusion.

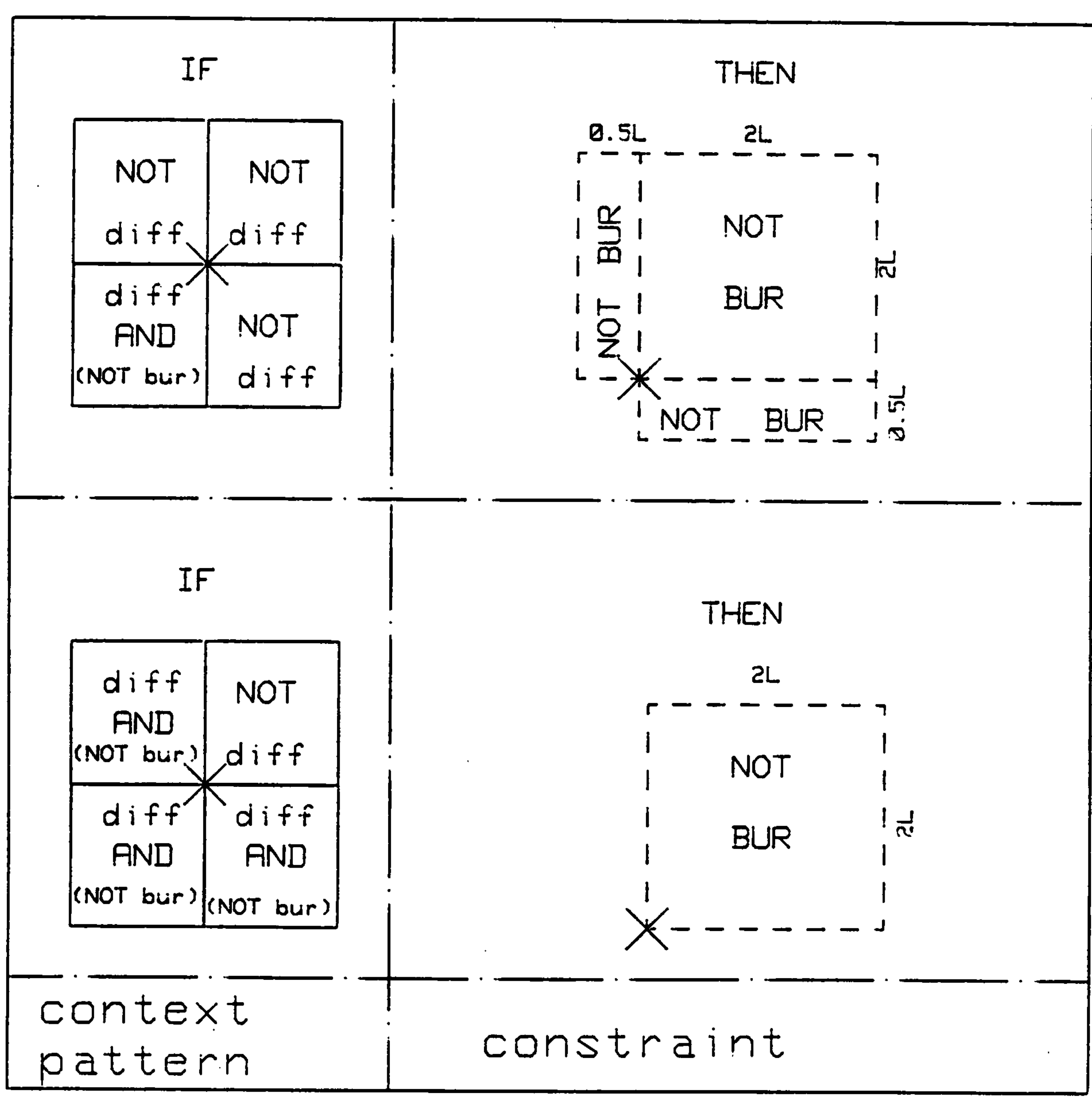


Figure 24. Specification of the interlayer spacing rule between diffusion and buried cut for the context patterns of diffusion.

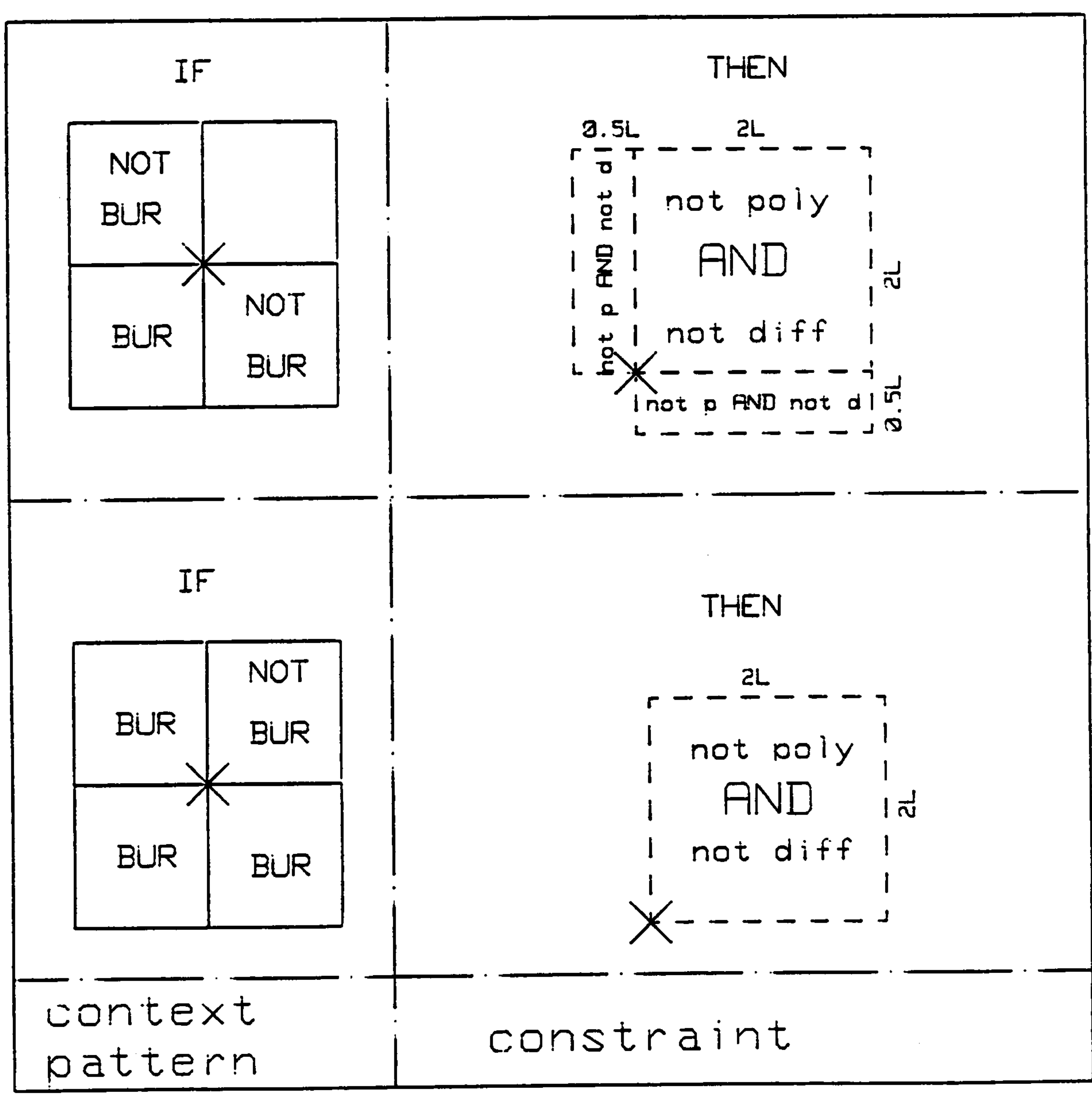


Figure 25. Specification of the interlayer spacing rule for the context patterns of buried cut.

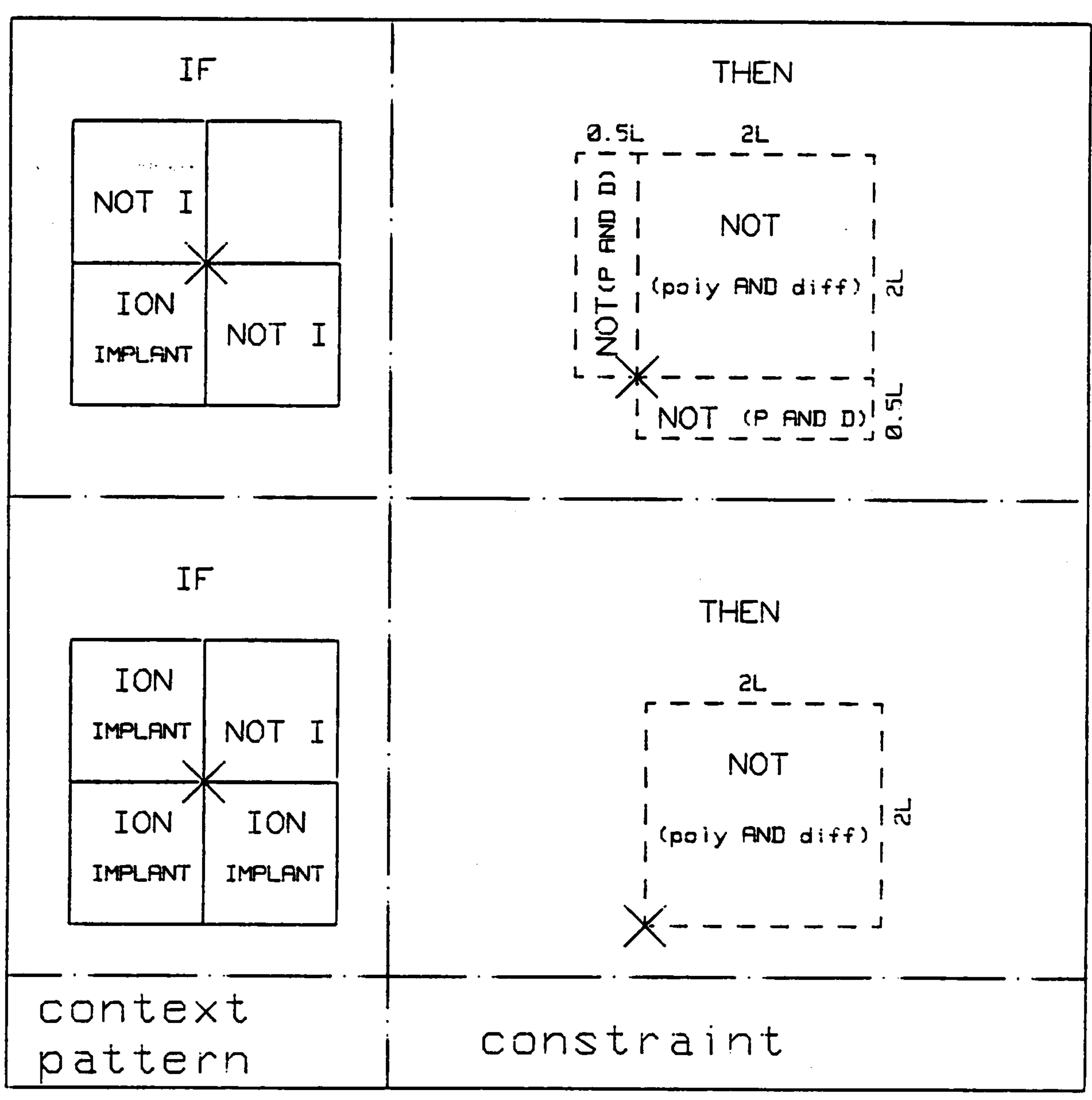


Figure 26. Specification of the interlayer spacing rule between transistor gate and ion-implantation for the context patterns of ion-implantation.

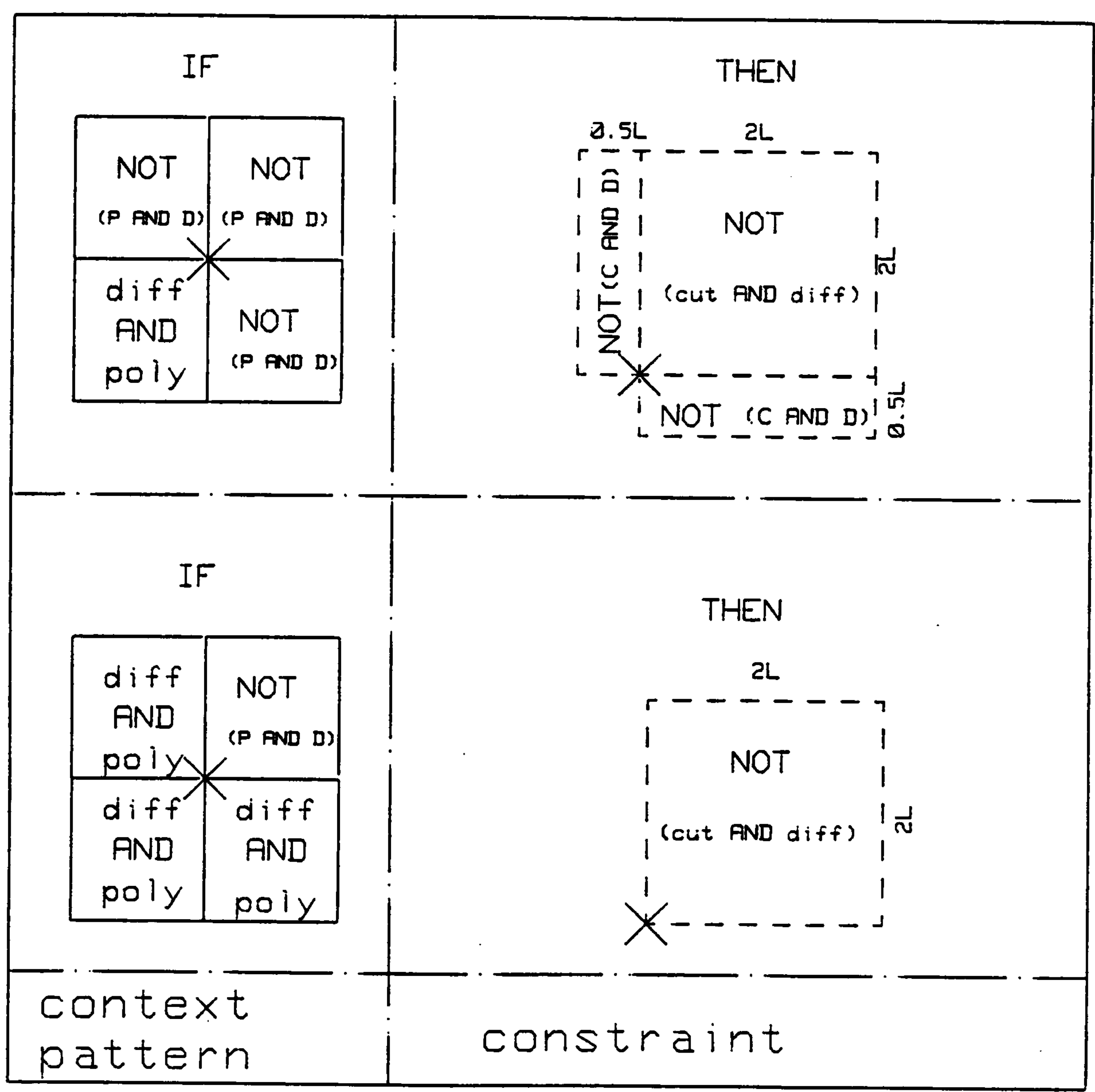


Figure 27. Specification of the interlayer spacing rule between transistor gate and ion-implantation for the context patterns of transistor gate.

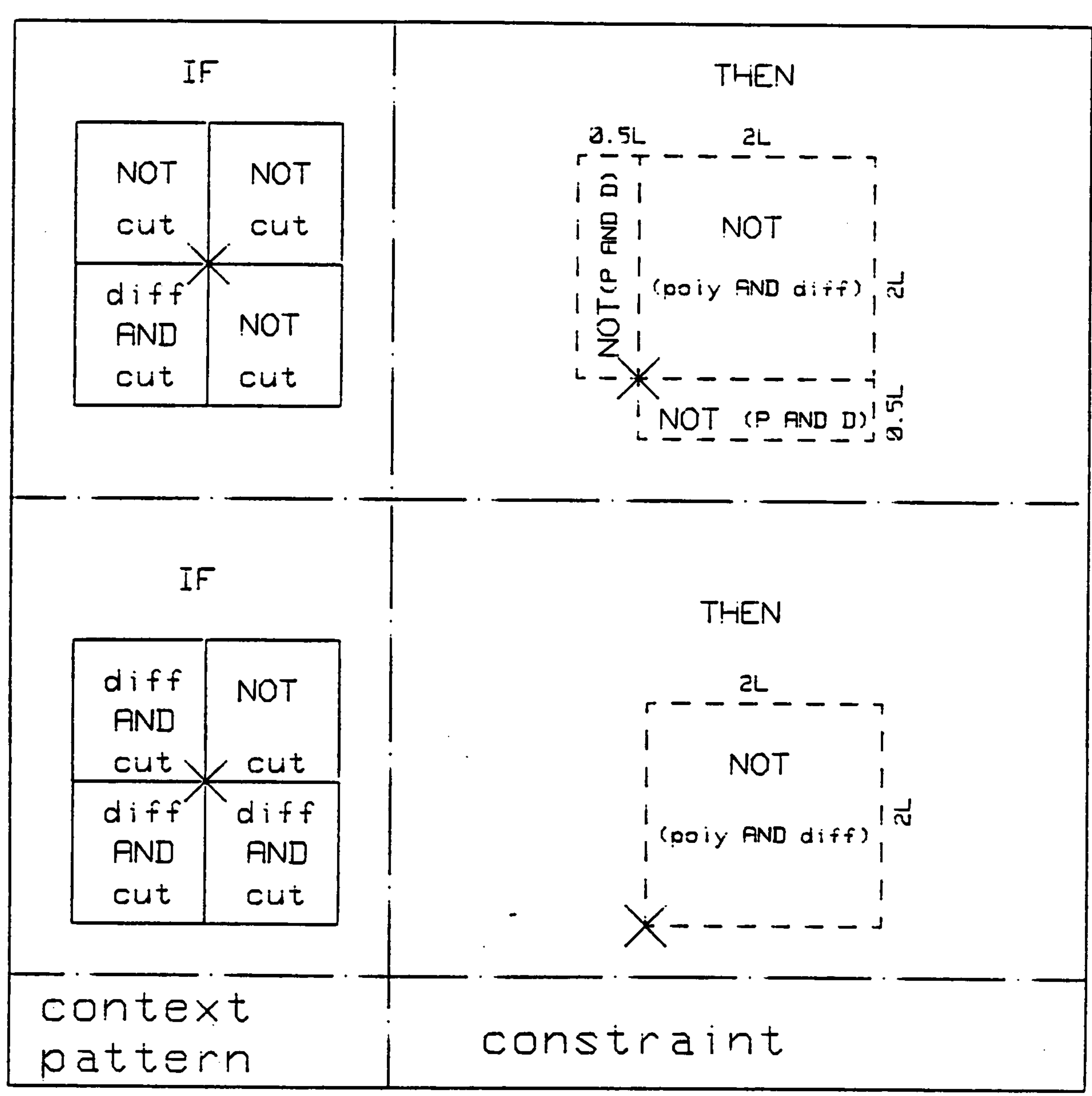


Figure 28. Specification of the interlayer spacing rule between transistor gate and contact cut to diffusion for the combined context patterns of diffusion and contact cut.

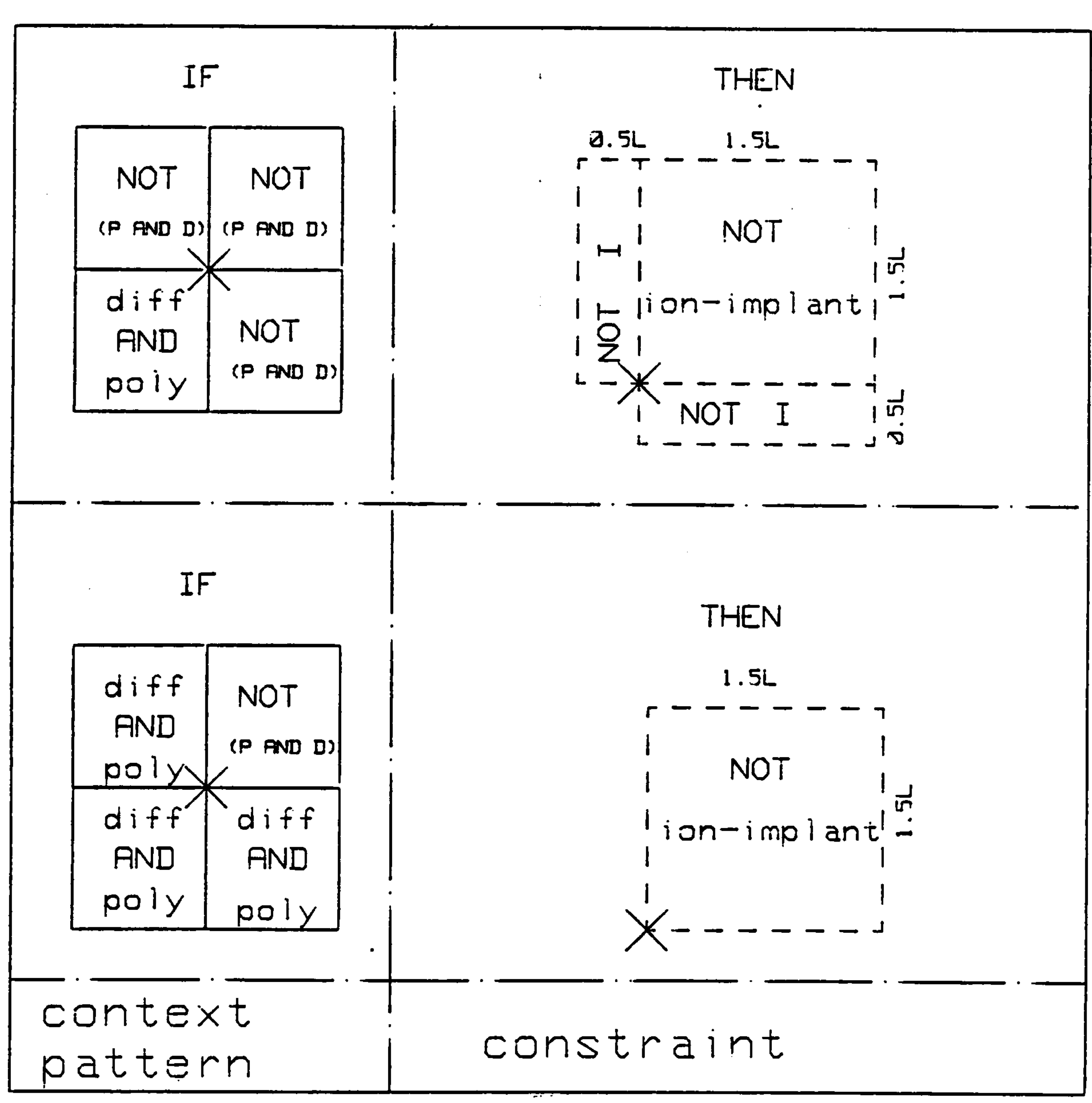


Figure 29. Specification of the interlayer spacing rule between transistor gate and contact cut to diffusion for the context patterns of transistor gate.

mentioned above, there are two exceptions for the context patterns of the ion-implant and buried cut because there is no single layer spacing rule for the buried cut and ion-implant. For other context patterns, the constraints can be obtained by rotation of the constraints in these figures.

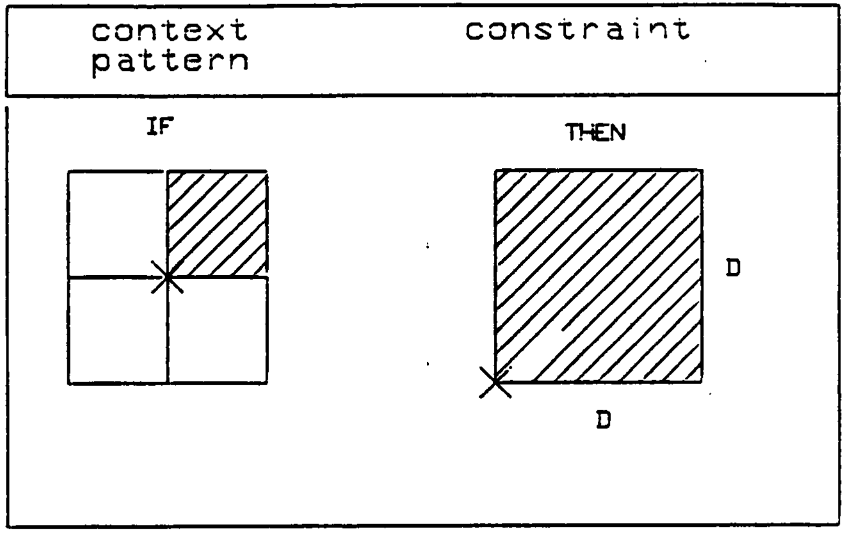
4.2.3 WIDTH RULE SPECIFICATION

The width rule is the requirement for the minimum width of a layer. Identification of the end points of a rectangle is the key to specify this constraint. The width rule constraints for five different types of corners are as follows.

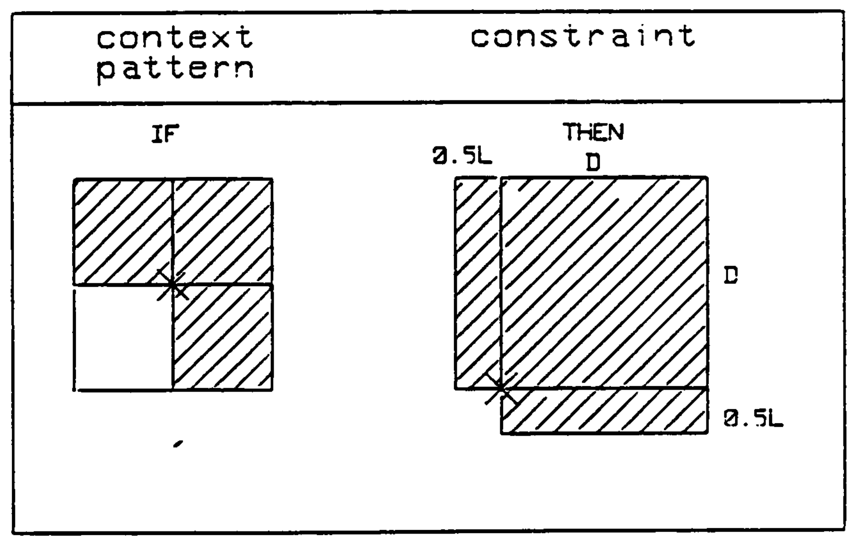
For convex type corners, the layer should be extended by the minimum width in the direction from the corner towards the interior of the rectangle. This is because a convex type context pattern indicates that the corner is the end point of the rectangle. Thus, for a convex corner, the width rule can be specified as shown in Figure 30(a). Plaid corners are not checked for the width rule errors because they are already in violation of the spacing rule.

Since there are three end points in the concave type corners, all the three end points should be extended into the interior of the rectangle in order to satisfy the minimum width requirement. Figure 30(b) shows the constraint for the concave type corner.

In case of one-side corners, we do not need to specify the constraints for the same reason as in the spacing rule specification, i.e., since the entire layout consists of only rectangles, if the width of a layer is satisfies the minimum



(a)



(b)

D : minimum width requirement

(metal : 3L, diffusion,poly,cut : 2L)

Figure 30. (a) Specification of the width rule for a convex corner. (b) Specification of the width rule for a concave corner.

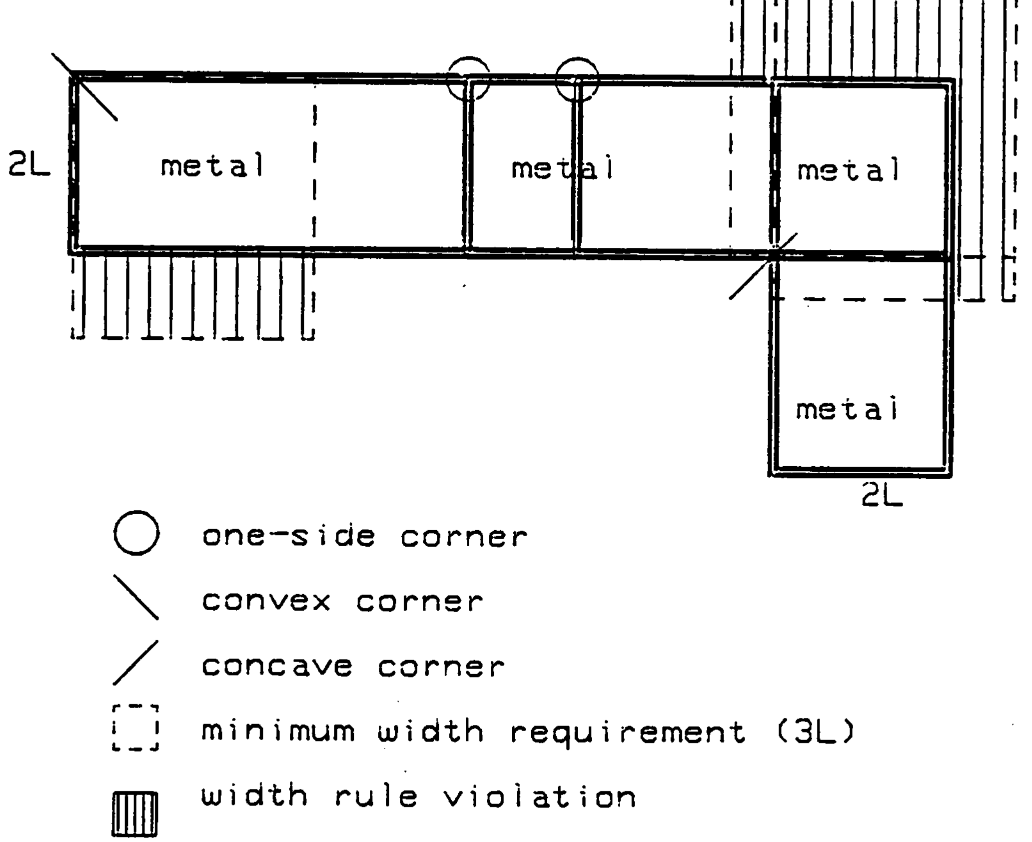


Figure 31. Width rule violations at convex, concave and one-side corners.

width requirement at the end points, then the width at any point between the end points is also satisfactory. In other words, width rule violation at any one-side corner can be detected at the convex or concave corner. Figure 31 is one example of a width rule violation at a one-side corner.

Since the all-filled corner does not contain any end point, the constraint for the width rule is not necessary. Consequently, only eight constraints are necessary to specify the width rule.

4.2.4 TRANSISTOR OVERHANG RULE SPECIFICATION

The transistor overhang rule is defined as the necessary extension of the poly and diffusion around the transistor (TR) gate region. The constraints for poly and diffusion should be determined based on the boundary of a TR gate region.

To explain the TR overhang rule specification, the most common geometric configuration is shown in Figure 32. As we see in this example, two features must be considered in order to specify the TR overhang rule. One is the direction of the extension of poly and diffusion and the other is the type of the layer which should be extended. The direction of the extension should be determined based on the boundary of the TR gate region. The layer type should be same as the layer which forms the border line with the TR gate region. In other words, if poly is abutted to the TR gate region, poly should be extended in the opposite direction to where the TR gate region exists. On the other hand, diffusion is abutted, diffusion should be extended. Considering these two things, TR overhang rule can be specified by following approach.

Since two layers are involved in the TR overhang rule, in order to specify the constraints, all possible combined context patterns of poly and diffusion should be found first. There are 15 possible context patterns for one layer at a

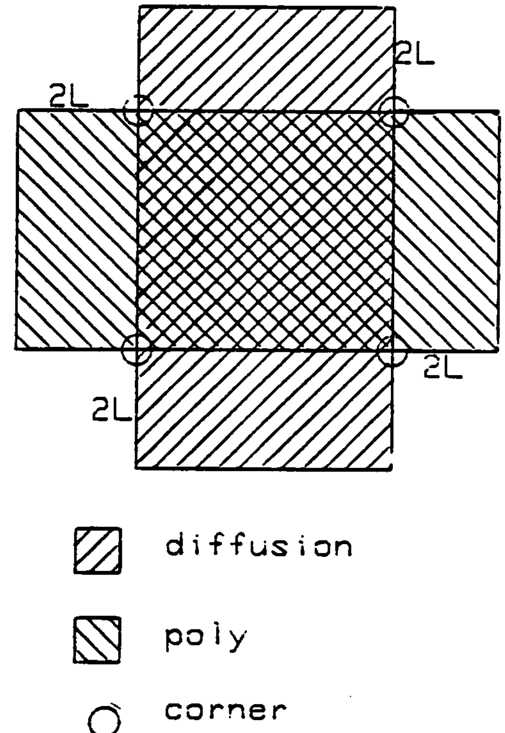


Figure 32. Corners in a transistor.

single corner. Thus, there are 15 possible combined context patterns of poly layer for one context pattern of diffusion. Figure 33(b) shows the 15 combined context patterns for the one convex type context pattern of diffusion in the Figure 33(a).

The 15 combined context patterns are divided into three groups according to the following characteristics. Group (A) contains context patterns which have no overlap between poly and diffusion. Thus, these context patterns cannot be considered as transistors, and TR overhang rule checking is unnecessary. However, all these context patterns violate the interlayer spacing rule between poly and diffusion. Therefore, the interlayer spacing rule violation should be reported. In group (B) and (C) context patterns, there is at least one overlapped quadrant. Since there exists a vacant quadrant which is abutted to the overlapped quadrant (i.e., TR gate region) in the context patterns of group (B), these context patterns already violate the TR overhang rule. Therefore, for the context patterns included in group (B), TR overhang rule error can be reported without checking any constraint. On the other hand, the context patterns of group (C) have the possibility of forming legal transistors. Thus, for these context patterns, constraints need to be specified. Figure 34(a) shows the constraints for the context patterns of group (C).

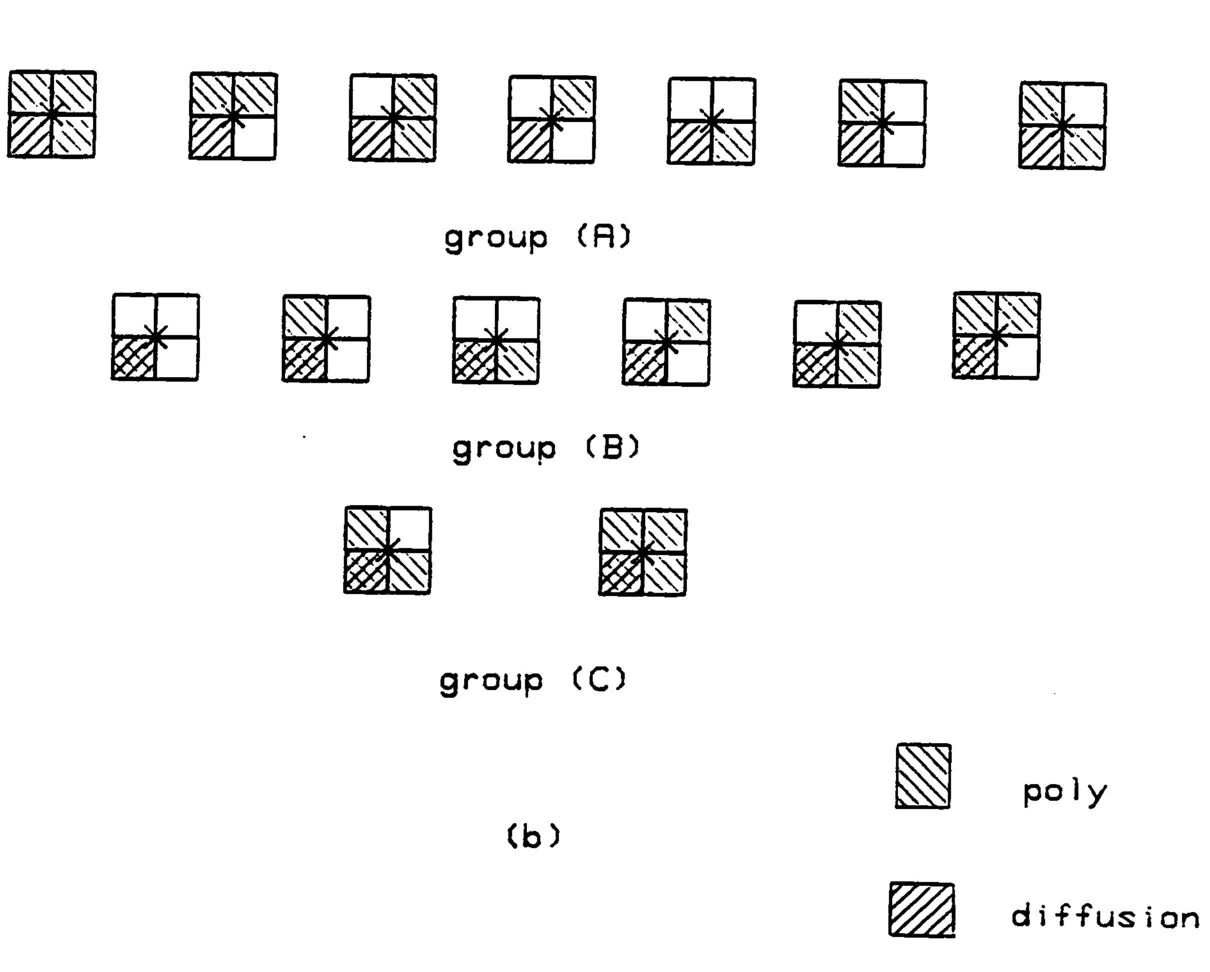


Figure 33. Combined context patterns of diffusion and poly for a convex type context pattern of diffusion.

For the other three convex type context patterns of the diffusion, all the possible combined context patterns can be obtained by rotating the combined context patterns in Figure 33 by 90, 180, 270 degrees as mentioned earlier. As a result, all possible combined context patterns can be divided into three groups and the necessary constraints can be obtained. Figure 34(b), (c), (d) show the legal context patterns and corresponding constraints obtained by rotation.

For the context patterns obtained by combining poly context patterns with a one-side type context patterns of diffusion, the TR overhang rule can be specified using an approach similar to the one for a convex diffusion corner described above. As the first step, the 15 combined context patterns for a one-side diffusion corner of Figure 35(a) are divided into three groups shown in Figure 35(b). By the reasoning used in a convex diffusion corner, group (A) context patterns contain the interlayer spacing rule errors and all group (B) context patterns violate the TR overhang rule. Only group (C) patterns can form legal transistors. Figure 36 shows the corresponding constraints for these legal patterns. Again, by rotation we can find the combined context patterns and constraints for the other three one-side context patterns of diffusion.

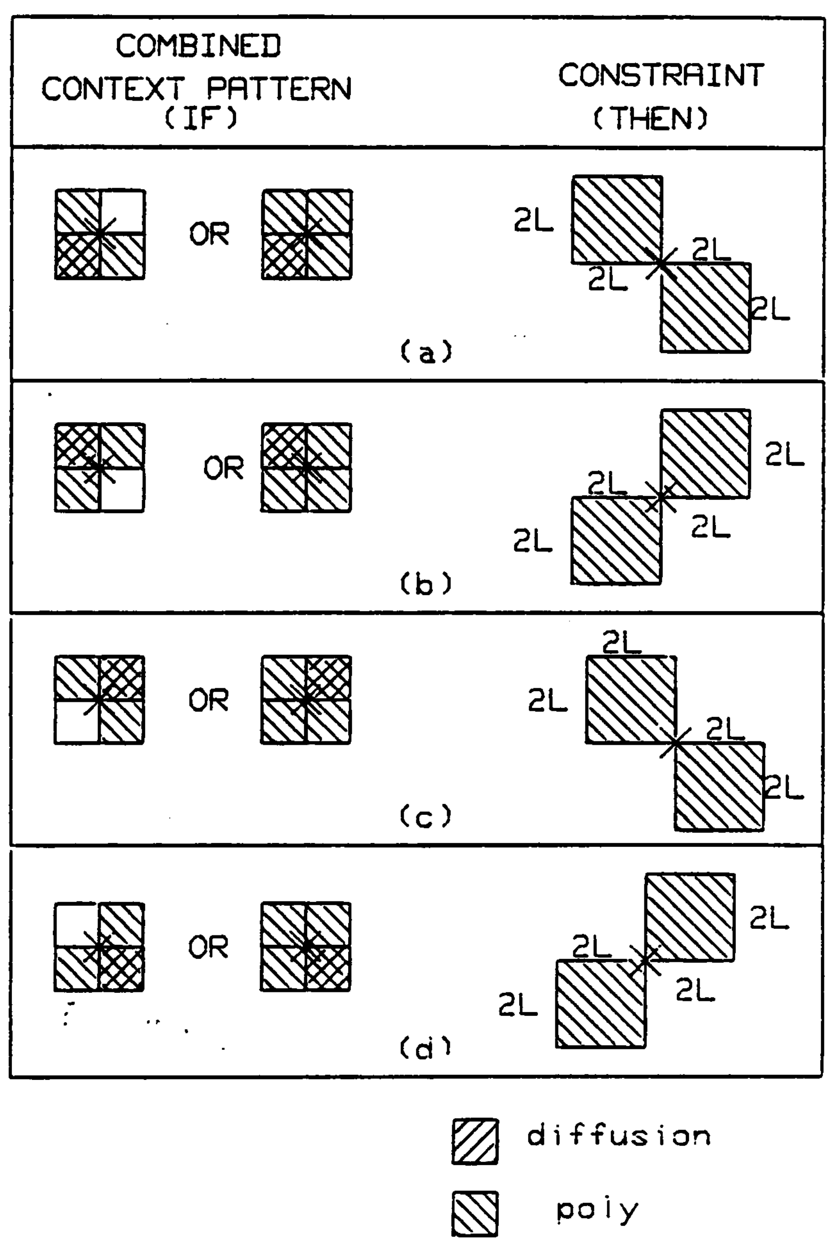


Figure 34. Specification of the transistor overhang rule for the combined context patterns of group (C) in Figure 33.

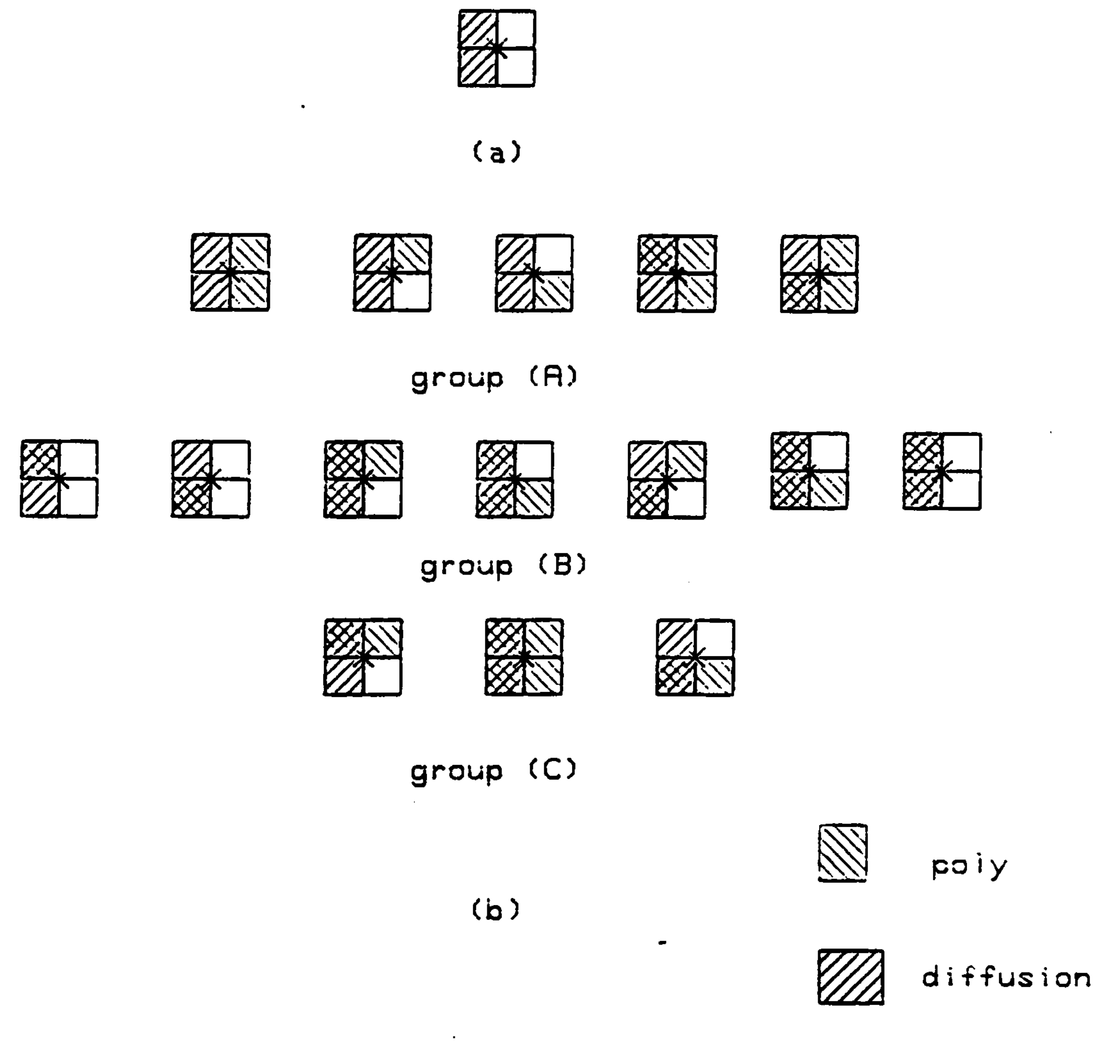


Figure 35. Combined context patterns of diffusion and poly for a one-side context pattern of diffusion.

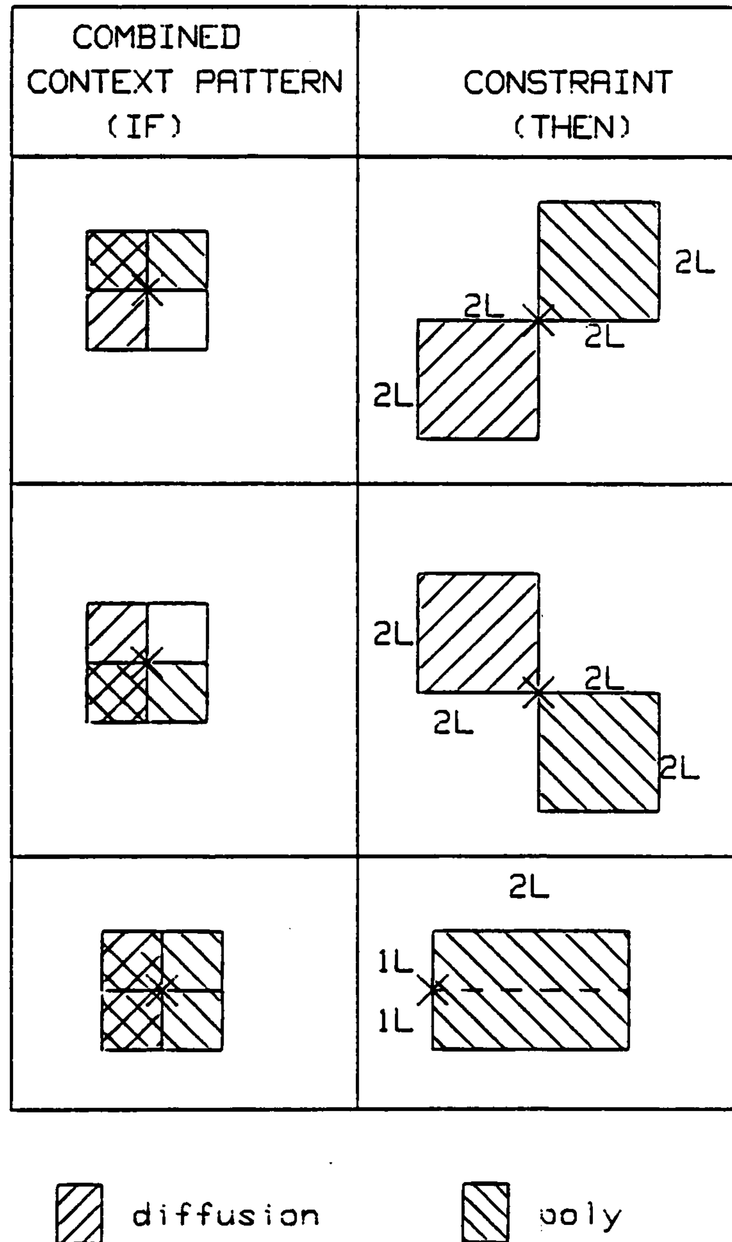


Figure 36. Specification of the transistor overhang rule for the combined context patterns in Figure 35.

Using the same approach, we can get the legal combined context patterns and corresponding constraints for the concave type context patterns of diffusion. Figure 37 shows one concave type context pattern of diffusion and the corresponding 15 combined context patterns. The constraints for the legal combined context patterns are shown in Figure 38. Three more sets of constraints can be found by rotation.

If the four quadrants are filled with diffusion, there are 15 combined patterns. These 15 patterns can be divided into five groups in Figure 39 based on the overlap pattern. Notice that the single layer spacing rule of poly layer is already violated in two plaid type overlap patterns. Thus, these plaid type overlap patterns cannot be considered as legal transistors. Figure 40 shows the constraints for one combined pattern in each group. The constraints for other combined patterns can be found by rotation of the constraints in Figure 40. If all four quadrants are overlapped as in Figure 39(e), the constraint is not necessary because this context pattern does not contain the boundary of the gate region.

Since two plaid type context patterns of diffusion contain the spacing rule violation, no combined context pattern can be considered as a legal transistor. Moreover, this spacing rule violation will be detected by the single layer

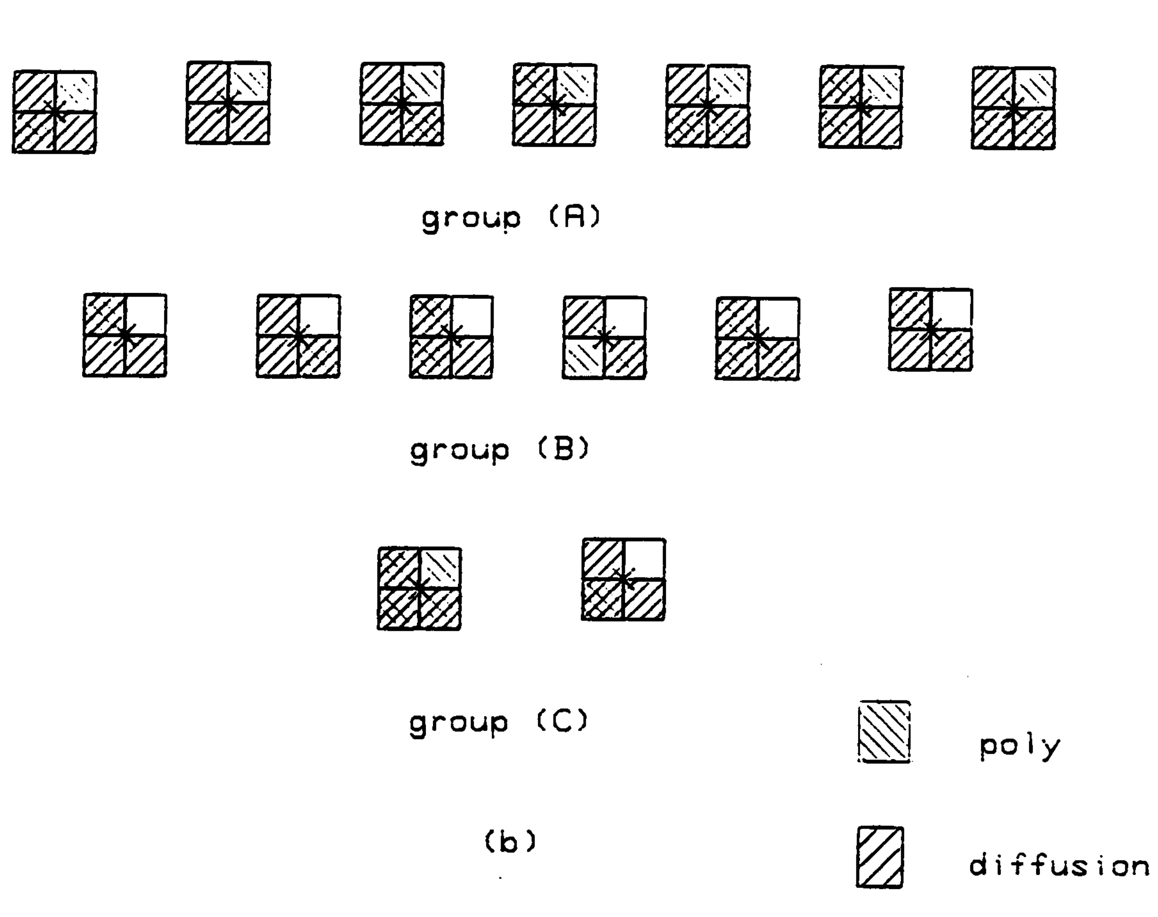


Figure 37. Combined context patterns of diffusion and poly for a concave type context pattern of diffusion.

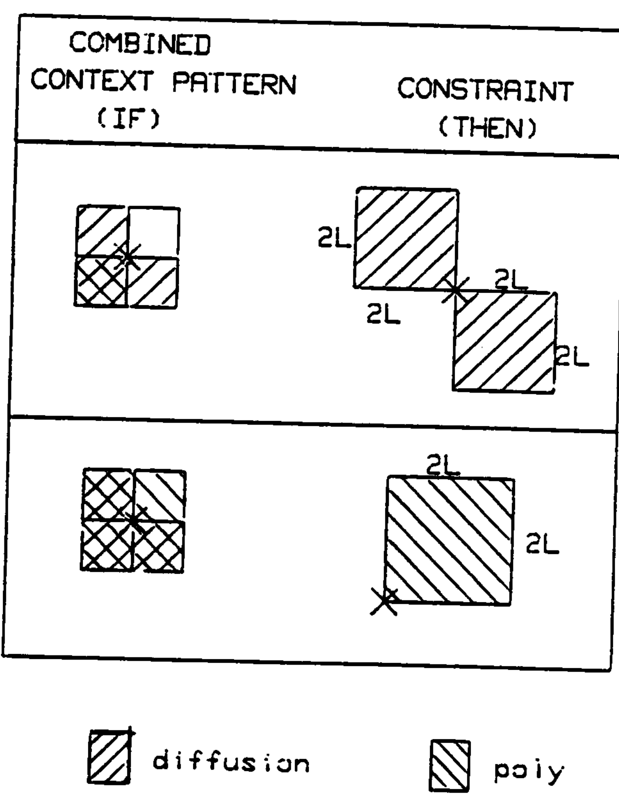


Figure 38. Specification of the transistor overhang rule for the combined context patterns in Figure 37.

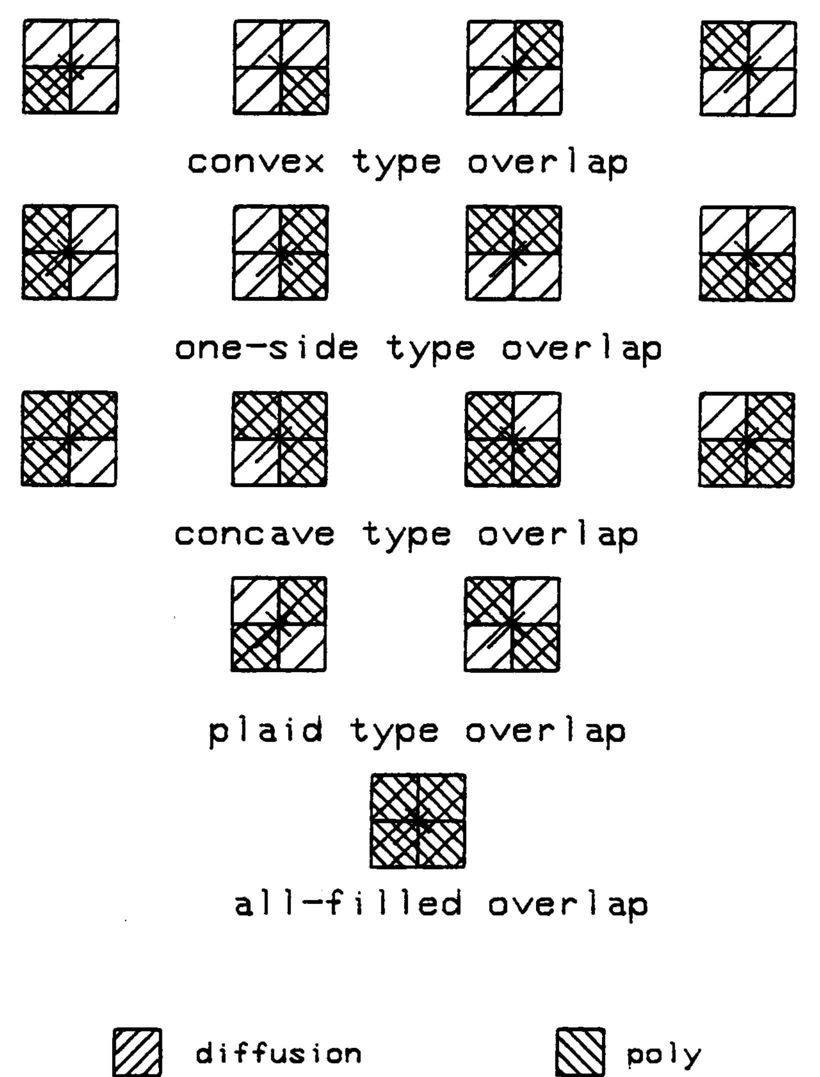


Figure 39. Combined context patterns of diffusion and poly for the all-filled type context pattern of diffusion.

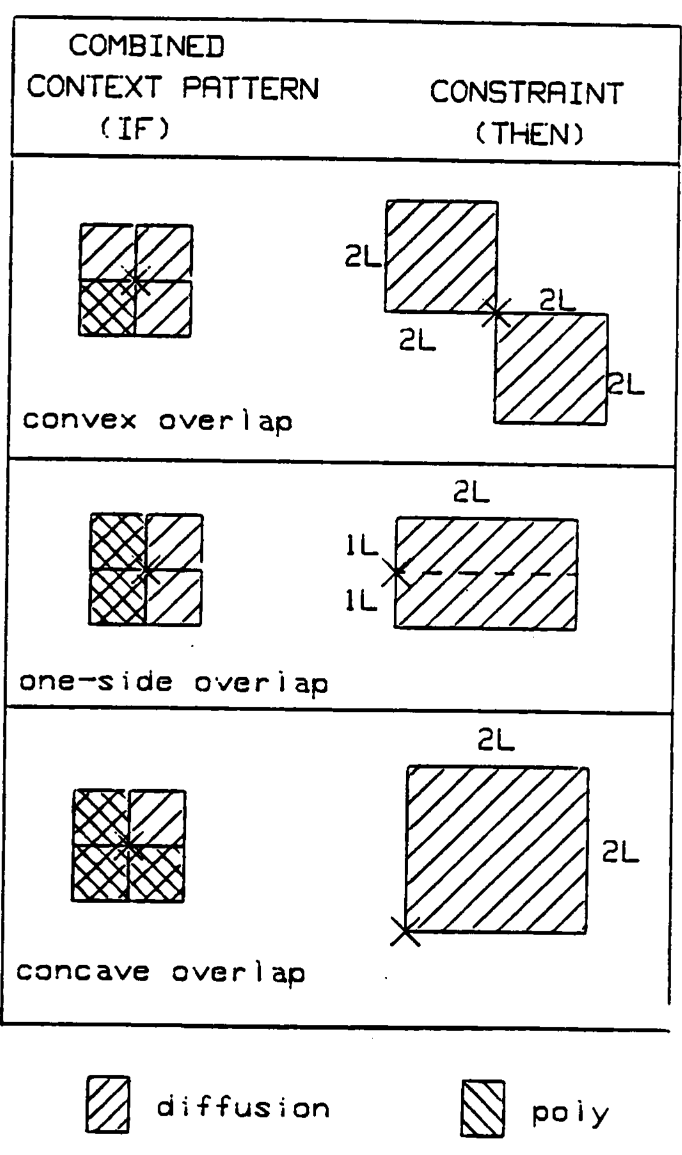


Figure 40. Specification of the transistor overhang rule for the combined context patterns in Figure 39.

spacing rule checking. Therefore, it is not necessary to check the TR overhang rule for the combined context patterns with these plaid type context patterns.

As shown above, many combined context patterns do not need to be checked. Thus, TR overhang rule can be specified very easily by grouping and rotation.

4.2.5 ION-IMPLANT OVERHANG RULE SPECIFICATION

The ion-implant overhang rule states the requirement for the minimum extension of the ion-implant beyond the TR gate region for proper depletion mode transistors. In order to make a proper depletion mode transistor, a proper enhancement mode transistor should first be formed. If the TR overhang rule is violated, ion-implant overhang has no meaning because the gate region which is referenced to build the ion-implant overhang may be incorrect. In other words, when the designer corrects the TR overhang rule error, TR gate region can be changed and ion-implant should be changed based on the TR gate region. Therefore, it can be said that it is not necessary to check the ion-implant rule for an illegal transistor. It means that we do not need to specify the constraints for illegally combined context patterns of poly and diffusion.

For legal combined context patterns, if the ion-implant layer is found at a corner, it indicates a depletion mode transistor. The constraints for the ion-implant overhang rule can be specified in a manner similar to the TR overhang rule. Since the ion-implant overhang rule should be specified based on the TR gate region, the overlapped context pattern is used for specifying the constraints. Figure 41 and Figure 42 shows the constraints for each type of over-

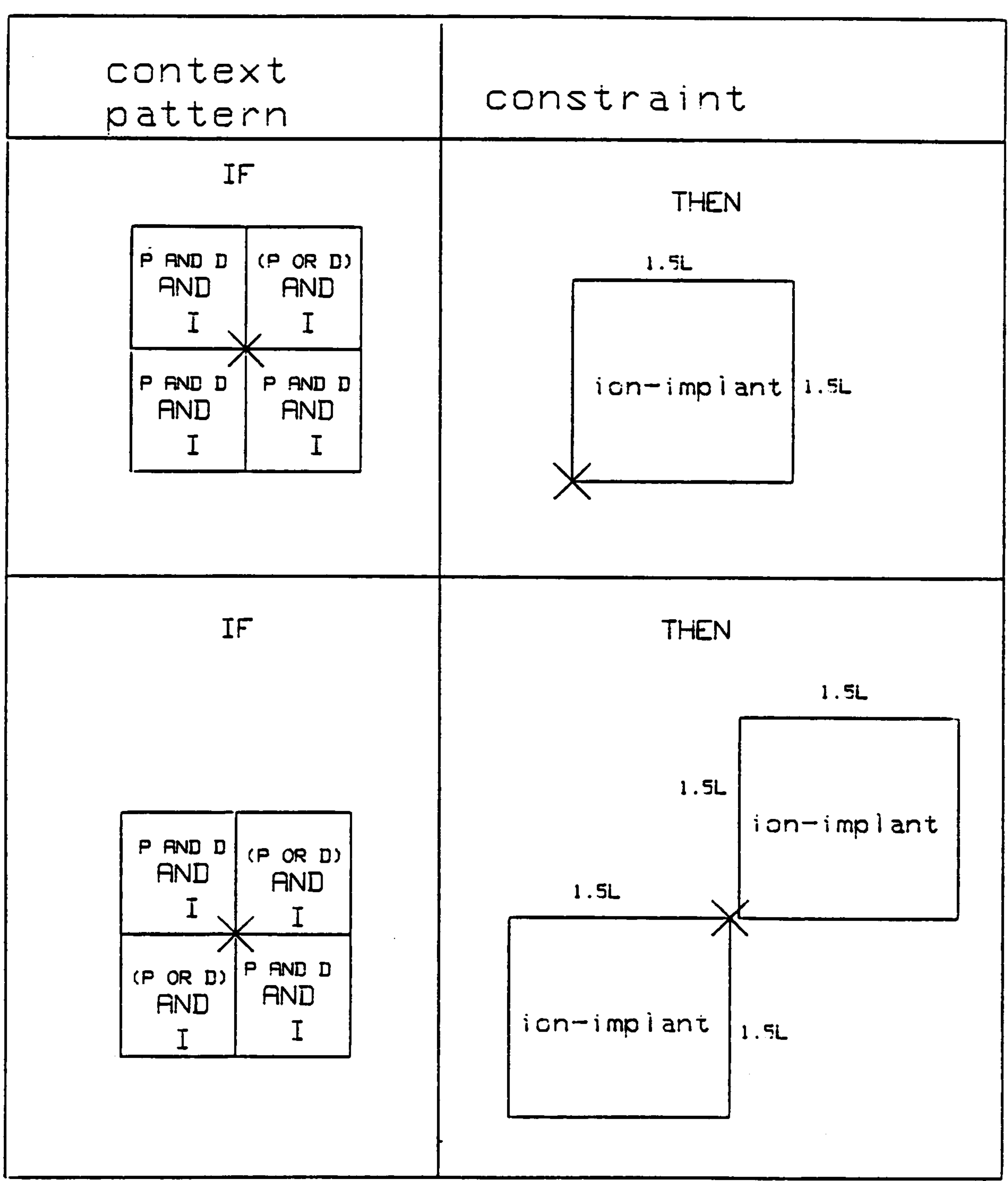


Figure 41. Specification of the ion-implant overhang rule for the context patterns of depletion mode transistors.

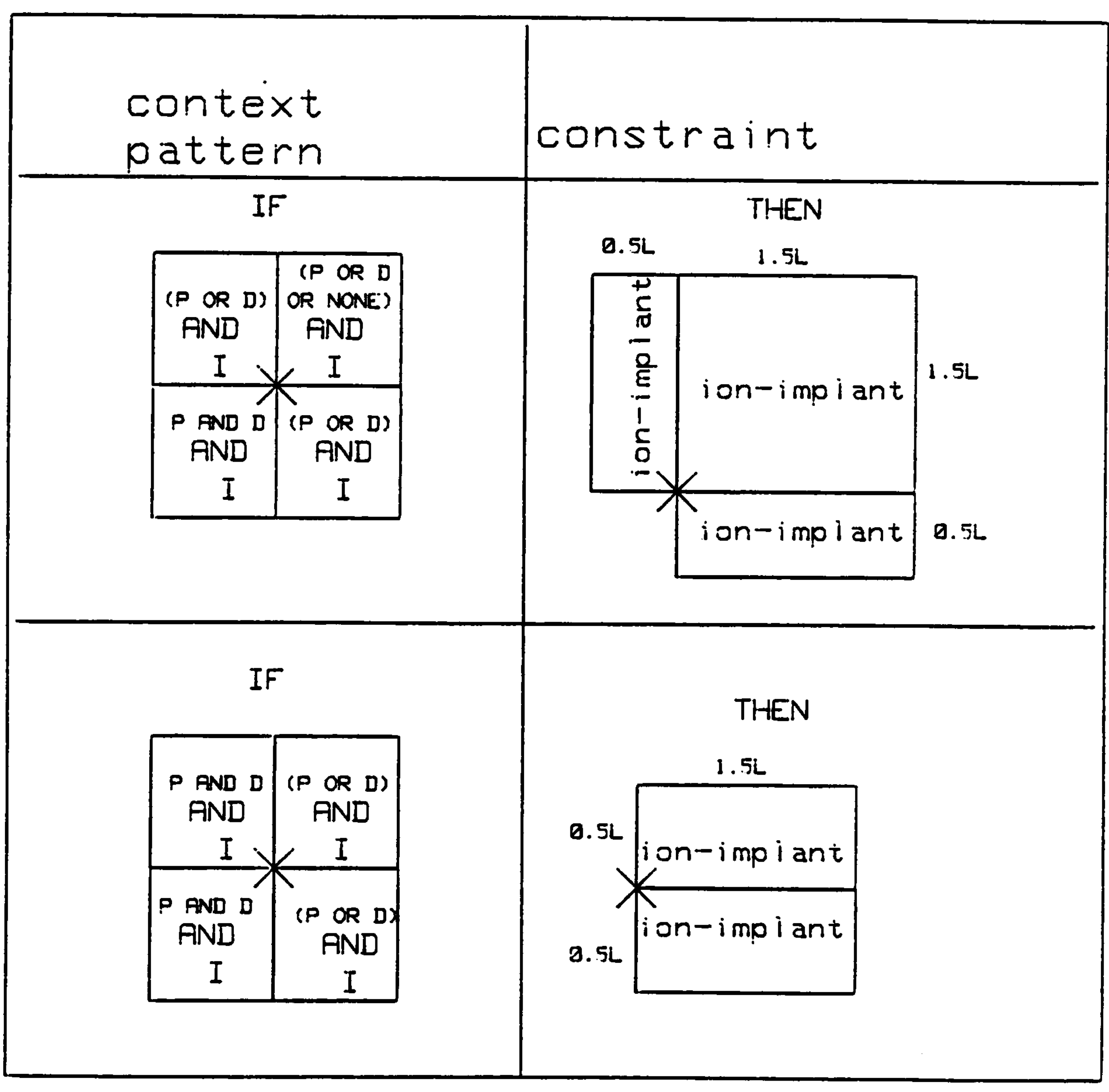


Figure 42. Specification of the ion-implant overhang rule for the context patterns of depletion mode transistors.

lapped context patterns. The entire set of constraints can be found by rotating these constraints. If four quadrants are covered with poly-diffusion overlap and ion-implant exist at the corner, these four quadrants should be covered with ion-implant region.

4.2.6 CONTACT CUT SURROUNDING RULE SPECIFICATION

Since the contact cut has a particular shape as shown in Figure 43, rule specification can be much simpler in spite of the presence of three layers. In Figure 43, we immediately note that all corners of a contact cut are convex corners. If the contact cut is made by connecting two or more small contact cuts, one-side or all-filled corners can exist as shown in Figure 44. Although a contact cut which contains the concave type corner is legal as in Figure 45, this kind of contact cut wastes chip area. Thus, if a concave contact cut is found, it is better to warn the designer to correct the shape of the contact cut. From this point of view, we can say that only convex, one-side and all-filled contact cut corners are legal. For these three legal context patterns, constraints can be specified as in Figure 46 - Figure 49.

In context patterns of Figure 47, the reason for the specification "NOT-POLY AND DIFF" and "POLY AND NOT-DIFF" is to distinguish a pure contact cut from a butting contact. As we know, there are four one-side type corners in the butting contact and both layers, poly and diffusion, exist at these four corners as shown in Figure 51 in the next section. Moreover, if a contact cut corner without poly or diffusion is found, a surrounding rule violation should be reported.

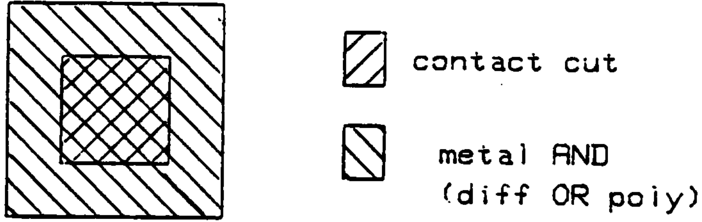


Figure 43. Contact cut.

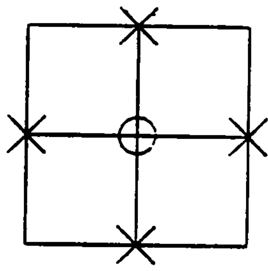


Figure 44. One-side and all-filled corners in the contact cut containing more than one rectangle.

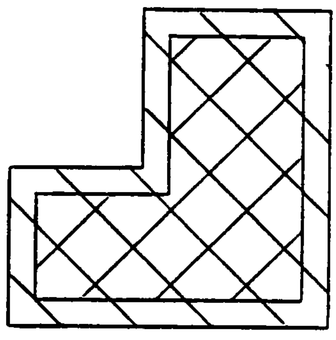


Figure 45. A contact cut containing a concave type corner.

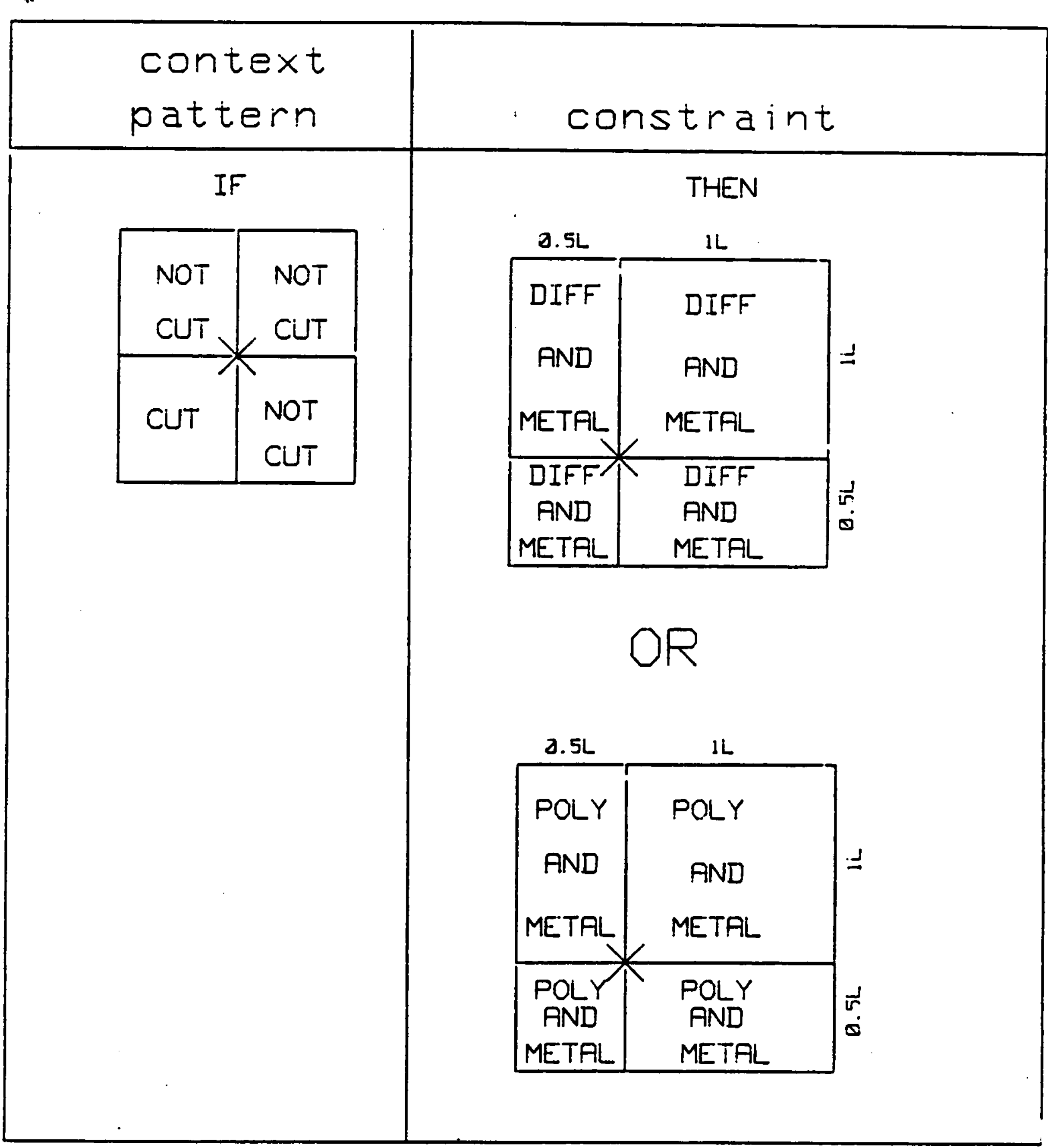


Figure 46. Specification of the contact cut surrounding rule.

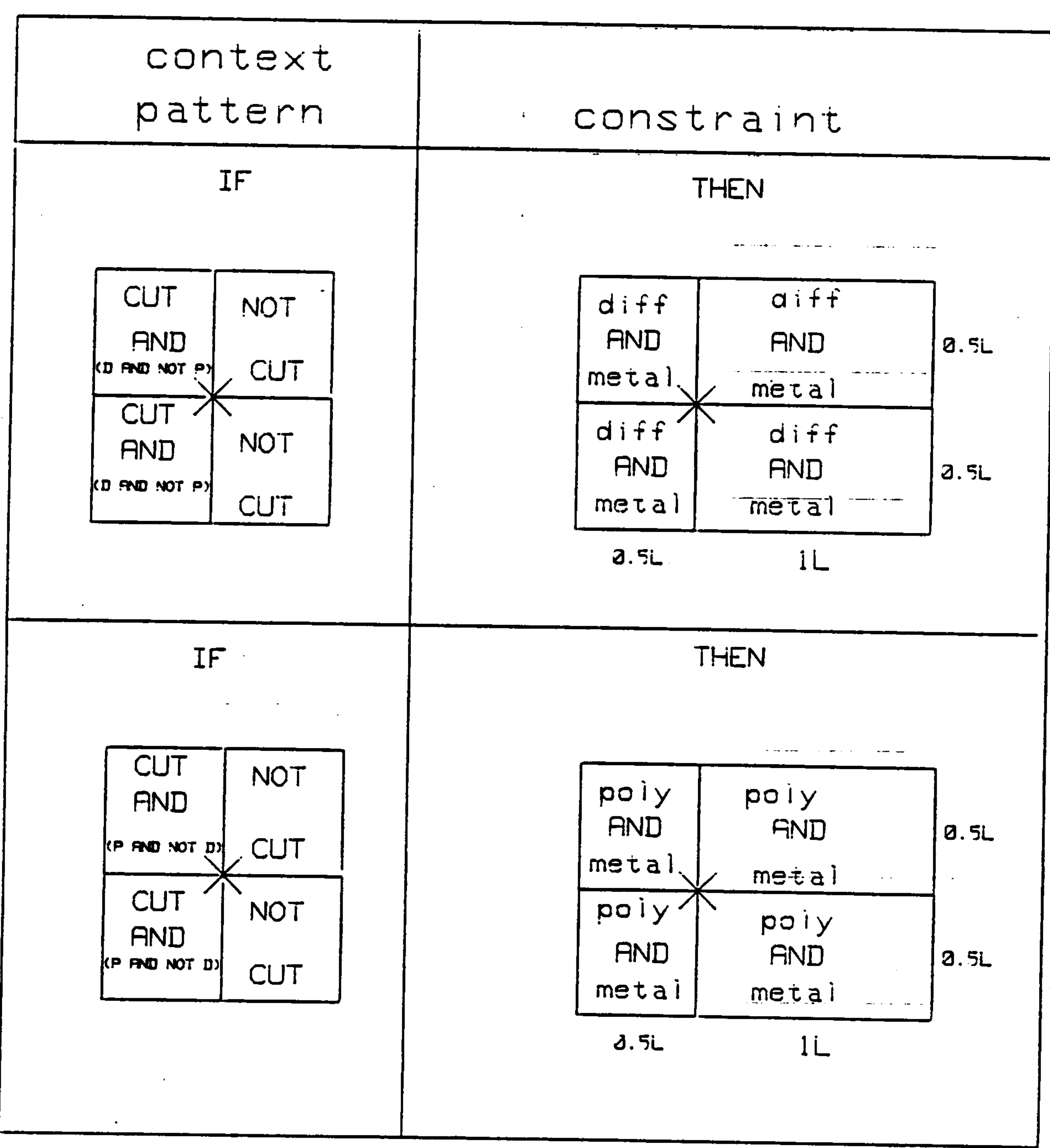


Figure 47. Specification of the contact cut surrounding rule.

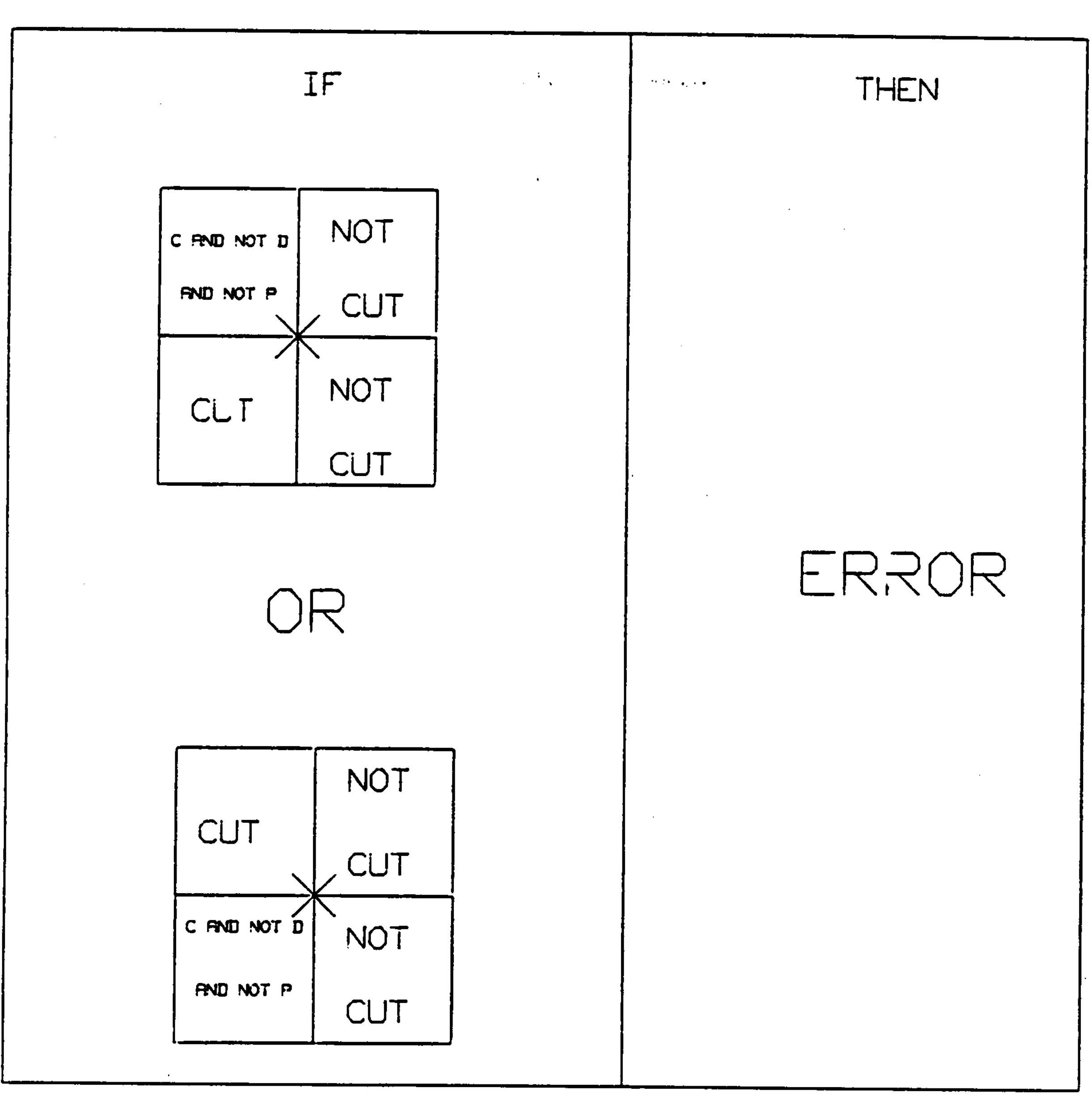


Figure 48. Specification of the contact cut surrounding rule.

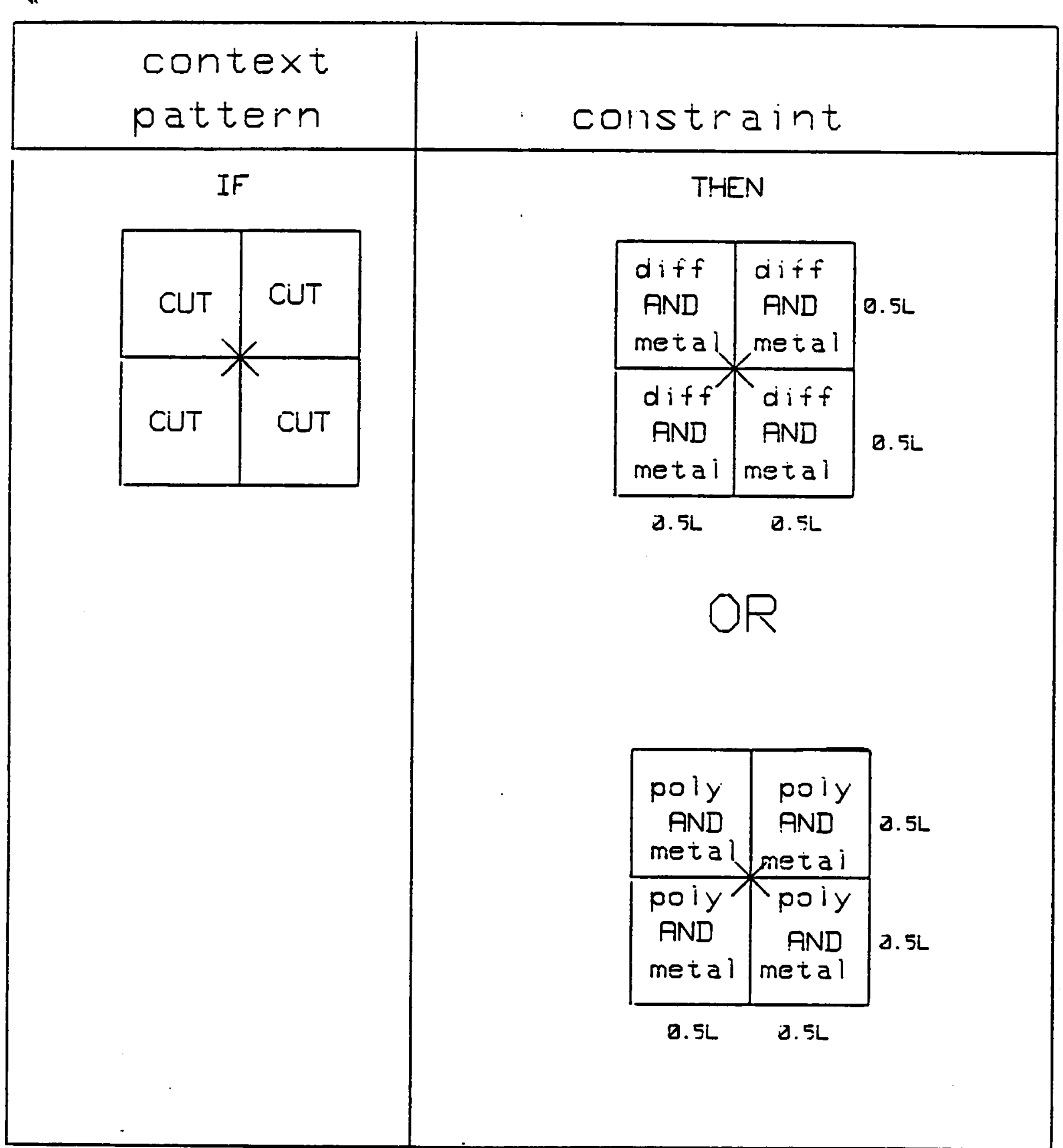


Figure 49. Specification of the contact cut surrounding rule.

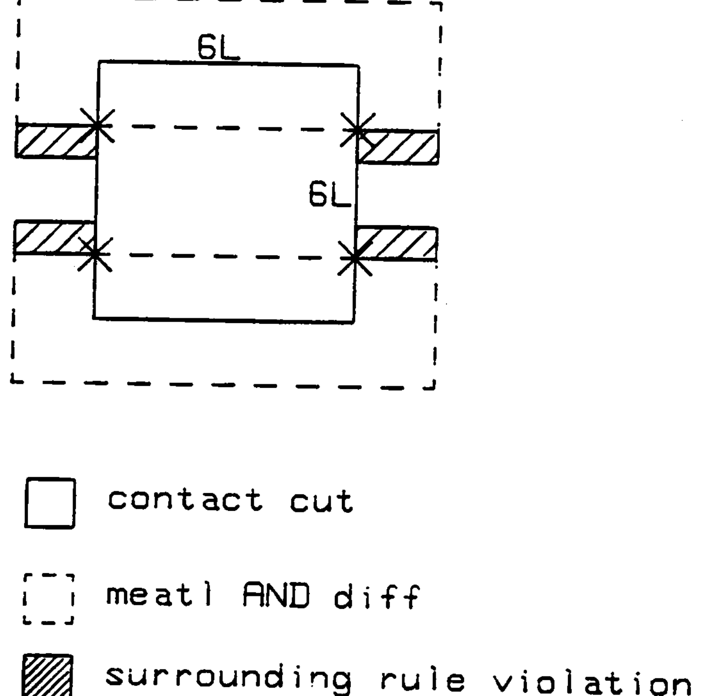


Figure 50. Contact cut surrounding rule violations detected at one-side corners.

The constraints described above are sufficient to detect the errors in larger contact cuts. Suppose that there is a 6λ X 6λ contact cut in which some area is not covered by metal and diffusion properly as shown in Figure 50. Since, the metal layer is disconnected, the one-side type corners marked with X will be created. Thus, these surrounding rule violation can be detected by the constraints shown in Figure 47. Consequently, all kinds of surrounding rule violations for contact cut can be detected.

4.2.7 BUTTING CONTACT RULE SPECIFICATION

The butting butting contact rule can be divided into two parts. The first is the surrounding requirement as in the pure contact cut while the second is an overlap requirement between poly and diffusion. For the surrounding requirement, the constraints in Figure 46 - Figure 49 in the contact cut surrounding rule can be used to detect all the surrounding rule violations since this requirement is same as the surrounding requirement for the pure contact cut. The shaded region in Figure 51(a) indicates the area which is covered by the constraints of the pure contact cut. Within the unshaded region, which is not covered by the surrounding constraint, a set of proper constraints should be specified for all the eight corners marked with Xs and circles in Figure 51(b).

In Figure 51(b), poly-diffusion overlaps are found at the corners marked with circles. The overlap between poly and diffusion also exists in the transistor gate region. To distinguish between two cases, the contact cut is searched within the region which is a 1.5λ X 1.5λ square extended from the corner of poly-diffusion overlap. If the overlap is identified as the part of the butting contact, the butting contact rule checking should be performed. Otherwise, the transistor overhang rule will be performed. Due to the pe-

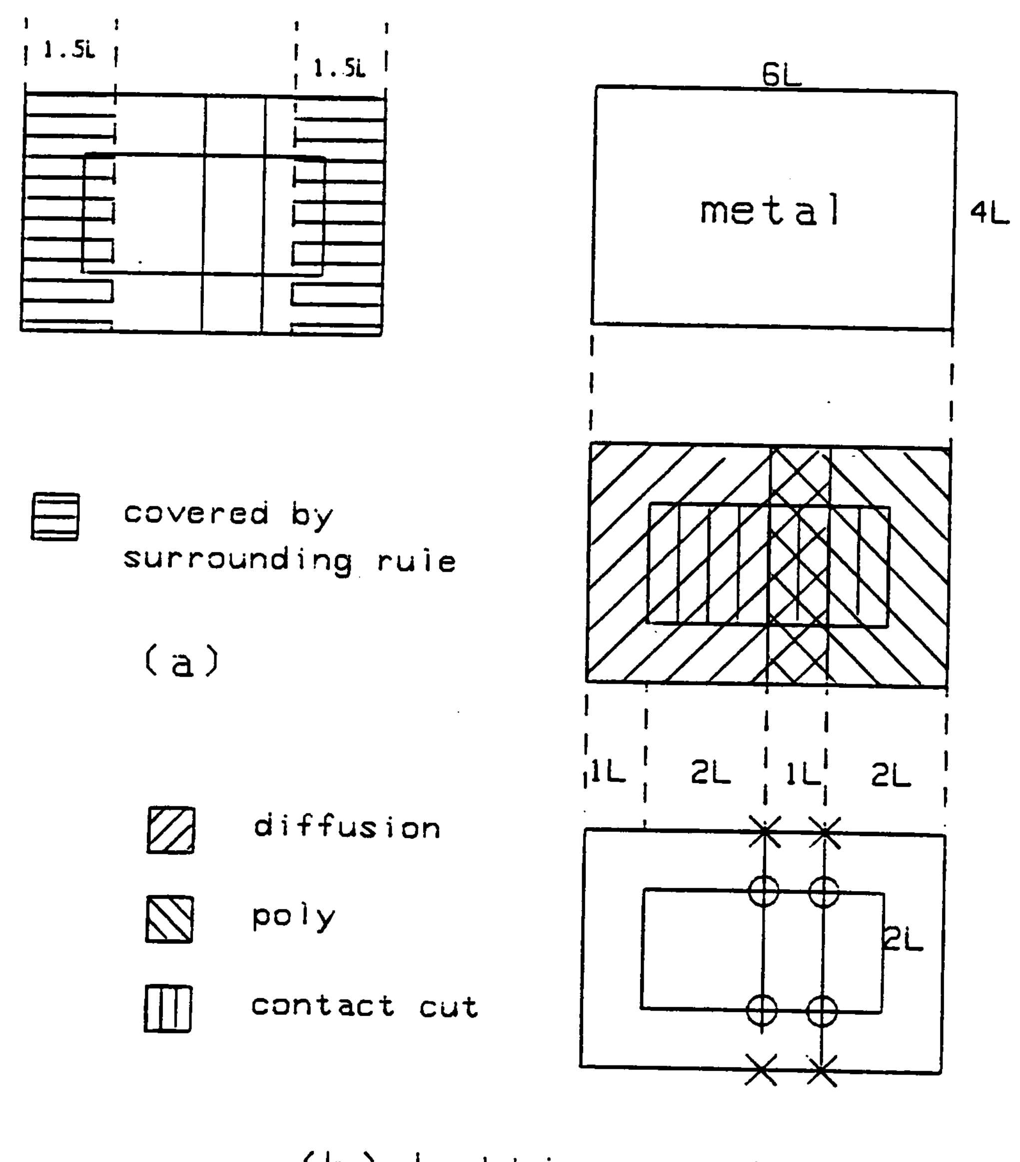


Figure 51. (a) Region which is specified by the contact cut surrounding rule. (b) Butting contact.

culiar configuration of the butting contact, the type of the contact cut context pattern of the corner at which three layer overlaps should be one-side at the corners marked with Xs in Figure 51(b). In other words, if the overlap between poly and diffusion is identified as part of the butting contact and the overlap pattern is not one-side, then there must be violation of the butting contact rule. This is also true for vertically placed butting contacts. If the overlap pattern is one-side, a check for overlap requirement of the butting contact will be carried out at the corners marked with circles in Figure 51. The design rule checking procedure for a corner with poly-diffusion overlap is summarized as Figure 52(a).

If a corner contains poly, diffusion and contact cut like the corners marked with circles, it indicates the butting contact. If the butting contact overlap is made properly, the context pattern of the contact cut should be one-side. Other types of contact cut corners should be reported as the butting contact rule violation. If the contact cut corner has the one-side type context pattern at circled corners, the combination of three context patterns should be one of the four rows in Figure 53 for the proper overlap of the butting contact. Figure 54 shows the constraints for these four context patterns. These constraints are made to cover the area which is not covered by the surrounding constraints.

```
IF poly AND diffusion AND not_cut AND not_buried THEN
   ckeck_butting_contact_or_not;
   IF butting_contact THEN
      check_butting_contact_rule;
   ELSE
      check_transistor_overhang_rule;
   END IF
END IF
```

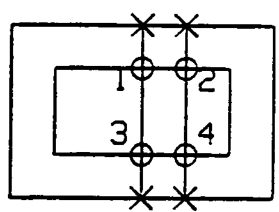
(a). Rule checking procedure for the corner of poly-diffusion overlap.

```
IF poly AND diffusion AND contact_cut THEN
   IF contact_cut_pattern <> one_side THEN
      report_butting_contact_error;
   ELSE
      IF combined_pattern_of_poly_diffusion_cut=
         legal THEN
         check_the_constraint;
      ELSE
         report_butting_contact_rule_error;
      END IF
   END IF
END IF
```

(b). Procedure for the butting contact rule checking.

Figure 52. (a) Checking procedure for the poly-diffusion overlap. (b) Procedure for the butting contact rule checking.

The above butting contact checking procedure for the corner where three layers exist is be summarized as Figure 52(b). For the other three butting contact configurations shown in Figure 55, the similar constraints can be obtained by rotating the constraints of Figure 54.



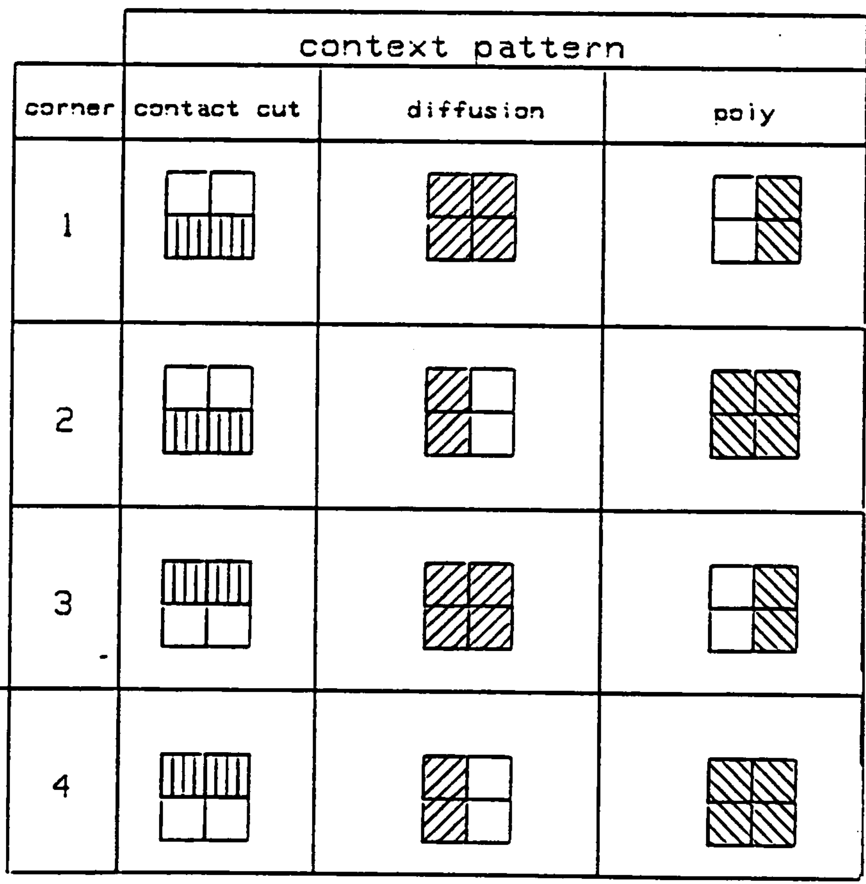


Figure 53. Combined context patterns of cut, diffusion and poly at the corners of contact cut in the butting contact.

length unit : lambda ◠ : corner

	constraint			
corner	contact cut	diffusion	poly	metal
1				
2				
3				
4				

Figure 54. Constraints for the context patterns in Figure 54 for the butting contact.

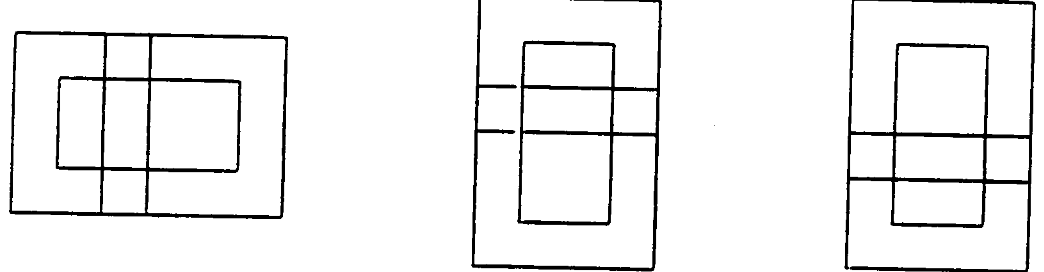


Figure 55. Three configurations of the butting contact obtained by rotation of the configuration in Figure 51.

4.2.8 BURIED CONTACT RULE SPECIFICATION

As described in Chapter 2, there are three types of valid buried contacts as shown in Figure 56. In these three types of buried contacts, the type of context patterns for the buried cut at any corner is one of three types: convex, one-side and all-filled. Even if the buried cut square consists of more than one rectangle, the corner type should be one of the above three types. Therefore, if any other type of context pattern for a buried cut is found, a rule violation is reported without checking any constraints. In spite of this simplification, there are many corners to be checked in one buried contact. If all of these corners are checked, it will take a long time to process all the corners. To solve this problem, one context pattern for buried cuts is selected so that all the requirements of a buried contact can be checked at one corner.

Since all buried contacts are bounded by 6λ X 6λ buried cut square, the upper right corner exists in any buried contact. Based on this, the buried contact rule checking can be represented as Figure 57. Since different configurations can be generated by rotating each configuration in Figure 56, the total number of allowable configurations is 10. Each of these possible configurations must be considered. Among these ten configurations, there are two config-

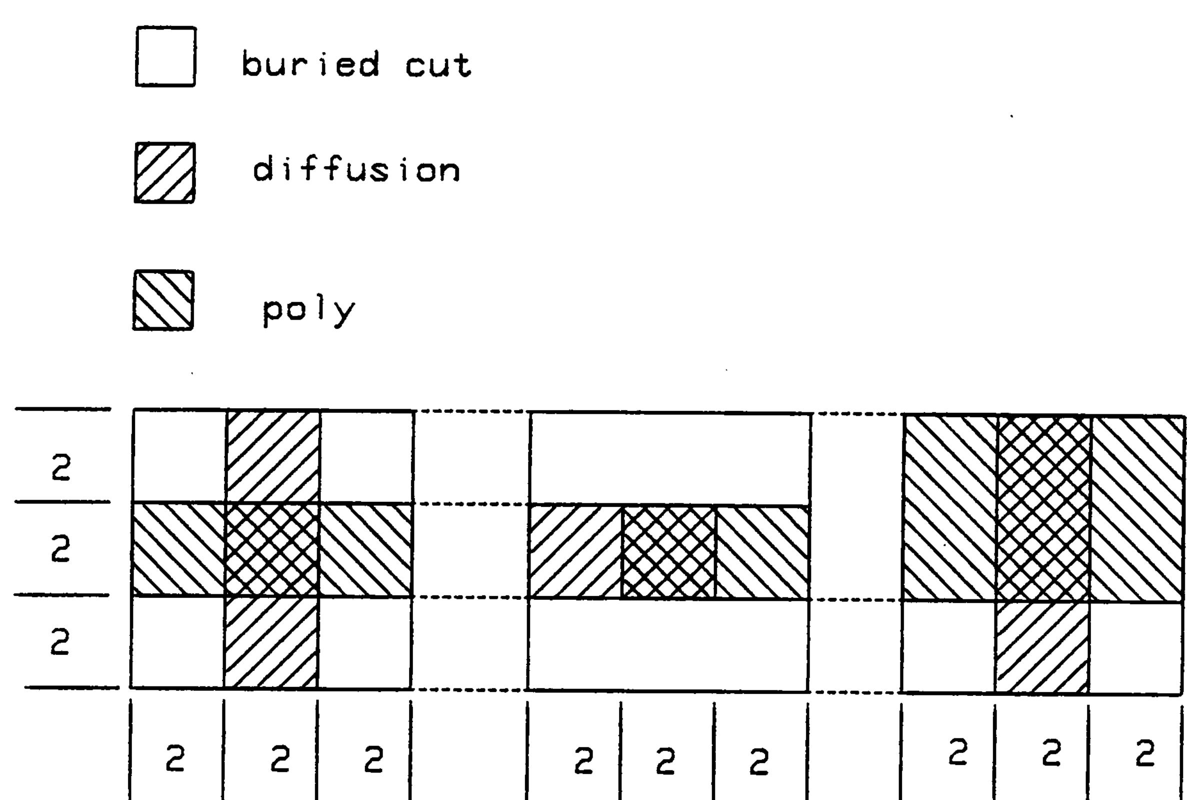


Figure 56. Buried contacts.

```
IF buried_cut THEN
   IF type_of_context_pattern =
      ( convex OR one_side OR all_filled ) THEN
      IF context_pattern represents upper_right_corner
         THEN
         buried_contact_rule_checking;
      END IF
   ELSE
      report_buried_contact_rule_error;
   END IF
END IF
```

Figure 57. Procedure for the buried contact rule checking.

urations that can be identified using the context pattern of the upper right corner. This is because only these two configurations have poly at the upper right corner of the buried cut square as shown in Figure 58. The other eight configurations have no layer except buried cut at this corner. Therefore, if no layer is found at this corner, it indicates one of the other eight configurations. Constraints for the right-corner-checkable configurations are given in Figure 58.

For the other eight configurations, if all eight possible configurations are checked, it will also take much execution time. One way to reduce the number of configurations to be checked is to inspect the other corner. For example, if there exists a poly at the corner 2λ to the left of the upper right corner of the buried cut square, the number of possible configurations is reduced to three as shown in Figure 59. Constraints for this group of configurations are given in Figure 59. Therefore, if the requirement of one of these three configurations is met, then it represents a valid buried contact. If none of these three requirements is met, a rule violation should be reported. If a diffusion layer is found at the position 2λ to the left of the upper right corner, it represents one of three configurations in Figure 60. For the case that there is no layer except buried cut at this position, it should be one of two configurations

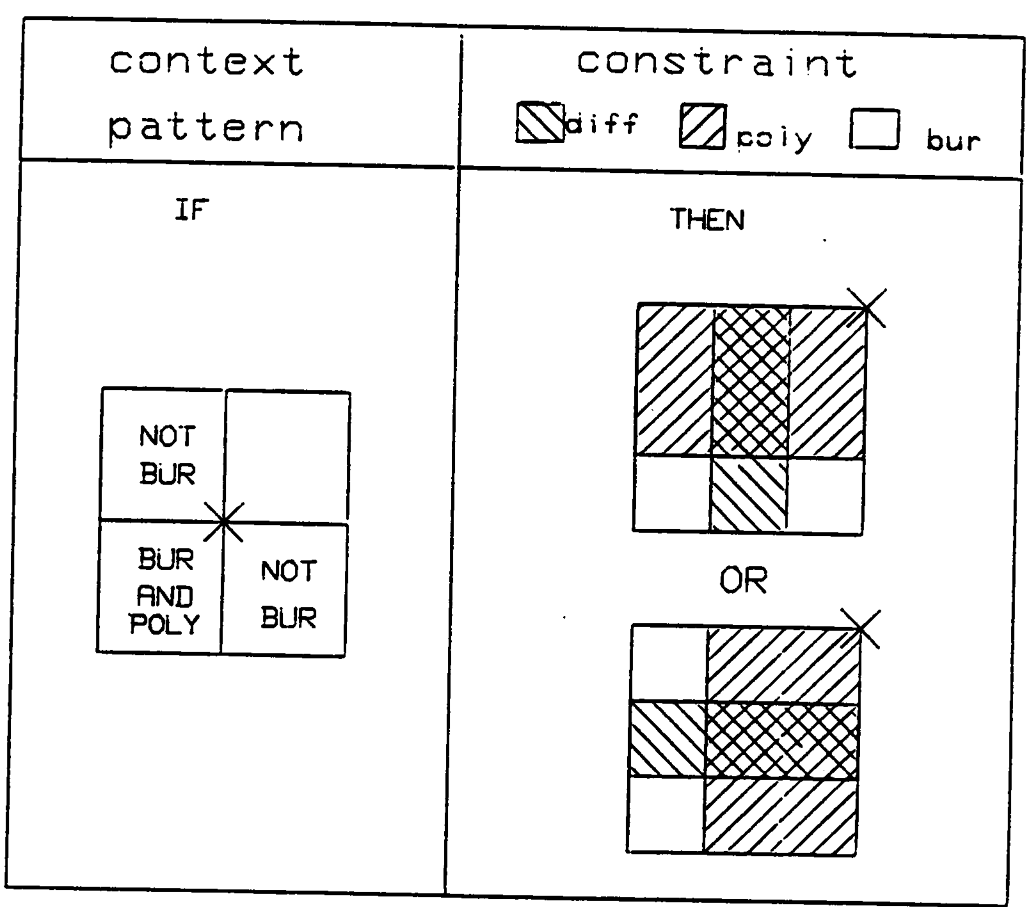


Figure 58. Specification of the buried contact rule for the corner of buried cut with poly.

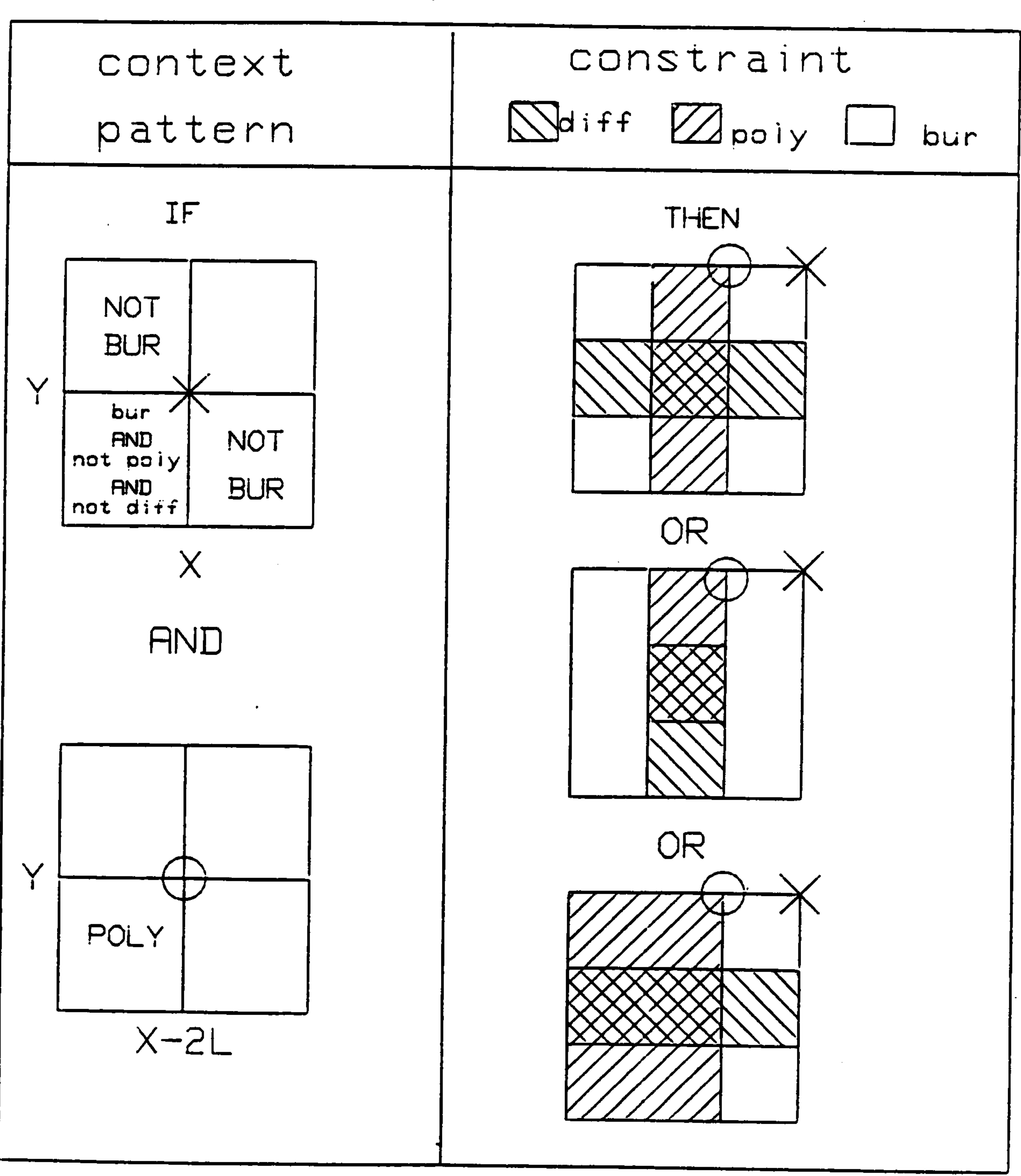


Figure 59. Specification of the buried contact rule for the corners of only buried cut when a corner of poly exists at left.

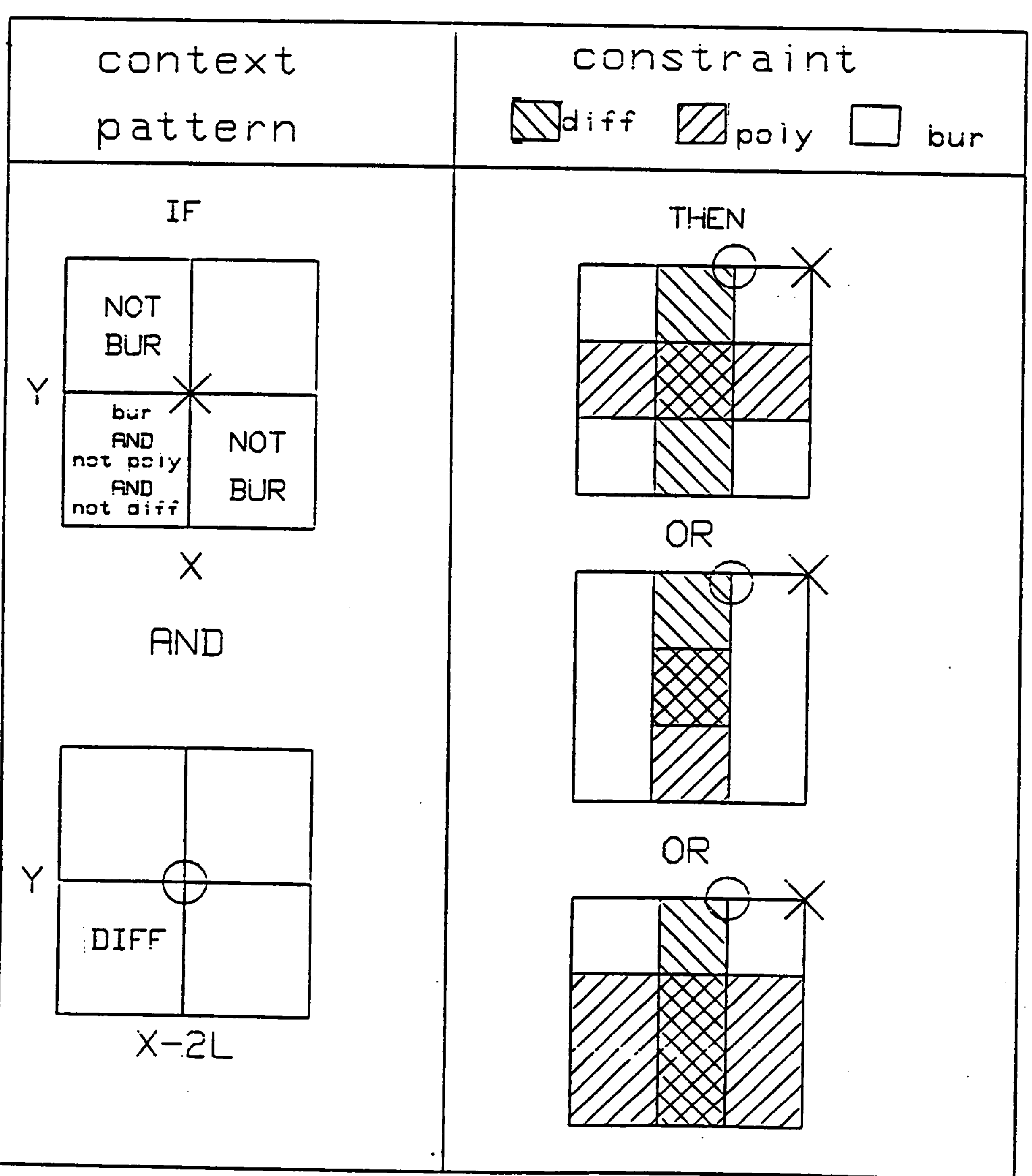


Figure 60. Specification of the buried contact rule for the corners of only buried cut when a corner of diffusion exists at left.

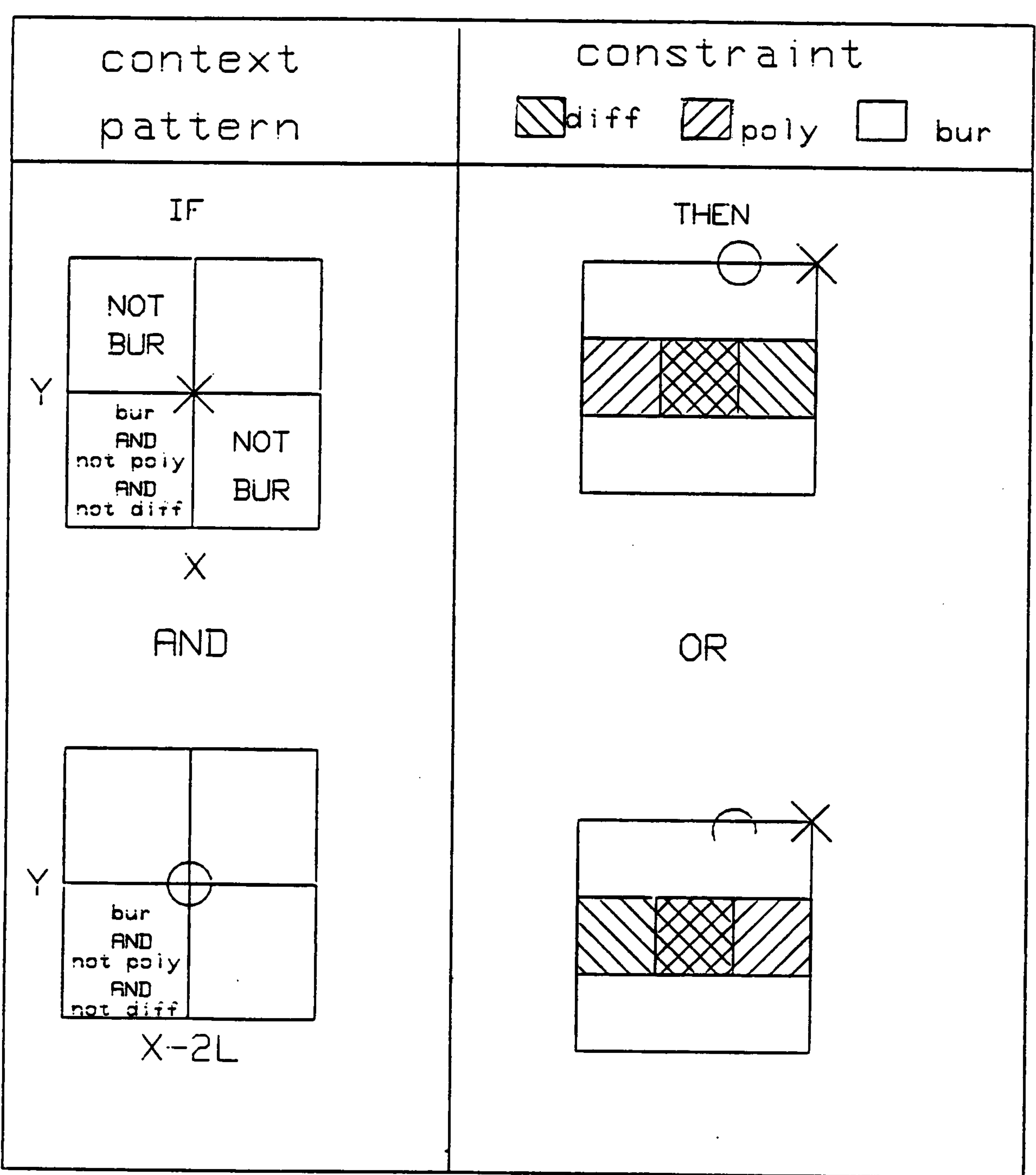


Figure 61. Specification of the buried contact rule for the corners of only buried cut when no layer exist at left except buried cut.

in Figure 61. Constraints for each of these group of configurations are shown in Figure 60 and Figure 61. Using this grouping technique, all ten configurations can be covered.

4.3 CORNER DETECTION AND BIT-MAP GENERATION

To apply the constraints specified in the previous sections, we must know the position of each corner and where a layer exists. Since the constraints are specified locally, only local information for layout is needed. By locally we mean the small area of the design, i.e., the maximum area of the constrained region is $6\lambda \times 6\lambda$ in the buried contact rule. Although there are many ways to represent the layout, a layout bit-map as used in the raster-scan method is appropriate for a representation of the local area of the layout. In this section, we will describe the algorithm for corner detection and layout bit-map generation.

Since it is assumed that the entire layout consists of only rectangles, the only available information for each rectangle is two coordinates (i.e., the lower left corner (xl,yl) and the upper right corner (xu,yu)) as shown in Figure 62. These two coordinates contain two kinds of information as follows:

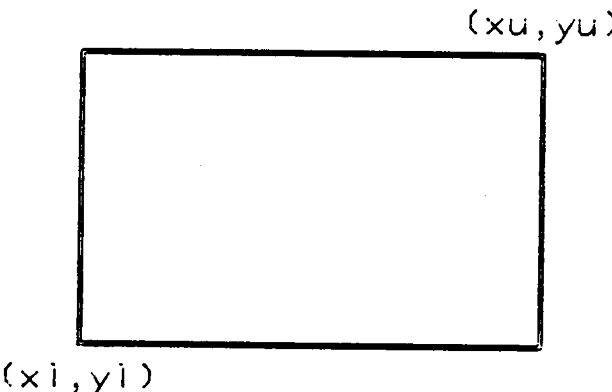


Figure 62. Two coordinates of a rectangle.

1. Coordinates of the four corners of the rectangle, i.e., (xl,yl), (xu,yl), (xu,yl), (xu,yu).

2. Existence of layer in the interval [xl,xu] from yl to yu.

Using this information, corner detection and the bit-map generation, which are static two dimensional problems, can be translated to dynamic one dimensional problems [2].

A static two dimensional problem can be translated to a dynamic one dimensional problem as explained below. Consider a horizontal line below the bottom edge of a rectangle. We will call this line a "scanning line". As the scanning line is moved upward, it will coincide with the bottom edge of the rectangle and then the top edge of the rectangle. Since we know the position (y-coordinate) of the scanning line, we can detect the coordinates of the lower two corners of the rectangle when the scanning line coincides with the bottom edge of the rectangle. To generated the bit-map, a set of intervals ,called ACTIVE_SET, along the x-axis is retained until the scanning line reaches the end of design. When the scanning line hits the bottom edge of a rectangle, the interval [xl,xu] will be inserted to the ACTIVE_SET. Existence of a certain interval in the ACTIVE_SET shows the presence of a certain layer within that interval. Similarly, the coordinates of the top two corners of the rectangle can be detected

when the scanning line coincides with the top edge of the rectangle. When the top edge is hit by the scanning line, the interval [xl,xu] will be deleted from the ACTIVE_SET because the layer no longer exists within the present scanning line context. To find intersections between rectangles, each time an interval is inserted to and deleted from the ACTIVE_SET, the intersecting points are searched between all the intervals in the set and the new interval. This is because intersections of two rectangles exist only at the bottom or top edge of one of the two rectangles.

The procedure given above is summarized in the following algorithm. Assume that the minimum x and y coordinates of the layout are (0,0).

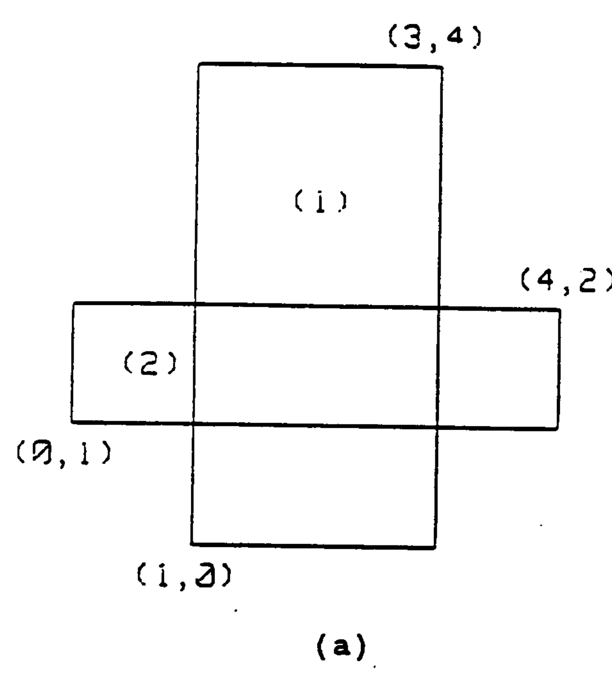
```
1.  Sort the rectangles in order of the y-coordinates of the
    bottom edge and also in order of y-coordinates of the top
    edge. During the sorting, find the maximum x-coordinate
    "xmax" and maximum y-coordinate "ymax" of the entire
    layout;
2.  FOR i:= 0 to ymax-1 DO
        FOR j:= 0 to xmax-1 DO
            BIT_MAP[i,j] :=0; { initialize the BIT_MAP }
3.  ACTIVE_SET := empty_set;
4.  CORNER_SET := empty_set;
5.  FOR scanning_bar := 0 TO ymax DO  BEGIN
```

```
a.  FOR each rectangle with yl = scanning_bar DO  BEGIN
    1)  insert (xl,yl) and (xu,yl) to the CORNER_SET;
    2)  find the intersection between the interval
        [xl,xu] and all the intervals in the active set
        and if it exists, insert the intersection to the
        CORNER_SET;
    3)  insert the interval [xl,xu] into ACTIVE_SET;
    4)  END;
b.  FOR each rectangle with yu = scanning_bar DO BEGIN
    1)  insert (xl,yu) and (xu,yu) to the CORNER_SET;
    2)  find the intersection between the interval
        [xl,xu] and all the intervals in the ACTIVE_SET
        and if it exists, insert the intersection to the
        CORNER_SET;
    3)  delete the interval [xl,xu] from the ACTIVE_SET;
    4)  END;
c.  FOR each interval in the ACTIVE_SET DO
    1)  FOR i:= xl to xu-1 DO BEGIN
              BIT_MAP[scanning_bar,i] := 1;
d.  END;
```

To see how this algorithm works, we will apply the above algorithm to the layout which consists of two rectangles shown in Figure 63(a) and will trace the algorithm step by step. Suppose that coordinates of the two rectangles are stored in the array as in Figure 63(b). Numbers which cor-

respond to the steps of the algorithm are shown as --() at the end of each operation.

1. First two sort arrays will be produced according to the order of y-coordinates of the top edge and bottom edge of the rectangle as shown in Figure 63(c). xmax = 4, ymax = 4 -- (1)
2. BIT_MAP[i,j] = 0 (i=0..3, j=0..3) -- (2)
3. ACTIVE_SET = { } -- (3)
4. CORNER_SET = { } -- (4)
5. scanning_bar = 0 -- (5)
    - box[low_sort[1]].yl = scanning_bar, do FOR loop -- (5.a)
        - CORNER_SET = {(1,0),(3,0)} -- (5.a.1)
        - NO intersection -- (5.a.2)
        - ACTIVE_SET = {[1,3]} -- (5.a.3)
    - box[low_sort[2]].yl ≠ scanning_bar, skip FOR loop -- (5.a)
    - box[up_sort[1]].yu ≠ scanning_bar, skip FOR loop -- (5.b)
    - BIT_MAP[0,1] = 1, BIT_MAP[0,2] =1 -- (5.c)
6. scanning_bar = 1 -- (5)
    - box[low_sort[2]].yl = scanning_bar, do FOR loop -- (5.a)
        - CORNER_SET={(1,0), (3,0), (0,1), (4,1)} -- (5.a.1)



(a)

array of rectangle : BOX

index	xl	yl	xu	yu
1	1	Ø	3	4
2	Ø	1	4	2

(b)

LOW_SORT_ARRAY

index	box key
1	1
2	2

UP_SORT_ARRAY

index	box key
1	2
2	1

(c)

Figure 63. (a) Two rectangles in the layout. (b) Database for two rectangles. (c) Two sort arrays in order of bottom and top edge of two rectangles.

- find intersecting point between [0,4] and [1,3] ->CORNER_SET={(1,0), (3,0), (0,1), (4,1), (1,1), (3,1)} -- (5.a.2)
- ACTIVE_SET = {[1,3],[0,4]} -- (5.a.3)

• box[up_sort[1]].yu ≠ 1 ->skip FOR loop -- (5.b)

• BIT_MAP[1,i] = 1 (i=1..2), BIT_MAP[1,i] = 1 (i=0..3) -- (5.c)

7. scanning_bar = 2 -- (5)

• box[up_sort[1]].yu = 1 -> do FOR loop -- (5.b)
  - CORNER_SET={(1,0), (3,0), (0,1), (4,1), (1,1), (3,1), (0,2), (4,2)} -- (5.b.1)
  - find intersecting point between [0,3] and [1,3] ->CORNER_SET={(1,0), (3,0), (0,1), (4,1), (1,1), (3,1), (0,2), (4,2), (1,2), (3,2)} -- (5.b.2)
  - ACTIVE_SET = {[1,3]} -- (5.b.c)

• BIT_MAP[2,i] = 1 (i=1..2) -- (5.c)

8. scanning_bar = 3 -- (5)

• box[up_sort[2]]].yl ≠ 3 -> skip FOR loop -- (5.b)

• BIT_MAP[3,i] = 1 (i=1..2) -- (5.c)

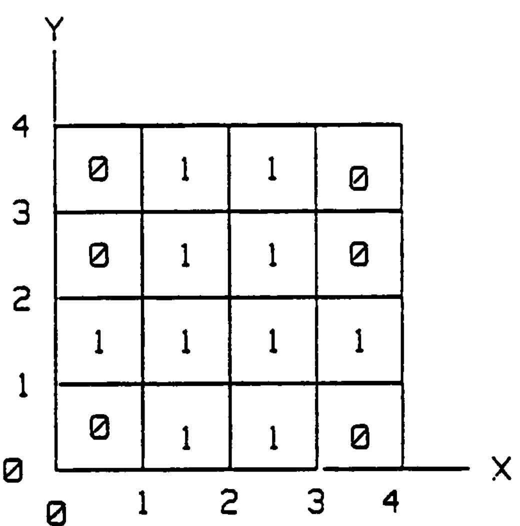
9. scanning_bar = 4 -- (5)

• box[up_sort[2]].yl = 4 -> do FOR loop -- (5.b)
  - CORNER_SET = {(1,0), (3,0), (0,1), (4,1), (1,1), (3,1), (0,2), (4,2), (1,2), (3,2), (1,4), (3,4)} (5.b.1)
  - NO intersections -- (5.b.2)
  - ACTIVE_SET = { } -- (5.b.3)

10. scanning_bar=5 > ymax -> END.

As shown in Figure 64, the final BIT_MAP represents the original layout and the final CORNER_SET contains all the corners of the two rectangles.

In fact, if we consider the complexity of a VLSI circuit layout, it requires too much memory space to save all the corners and the bit-map of an entire layout as in the above example. Since local areas of the layout are used in order to identify context patterns and check the constraint, we do not need to save the entire bit-map. Actual implementation of this algorithm will be treated in the next chapter.



CORNER_SET = ((1,0),(3,0),0,1),
(4,1),(1,1),(3,1),
(0,2),(4,2),(1,2),3,2)
(1,4),(3,4))

Figure 64. Final bit-map and detected corners.

4.4 A LIMITATION OF THE CORNER-BASED DRC ALGORITHM

The Corner-based algorithm has a limitation in its ability to perform spacing rule checks. Since the rule checking depends only on local layout information, the connectivity of layout cannot be considered. Due to the lack of this connectivity information, false errors can be reported during the spacing rule checking. For example, consider three metal rectangles connected together as shown in Figure 65. The corner marked with an X is one of the convex corners. When the corresponding constraint is applied, a spacing rule violation will be reported. However, the two rectangles A and B are connected by the third rectangle C thus, the spacing rule violation has no meaning and it should not be reported. This kind of false error can occur in all kinds of spacing rule checks. To solve this problem, information on the circuit connectivity as produced by a circuit extractor is necessary. Circuit extraction is beyond the scope of this thesis and as such the implemented DRC system will suffer this minimal shortcoming.

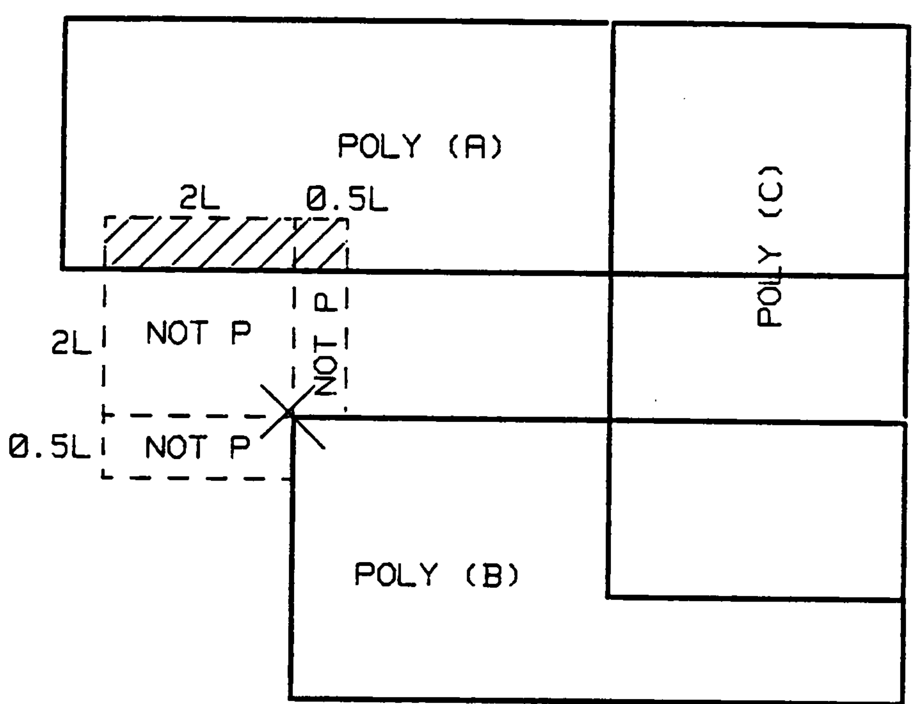


Figure 65. A false violation detected in the spacing rule checks.

5.0 IMPLEMENTATION OF CORNER-BASED DRC ALGORITHM

5.1 DESIGN RULE CHECKER

5.1.1 CORNER DETECTION AND BIT-MAP GENERATION

As described in the previous chapter, each time the scanning bar hits the bottom or top edge of a rectangle the corners on the scanning bar are detected. Obviously, these corners have the same y-coordinates. To check the design rules for these corners, the bit-map for the local area of layout is necessary since the constraints are specified for a local area. To determine the size of window in which the bit map should be generated, the following two things should be considered. First, if the size of the design is $n\lambda$ X $m\lambda$, the width of the window should be $n\lambda$ allowing a corner can exist at any position on the scanning bar. Second, in the NMOS design rule set, the maximum distance requirement is 6λ (in the buried contact rule). Thus, 6λ is sufficient to check possible the constraints. Therefore, to check the constraints of all corners on the scanning bar we need a bit-map window of $n\lambda$ X 6λ. Moreover, since the corners are detected while the scanning bar moves upward, the window also should move whenever the scanning bar moves. In other words, the scanning window with a size of $n\lambda$ X 6λ will scan the entire

layout from bottom to top and the bit-map within this scanning window should be available all the time.

Since this DRC software is written for ICICLE package , 0.5λ, which is the basic unit of ICICLE, is used as the unit of the grid of bit-map. Based on this grid unit, the window size is determined as 2n X 13 in actual implementation. To implement the scanning window, 13 arrays in which each array has 2n bit elements are used as in Figure 66. Each bit in the array is set to 1 or 0 according to the existence of a layer in that position. Since there are six layers in NMOS layout, six scanning windows are necessary to represent all layers.

Line 13 in the scanning window plays the role of scanning bar. As a result, whenever line 13 hits the top or bottom edge of any rectangle, corner detection and bit-map generation will be performed by the algorithm in the previous chapter. The generated bit-map is stored in the 13th array. When the scanning window moves upward by 1 unit, the ith array is copied into the (i-1)th array. In this manner, the bit-map within the scope of scanning window is available all the time.

When a corner is detected by the scanning-bar, it is located on the scanning bar, i.e., line 13 of the scanning

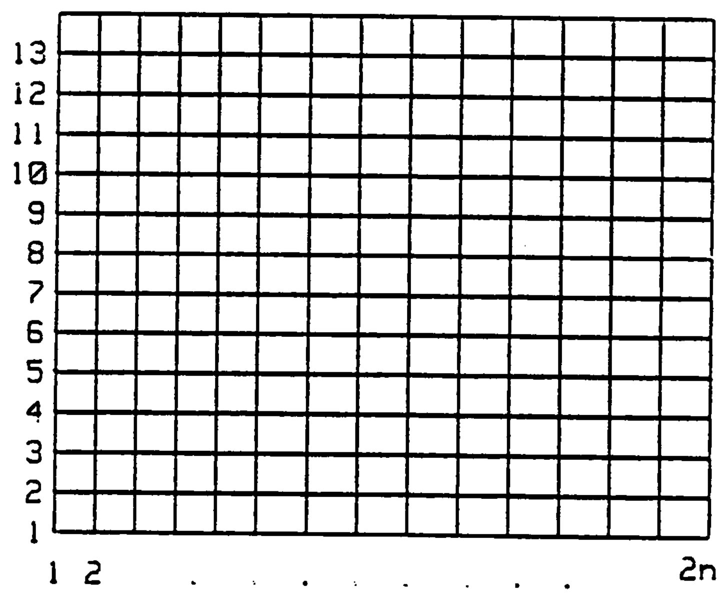


Figure 66. Grid on the scanning window.

window. However, in order to check the several constraints at once, the corner should be located on line 8 of the scanning window. The reason is as follows.

Suppose that one convex type metal corner is detected as shown in Figure 67. This type of metal corner has 3λ minimum width rule constraint as well as the 3λ spacing rule constraint. In order to be able to check these two constraints simultaneously, the bit-map of region 3λ above and 3λ below from the corner should be available. Figure 67 shows that this is possible when the corner is on line 8 of the scanning window. Therefore, the corner which is detected by line 13 should be stored until the corner is located on line 8 by the moving scanning window. For this reason, three stacks are assigned to each line from line 13 to line 8 because there are three kinds of corners, i.e., bottom-edge, top-edge and intersection corner, according to where they exist in a rectangle. In the bottom-edge and top-edge corner stacks, each element in the stack is a key of rectangle. An element of intersection corner stack is the exact x-coordinate of the intersection corner. The stacks are also copied into the stacks of next line when the scanning window moves upward. Therefore, the constraint checking will be performed on the corners only in the stacks on line 8.

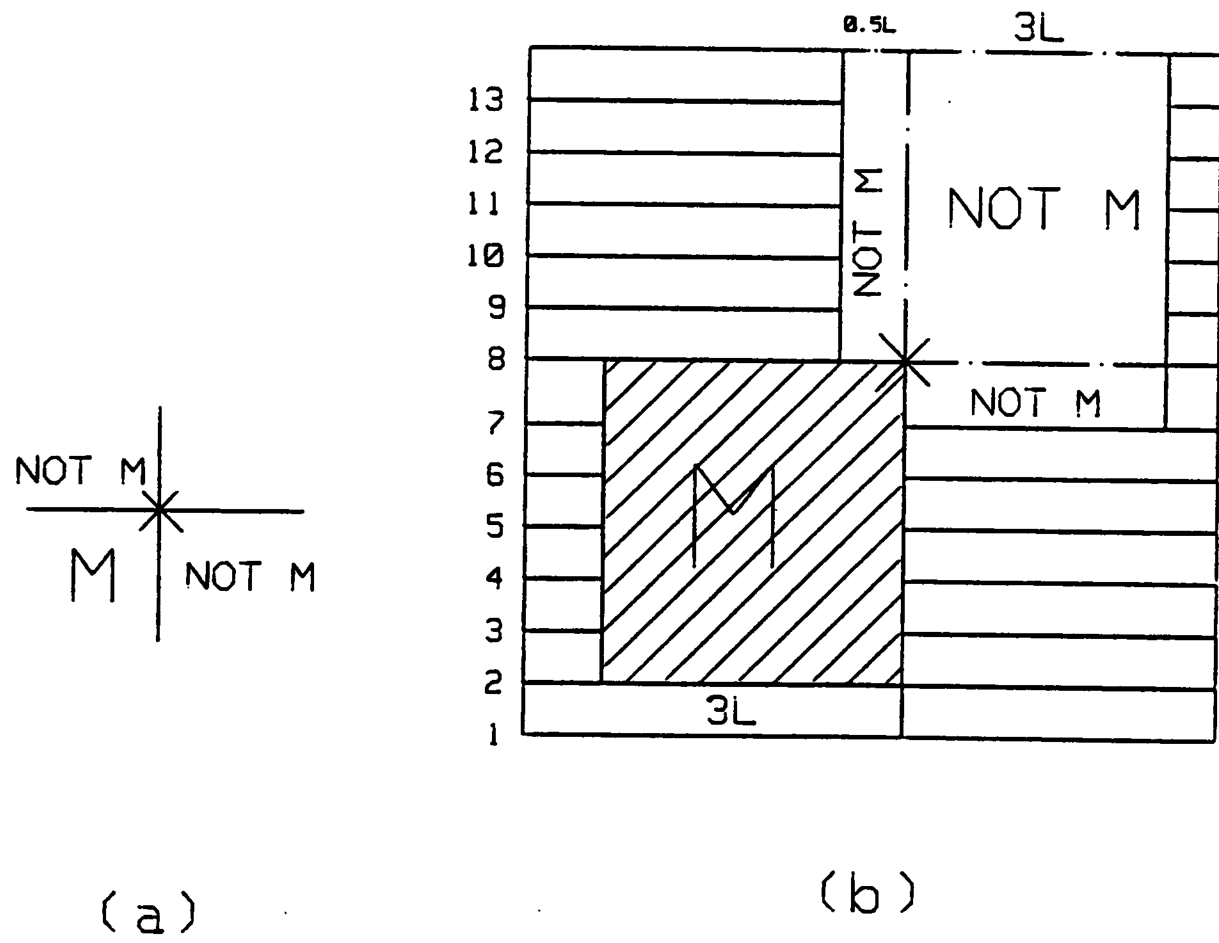


Figure 67. (a) A concave corner of the metal layer. (b) Constraints for the width and spacing rules in the scanning window.

There is one exception to this scanning scheme which is invoked for buried-contact rule checking. As explained in the previous chapter, buried contact rule checking is applied for only one particular context pattern, i.e., the context pattern which represents the upper right corner of the buried cut square. If this type of buried cut corner is detected on line 13, the bit-map of the whole buried contact is already available in the scanning window since the edge which is coincides by the scanning bar is the top edge of the buried square. Consequently, the constraint for the buried cut corner can be applied at line 13. If the buried cut corner is checked at line 8 as in the corners of other layers, we need a larger sized bit-map, i.e., more memory space is necessary.

All of the above actions can be summarized algorithmically as in Figure 68. This pseudo code shows the overall program flow of the design rule checker. Note that the design rule checking for the entire layout is accomplished by repeated operation of corner detection, bit-map generation and corner processing while the scanning window is scanning the entire design. In the next subsection we will discuss the implementation of the corner processing.

```
TYPE
   rectangle = RECORD
                  layer,xl,yl,xu,yu : integer ;
               END;
VAR
   box : array[no_of_rectranlges] of rectanle;


BEGIN
   initialize_bit_map;
   corner_stack_initialization;
   top_sort_stack := sort( top_edge,box );
   bottom_sort_stack := sort( bottom_edge,box );

   FOR scanning_bar := ymin TO ymax + 5 DO BEGIN
      top_corner_stack_13 := empty;
      bottom_corner_stack_13 := empty;
      intersection_corner_stack_13 := empty;

      WHILE box[ TOP(bottom_sort_stack) ].yl = scanning_bar DO
      BEGIN
         box_key := TOP(bottom_sort_stack);
         PUSH(box_key, bottom_corner_stack_13);
         POP (bottom_sort_stack);
         END;

      WHILE box[ TOP(top_sort_stack) ].yu = scanning_bar DO
      BEGIN
         box_key := TOP(top_sort_stack);
         PUSH(box_key, top_corner_stack_13);
         POP(top_sort_stack);
         END;

      delete_from_ACTIVE_SET( top_corner_stack_13 );
      insert_to_ACTIVE_SET( bottom_corner_stack_13 );
      find_intersections( intersection_corner_stack_13 );
      bit_map_generation( ACTIVE_SET,map_array_13 );

      corner_processing_for_buried_contact_rule_checking
      (corners_at_line_13);
      corner_processing_for_rule_checking( corners_at_line_8 );

      FOR i := 2 to 13 DO
         bit_map_array_i-1 := bit_map_array_i;

      FOR i := 9 to 13 DO BEGIN
         bottom_corner_stack_i-1 := bottom_corner_stack_i;
         top_corner_stack_i-1 := top_corner_stack_i;
         interseciton_corner_stack_i-1:= intersection_corner_stack_i;
         END;
      END;
END.
```

Figure 68. Overall flow of the design rule checker using the corner-based algorithm.

5.1.2 CORNER PROCESSING

In the corner processing procedure, actual design rule checking is performed by the identification of context pattern and constraints rule checking. The first step is the identification of the context pattern. Since there are four quadrants, i.e., four pixels, around a corner, all 15 possible context patterns can be represented as 4-bit binary numbers. Because each pixel is already represented by one bit depending on the existence of a layer, the 4-bit binary number can be extracted directly from the bit-map. For convenience of coding, this 4-bit binary number is converted to the decimal number as shown in Figure 69(a). This number will be called "pattern number" in the remainder of this chapter. Figure 69(b) shows the pattern numbers of all 16 context patterns. In the above manner, six pattern numbers for six layers will be found at each corner.

After the calculation of the pattern numbers, corresponding constraints will be checked. As described in Chapter 4, all the constraints are specified for a certain bounded region and the requirement is the presence or absence of certain layer(s) on that bounded region. These constraints can also be represented as number codes. Since each pixel of the bit-map represents the existence of a layer and the size of the pixel (0.5λ X 0.5λ) is known, the area of a

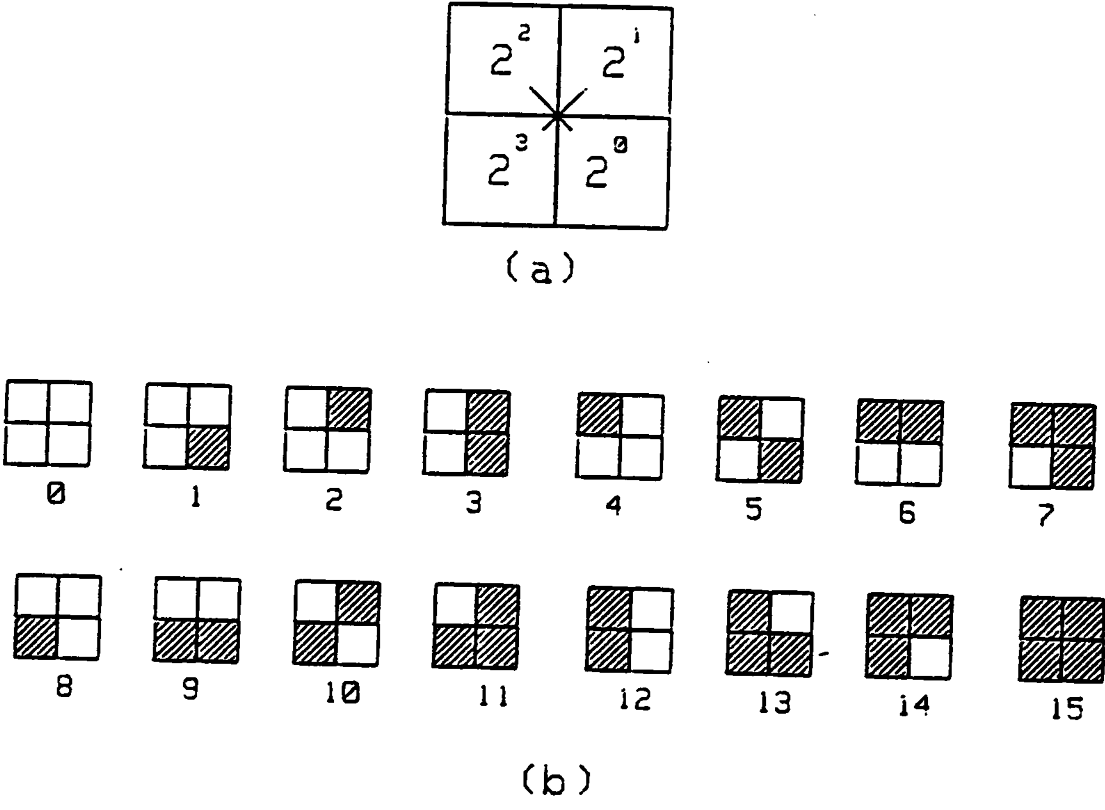


Figure 69. (a) Four numbers assigned to the four quadrants around a corner. (b) Sixteen pattern numbers for 16 context patterns.

bounded region can be represented as a number of pixels, e.g., the area of 2λ X 2λ square is 16 pixels. Therefore, constraints can be checked by counting the number of 1's or the numbers of 0's within the bounded region. To specify a bounded rectangle in the bit-map, we need the width and length of the bounded rectangle, the position of the corner, the direction of expansion, and the layer type. All the parameters except the direction of expansion can be determined directly by the constraint. The direction of expansion should be determined according to the context pattern. For the specification of the direction of expansion, one number is assigned to each corner of a rectangle as shown in Figure 70. A routine called "COUNT_PIXELS", shown in Figure 70, is written for counting the number of pixels containing 1 within a rectangle whose boundary is specified by the four parameters mentioned above.

The following example shows how a constraint is specified and checked using COUNT_PIXEL. Suppose that there are two poly boxes which violate the spacing rule and we want to check the spacing rule and width rule at the corner X as shown in Figure 72. Since the "pattern number" of the corner X is 8 (refer to Figure 69), the spacing and width rule checking can be represented as follows.

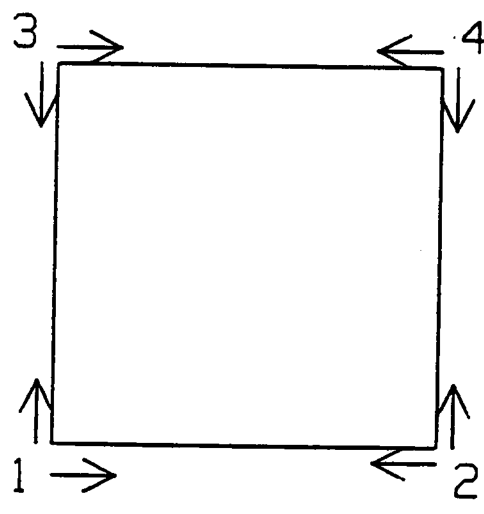


Figure 70. Numbers representing the expansion direction used in the routine COUNT_PIXEL.

```
 FUNCTION count_pixel (layer: bit_map;
                        corner_x,corner_y,width,length,
                        expansion_direction: integer)
                        : integer;
BEGIN

   CASE expansion_direction OF

        1: lower_bound:= corner_y;
           left_bound:= corner_x;

        2: lower_bound:= corner_y;
           left_bound:= corner_x-width;

        3: lower_bound:= corner_y-length;
           left_bound:= corner_x;

        4: lower_bound:= corner_y-length;
           left_bound:= corner_x-width;
   END; { CASE }

   count_pixel:= 0;
   FOR y:= lower_bound TO lower_bound+length-1 DO
      FOR x:= left_bound TO left_bound+width-1 DO
         count_pixel:= count_pixel + layer[x,y];

END;
```

Figure 71. Function COUNT_PIXEL.

```
IF poly_pattern_no = 8 THEN
   BEGIN
      space_area := COUNT_PIXEL(poly,x,y,4,4,1);-- (1)
                   +COUNT_PIXEL(poly,x,y,4,1,2);-- (2)
                   +COUNT_PIXEL(poly,x,y,1,4,2);-- (3)
      IF space_area ≠ 0 THEN ----------------------(4)
         report_error(space_po_po,x,y);------------(5)

      width_area := COUNT_PIXEL(poly,x,y,4,4,4);---(6)
      IF width_area ≠ 16 THEN ---------------------(7)
         report_error(width_po,x,y);---------------(8)
   END;
```

In the above code, line (1), (2), (3) specify the region R1, R2, R3, respectively using the parameters in COUNT_PIXEL. Since there is no poly layer in R2 and R3, the value of COUNT_PIXEL is 0. However, there is one pixel inside the R1 (shaded pixel in Figure 72), thus, the final value of the "space_area" will be 1. Therefore, the spacing rule violation will be reported as expected. Since the number of poly pixels in the region specified in line (7) is 16, the width rule constraint is satisfied. In this manner, all the constraints can be specified as number of pixels.

In the actual implementation, the poly pattern number, poly bit-map and required spacing distance are passed to the

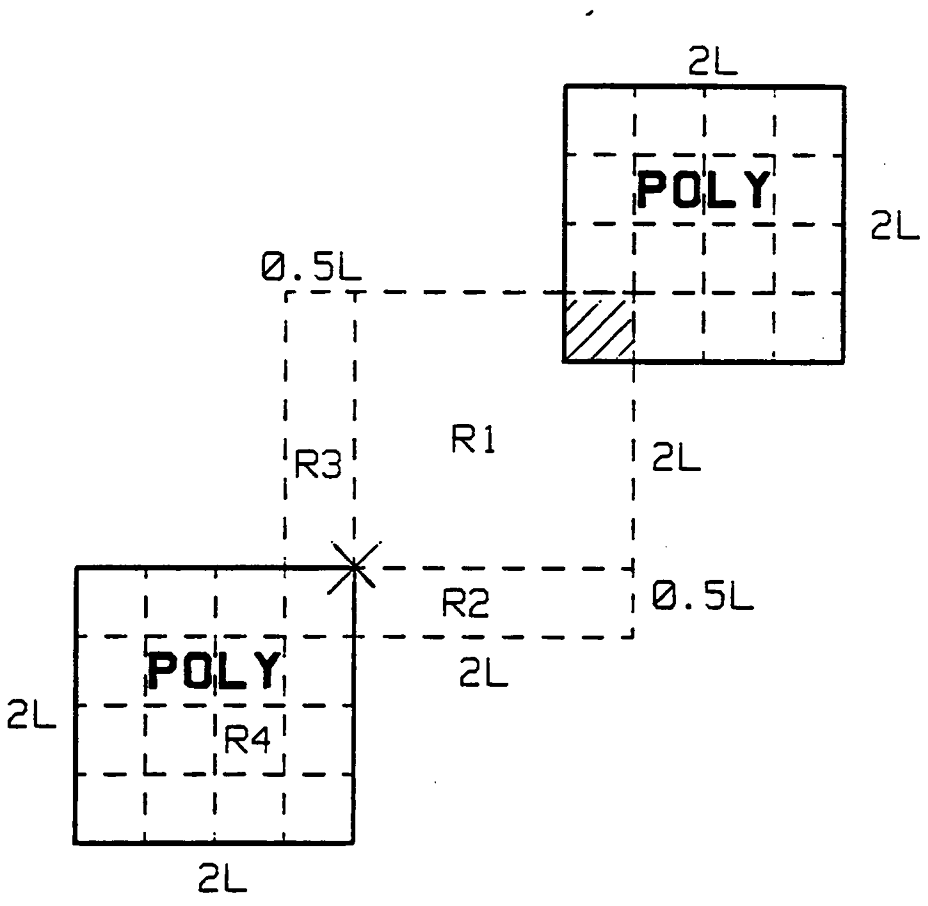


Figure 72. Spacing and width rule checks at a convex corner of poly using the routine COUNT_PIXEL.

spacing rule checking routine so that this routine can be used for spacing rule checking for other layers.

5.2 INTERFACING DRC WITH ICICLE

5.2.1 DATABASE CREATION FOR HIERARCHICAL DESIGN RULE CHECKING

In creating a database on which design rule checking will be performed, the structure of the design should be taken into account. ICICLE, to which this DRC is interfaced, generates a layout which has a hierarchical structure. In general, a tree structure layout is generated. A primitive element of this hierarchical structure is a symbol definition, usually called a "cell". A symbol definition is composed of a group of rectangles and/or reference to other symbol definitions.

Due to the inherent regularity of VLSI layouts, one symbol definition generally is used many times. Therefore, if design rule checking is performed hierarchically, i.e., each repeated symbol definition by a call is checked only once, much execution time can be saved compared to the "flattening" method. Here "flattening" means the expanding all the calls to the lower level symbol definitions in the hierarchy.

In hierarchical design rule checking, interaction between symbol definitions is a major problem because design rule can be violated between definitions even if each indi-

vidual definition has no inherent design rule violation. In most of VLSI layouts, connections between symbol definitions are made only at the boundary regions of the symbol definitions. To perform a check on the interaction between definitions, we need information on rectangles which reside at only boundary of the lower level symbol definitions. This, of course, is provided that rule checking for lower level definitions have already been performed. On the other hand, if there is a rectangle which penetrates a lower level definition, we need information on all the rectangles which are involved in the connection with the penetrating box from different symbol definitions.

There can be a case where flattened design rule checking is suitable in spite of the hierarchical structure of the layout. For example, the flattening method is very useful in checking design rules for a PLA, since the PLA consists of a number of small cells and there are many rectangles penetrating. In this case, a flattened DRC will be much more efficient.

To make it possible to check the design rules for all three cases, one module called "drc_data_base" is added to ICICLE. This module creates a database according to the option chosen by the user. The options are "Flattening", "Unflattening" and "User-defined-window". If a designer chooses

the "Flattening" option for a leaf cell in a tree hierarchy or in a definition like a PLA, a database will be created containing all the rectangles which belong to that symbol definition and all the expanded calls of the symbol definitions.

For the case where interaction exists only in the boundary of the definitions, the "Unflattening" option should be used. If this option is chosen, a database is created by following manner. First, all the rectangles of the definition which is called by the user will be put into the database. After that, rectangles ,which reside in the boundary regions, of the calls which are called by directly from the current editing definition are put into the database. Here the boundary region is defined as a symbol definition bounding box shrunk by 4λ. In other words, the bounding box of each call is shrunk by 4λ and all the rectangles which reside outside the shrunk bounding box are put into the database. Here 4λ is used based on the contact cut and maximum spacing requirement. For this reason, it is recommended that buried or butting contact not be used in the connection between definitions.

When the rectangles penetrate the lower level definition, the "User-defined window" should be used. To specify an area to be checked, the designer defines a window with the

two points, (xmin,ymin) and (xmax,ymax). This window is expanded by 3λ in order to check all possible spacing rule errors. After the expansion, all the rectangles inside the expanded window, i.e., the region which is bounded by (xmin-6,ymin-6) and (xmax+6,ymax+6), will be put into the database. Obviously, this window should contain all the rectangles which are involved in the connection around the penetrating box.

Using three options given above, hierarchical design rule checking can be performed in the following manner. First, check the design rules of the leaf cell using the "Flattening" option. If there is a design rule violation, it should be corrected before checking the other definitions. After leaf cells are checked, design rule can be checked for upper level definitions using the "Unflattening" option or "User-defined window" option depending on the existence of the penetrating rectangles.

Once the database is created, it will be passed to the design rule checking routine. Design rule checking is performed on this database using the sequence explained in the previous section.

5.2.2 REPORTING OF DESIGN RULE VIOLATIONS

Typically, design rule checking is performed in batch mode. In this situation, it is very difficult for the designer to correct the design rule errors as the design evolves. Therefore, the design rule checking should be performed interactively during the design of a layout and the operation of DRC should be compatible with the IC layout editing system. Basically, there are two requirements for an interactive design rule checker. First, it should be easy to invoke the design rule checker inside the layout editor. Second, the information about the design rule error should be provided in such a way that the designer can correct any design rule errors easily.

The information necessary for the designer to easily correct a design rule error includes information on the position of the corner containing the error and the type of design rule error which has occurred. When ICICLE is being used, the name of the symbol definition which has the invalid corner is useful to the designer. This is because it is not allowed to edit the called definition during editing the calling definition. To provide these kinds of information about a design rule error, a linked list, shown in Figure 73, is constructed during corner processing. If a corner containing a design rule error is found, all the in-

formation about the corner will be passed to the error reporting routine. These passed parameters will be stored in a linked list structure shown in Figure 73. If there is no design rule error, the head of the list called "drc_error_list" will be left as NULL. After the processing of all of corners is finished, the pointer "drc_error_list" is passed to the error displaying routine. This error displaying routine displays a marker at invalid corners on the screen. Moreover, if the designer identifies a marked corner by positioning the cursor at that corner, this routine will display the definition name and types of design rule errors found. If there is no design rule error, an appropriate message will be displayed on the screen. In the above manner, the designer can get all the information about design rule violations and correct the violations very easily.

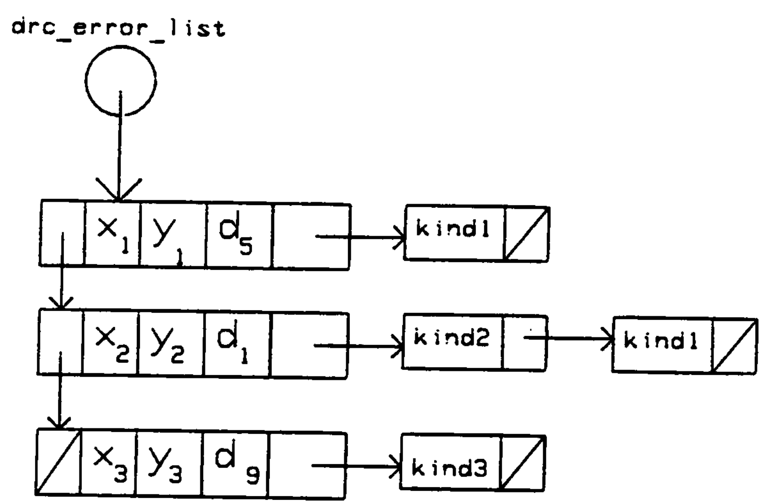


Figure 73. A linked list for design rule error reporting.

6.0 RESULT

The DRC program is written in the C programming language. It contains about 3,000 lines of code and comments. This software has been tested under VAX-11/785 using the students' designs which were developed in a VLSI design course at VPI & SU. As expected, the false violations are detected in the spacing rule checks when particular configurations of interconnected rectangles are used. These false violations , when recognized, should not cause the user too much difficulty.

In order to allow the DRC to be used interactively with ICICLE, a few commands are added to ICICLE [19]. Most users are very happy with the interactive operations. Moreover, it was found that the name of the symbol definition provided in the error reporting is very useful to the user when correcting design rule error. The implementation of the three options for creating database to be checked was also very successful. With these three options, very efficient design rule checking could be performed. Especially, the "User-defined window" option makes hierarchical design rule checking possible for the design containing the penetrating box.

7.0 CONCLUSION

We have presented an implementation of an interactive design rule checker for IC layout. The DRC uses what is known as the corner-based algorithm for rule checking.

The corner-based algorithm was selected due to the ease with the specification of various design rules, locality property and its relatively fast speed compared to the existing algorithms. One limitation of this algorithm is the detection of the false violation in the spacing rule checks. However, this limitation is tolerable if we consider the expensive computation for the connectivity checking as used in polygon algebra algorithm.

In interfacing the design rule checker with the layout editor, ICICLE, interactive operations were designed so that the DRC can be invoked inside the layout editing mode. For ease of correction of the detected design rule violations, all of the information about the violation, such as position, type of violation, and symbol definition name are provided in an interactive manner.

In creating the database used in checking, three options are implemented in order to allow for effective hierarchical

design rule checking. Due to the locality property of the checking scheme, the corner-based algorithm is appropriate for incremental design rule checking. However, an incremental DRC could not be easily implemented because of the structure of the layout database in ICICLE. To construct the incremental design rule checker, the database should be constructed based on the position of rectangles. Moreover, since all the rectangles of the layout are stored in a linear list in ICICLE, it takes a lot of time to access the rectangle in the end of the list.

Concerning the problems mentioned above, we feel that the data structure of the layout database should be determined based on the characteristics the CAD tools to be used. This means that consideration must be given to CAD tools which will be used in conjunction with the layout editor and DRC such as a circuit extractor, router etc. The software designer must be aware of the overall CAD tool scheme in which the layout database will be used.

8.0 REFERENCES

1. Arnold, M.H. and Ousterhout, J.K., "Lyra: A New Approach to Geometrical Layout Rule Checking," Proc. 19th Design Automation Conf., 1982, pp. 530-536.

2. Wilcox, P., Rombeck, M. and Caughey, D.M., "Design Rule Verification Based on One Dimensional Scans," Proc. 15th Design Automation Conf., 1978, pp. 285-289.

3. Mead, C.A. and Conway, L.A., Introduction to VLSI systems, Reading: Addison-Wesley, 1980.

4. Privite communication with MOSIS, 1984.

5. Ullman, J.D., Computational Aspects of VLSI, Computer Science Press, 1983.

6. Lyon, R.F., "Simplified Design Rules for VLSI Layouts," VLSI Design, First Quarter 1981, pp. 54-59.

7. Baker, C.M. and Terman, C., "Tools for Verifying Integrated Circuits Designs," Lambda, Fourth Quarter 1980, pp. 22-30.

8. Whitney, T., "A Hierarchical Design-Rule Checking Algorithm," VLSI Design, First Quarter 1981.

9. Baird, H.S., "Fast Algorithms for LSI Artwork Analysis," Proc. 14th Design Automation Conf., 1977, pp. 303-311.

10. McCaw, C.R., "Unified Shapes Checker-A Checking Tools for VLSI,", Proc. 16th Design Automation Conf., 1979, pp. 81-87.

11. Losleben, P. and Tompson, K., "Topological Analysis for VLSI Circuits," Proc. 16th Design Automation Conf., 1979, pp. 461-473.

12. Lindsay, B.W. and Preas, B.T., "Design Checking and Analysis of IC Mask Designs," Proc. 13th Design Automation Conf., 1976, pp. 301-308.

13. Rosenberg, L.M. and Benbassat, C., "CRITIC: An Integrated Circuit Design Rule Checking Program," Proc. 11th Design Automaiton Workshop, 1974, pp. 14-18.

14. Hoglund, I. and Berg, H., "LSI Circuit Design," Computer Aided Design, 1976, Vol. 18, No. 3., pp. 165-174.

15. Yoshida, K., Mitshuhash, T., Nakada, Y. Chiba, N., Ogita, K. and Nakatsuka, S., "A Layout Checking System for Large Scale Integrated Circuits," Proc. 14th Design Automation Conf., 1977, pp. 322-330.

16. Taylor, G.S. and Ousterhout, J.K., "MAGIC's Incremental Design Rule Checker," Proc. 21st Design Automation Conf., 1984, pp 160-165.

17. Iacaponi, M.J, "ICICLE : An interactive Color Integrated Circuit Layout Editor," VPI & SU Dept. of Electrical Eng. internal document, 1983.

18. Iacaponi, M.J., "ICICLE User's manual," VPI & SU Dept. of Electrical Eng. internal document, 1983.

19. Kim, K., "ICICLE-DRC User's Manual," VPI & SU Dept. of Electrical Engineering internal document, 1985.

20. Aho, A.V., Hopcroft, J.E. and Ullman, J.D., The Design and Analysis of Computer Algorithms, Reading: Addison-Wesley, 1974.

The vita has been removed from the scanned document