VLSI Design
1994, Vol. 1, No. 2, pp. 127–154
Reprints available directly from the publisher
Photocopying permitted by license only

Geometric Design Rule Check of VLSI Layouts in Mesh Connected Processors

S.K. NANDY
Laboratory for Computer-Aided Design, Indian Institute of Science, Bangalore, INDIA

R.B. PANWAR
University of Illinois at Urbana-Champaign, Urbana, Illinois, USA

(Received May 11, 1989, Revised February 25, 1990)

Design Rule Checking is a compute-intensive VLSI CAD tool. In this paper we propose a parallel algorithm to perform Design Rule Check (DRC) of Layout geometries in a VLSI layout. The algorithm assumes the parallel architecture to be a two-dimensional mesh of processors. The algorithm is based on a linear quadtree representation of the layout. Through a complexity analysis it is shown that it is possible to achieve a linear speedup in DRC with respect to the number of processors.

Key Words: *Layout Verification; VLSI CAD tools; Parallel Algorithms; Analysis tools; Design Automation*

INTRODUCTION

In a typical CAD environment for LSI/VLSI design, the entire design process involves the use of various CAD tools, such as circuit simulator, logic simulator, timing simulator, layout editor, design rule checker, circuit extractor, floor planner, routers etc. One of the tools which is computer intensive is the Design Rule Checker (DRC). In order to speed up DRC, various hardware and software solutions have been proposed [1, 2, 3, 4].

In [5] a linear time design rule checker was proposed which is based on a quadtree representation of mask layouts. A quadtree corresponding to a layout can be obtained by successively dividing the layout into four layout blocks corresponding to the quadrants North-west (NW), North-east (NE), South-west (SW) and South-east (SE) until a square block is obtained that is totally covered by a layout geometry (LG) or a part of it. Corresponding to each quadrant obtained during the subdivisioning process, we associated a node in the quadtree. The four quadrants constitute the four sons of the root node which represents the entire region covered by the layout. Note that the subdivision process is not continued beyond a certain threshold size that is set by technological constraints (lambda). A node in the quadtree is either a terminal node or a non-terminal node. A terminal node is Black if it denotes a region that is completely covered either by an LG or a part of an LG. A terminal node is white if it represents a region that is not covered by an LG or a part of an LG. A non-terminal node is a Gray node.

In spite of the many advantages associated with a painted quadtree representation of a layout (comprising several hundred thousand transistors), the main disadvantage would be that of storage. This is because for every node in the quadtree, we must store pointers to its parent and four siblings. Instead, if the quadtree was implemented as an array, we would have to allocate space for a full tree amounting to $(4^{n+1} - 1)/3$ nodes, for a tree of depth n. However, we would not need to store pointers to siblings and parent since their location is implicit in the array index of any given node. Further, in VLSI design, all regions are rectangular and evenly distributed over (most of) the chips bounding box. As a result, for highly dense layouts, it is presumable that the quadtree representing such a layout would grow close to its full size. Consequently, we find no storage efficiency in storing the tree in a pointer based structure as opposed to an array based structure.

In fact we would find in typical layouts that a pointer based structure would actually consume more memory space due to the pointer storage overhead corresponding to each of the four sons and parent. Thus an array based representation gives us a five fold saving in storage space per node. In addition an array based storage will permit many algorithmic refinements by facilitating direct hashing to nodes in the quadtree. Henceforth, we shall refer to an array based quadtree as a linear quadtree. Linear quadtrees are also suitable for archival storage of VLSI layouts, since only those array elements corresponding to black nodes (representing painted regions) need be stored. Thus, in the worst case 4^n nodes are to be stored for a tree of depth n. Since the total layout area covered by such a tree is $2^n \times 2^n$, the storage requirement for a linear quadtree is proportional to the area. A linear quadtree as a data structuring technique to represent mask layouts, and algorithms for neighbor finding and following the boundaries of regions (represented by a linear quadtree) is reported in [6]. For the sake of completeness, we reproduce the salient features of such a representation in the following section.

Since the underlying algorithm in [5] to follow the boundaries of regions in the layout is inherently sequential and is based on a pointer based quadtree representation of mask layout which is not storage efficient, in this paper we propose parallel algorithms for DRC based on linear quadtree representation of mask layouts.

The rest of the paper is organised as follows. In the following section we briefly describe the linear quadtree representation of a layout and explain how the various design rule checks are carried out by following the boundaries of LGs. We then present algorithms to perform DRC in parallel. We develop a scheme to partition the layout into various sub-layouts such that each sub-layout can be assigned a separate processor in the mesh and processed independently. The technical issues of combining the partial results of DRC at each processors (allotted a sub-layout) are described in detail. Lastly, we present a complexity analysis of the algorithm, and concluding comments.

LINEAR QUADTREE BASED DESIGN RULE CHECKER

To describe linear quadtree encoding of a layout, we define the following terms. A node in a quadtree is a number called K_value as shown below.

K_value = $k_n\, k_{n-1} \ldots k_1$ where $0 <= k_i < 5$

Each digit k_i denotes the path to be taken at level i. (0 = terminal 1 = NE, 2 = NW, 3 = SE, 4 = SW) to reach the node from root which is assigned a level 0. The root carries a K_value 00 . . . 0. This K_value gives the unique path from root to any node in the tree and can also be obtained for a given path. K_value has the following properties.

1. if $k_i <> 0$ then $k_j <> 0$ for all $j <= i$
2. if $k_i = 0$ then $k_j = 0$ for all $j >= i$
3. number of non-zero entries in K_value = level of the node
4. root node has all digits in K_value = 0
5. the most significant non-zero entry in K_value specifies the sontype of the node. A number 1 meaning the node is NW son of its father, 2 for NE, 3 for SW and 4 for SE.
6. a K_value yields to a unique decimal number given by

   Dec_value = $4^{n-1}*k_n + 4^{n-2}*k_{n-1} + \ldots 4*k_2 + k_1$
7. this Dec_value may be used to index the linear array representation with dimension [0 . . ($4^n - 1$)*4/3] for a quadtree of depth n. Each element of this array contains 2 bits (call c_0, c_1) of information. These represent the colortype of the node as follows.

```
type
  colortype = (white,gray,not_used,black);
```

c_0	c_1	color
0	0	White
0	1	Gray
1	0	not_used
1	1	Black

A white or black node is a terminal node and has no children. However, with regard to the array representation of quadtrees, the nodes allocated for their children are labeled "not_used". A gray node is a non-terminal node and has children which are either black, white or gray. For the rest of the paper, we adopt the definitions and algorithms given for neighbor finding and boundary following of regions as given in [6]. A node in a linear quadtree has the following representation.

```
node = packed record
         color:colortype   (2 bits c0 and c1)
       end;
```

The array corresponding to a layer in the layout (comprising many connected regions which are sim-

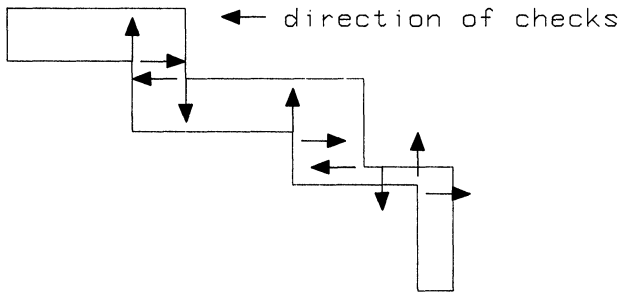


FIGURE 1 Width Checks of Mask Openings.

ilar in size and uniformly distributed) of size $2^n * 2^n$, will have the entries for all nodes in the tree. A n-level fully grown tree will have $(4^{n+1} - 1)/3$ nodes, requiring the following representation.

layer_in_layout = array[0 . . $((4^n - 1)/3) - 1$] of node;

(Total space required by the array for a $2^n * 2^n$ layout (of a single layer) is $(2/3)*(4^{n+1} - 1)$ bits or approximately $(4^n/3)$ bytes).

The various design checks that are carried out on LGs in a layout are given in Appendix A. These checks can be easily carried out by following the boundaries of LGs in different layers of the layout, and they can be classified as orthogonal checks and diagonal checks to meet the rectangular and square constraints.

Width check for polygons (LGs) in a particular layer is performed as follows. For every LG, the boundary following algorithm (BFA) traces its boundary and computes the length of each of its sides. Width violation occurs if the length of any side in an LG is not within the minimum width specifications. Checking for gaps between LGs in a layer is carried out by traversing through white leaf nodes in two orthogonal directions away from the corners of an LG (encountered during the traversal of its boundary). The orthogonal distance traversed before a black node is encountered is verified for spacing requirements.

Width checks of mask openings are carried out whenever the edges in LGs of the mask change directions (see Figure 1). This check is achieved in a manner similar to spacing checks. We traverse through the black nodes moving in the two orthogonal directions inwards into the LG, and accumulate the width of the mask opening until we encounter a white node. An error is flagged if the width is less than the minimum width. This check, however, cannot verify the extent of the mask openings between two corners of an LG(s). (See Figure 2). In order to do this, we need to perform a width check along the

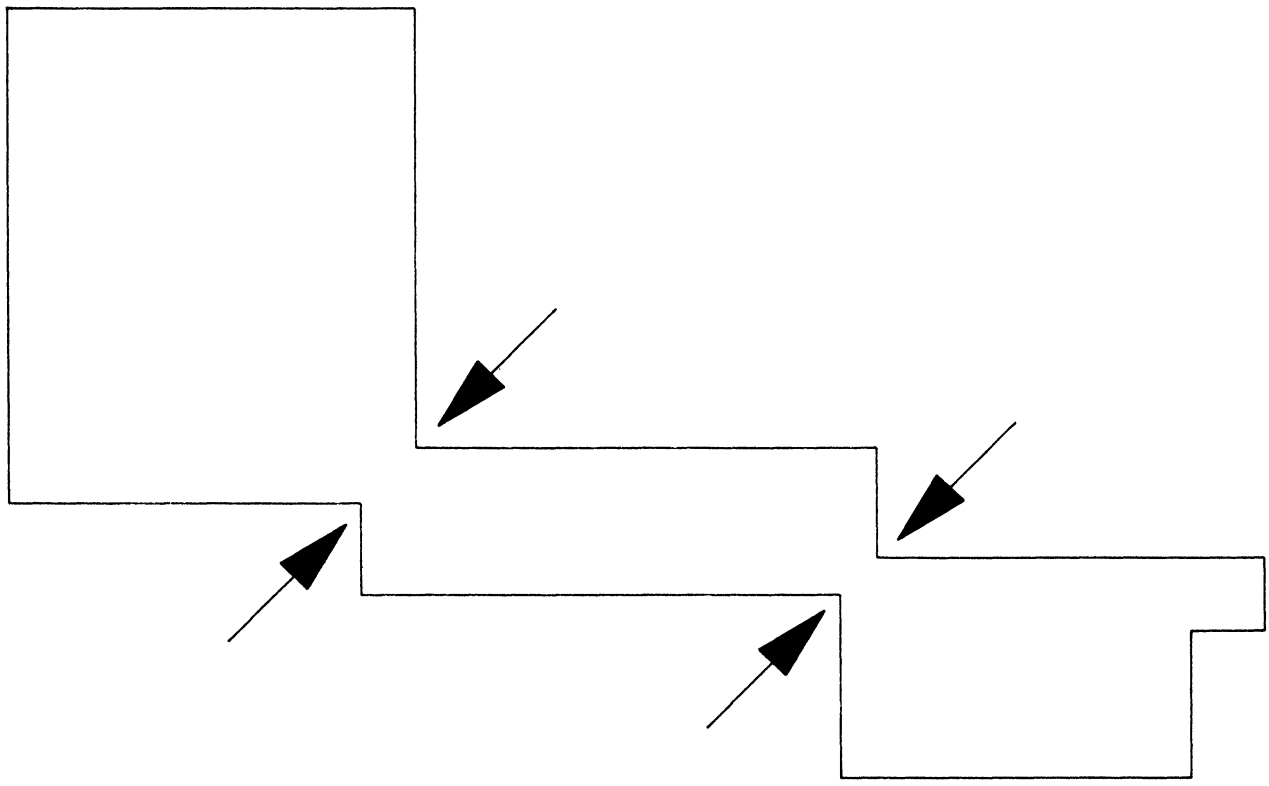


FIGURE 2 Width of Mask Opening between two Corners.

diagonal of the rectangular region defined by the two corners in question i.e. satisfy a rectangular constraint.

Thus any diagonal check can be achieved by travelling an orthogonal distance $d1$ along a particular direction followed by a distance $d2$ in a direction that is normal to previous direction in a clockwise fashion to meet the rectangular constraint. This is followed by a traversal of orthogonal distances $d2$ along a particular direction followed by a distance $d1$ in a direction that is normal to the previous direction in an anti-clockwise fashion to meet the rectangular constraint. At this instance, a diagonal width error is flagged if the rectangular constraint is violated. Similarly, diagonal spacing checks must be carried out at the corner of the polygon(s) (See Figure 3).

Since extension and overlap checks are primarily carried out for transistors, we obtain the quadtrees corresponding to the two layers viz. polysilicon and diffusion (for notational convenience we refer to the first tree as that of poly and the second as that of diffusion). In order to perform overlap check, we traverse through the leaf (terminal) nodes in poly until we encounter a black node. We then index into the corresponding node in diffusion and check whether it is black. If not the process of traversal of poly and comparison with diffusion continues until a black node is found in both poly and diffusion (this indicates the beginning of an overlapped region). As illustrated in Figure 4 this node is the upper left corner of the overlapped region. The BFA is invoked and we traverse black nodes of poly (along the east direction) ensuring that the corresponding nodes in diffusion are also black. The end of an edge corresponding to an overlap region is said to be found when we encounter a white node either in poly or in diffusion. The length of this edge is checked for minimum overlap requirements. If the white node encountered was that of poly, then the BFA continues to find the other overlapped edges in the clockwise direction. If not, then the BFA continues on poly and the edge being traversed now corresponds to that of extension. This edge is verified for the corresponding extension width requirements. Following this the overlap check is resumed at the last node from where extension check was initiated.

Similarly context-sensitive checks can be carried out by performing appropriate width and spacing checks on LGs belonging to two or more layers.

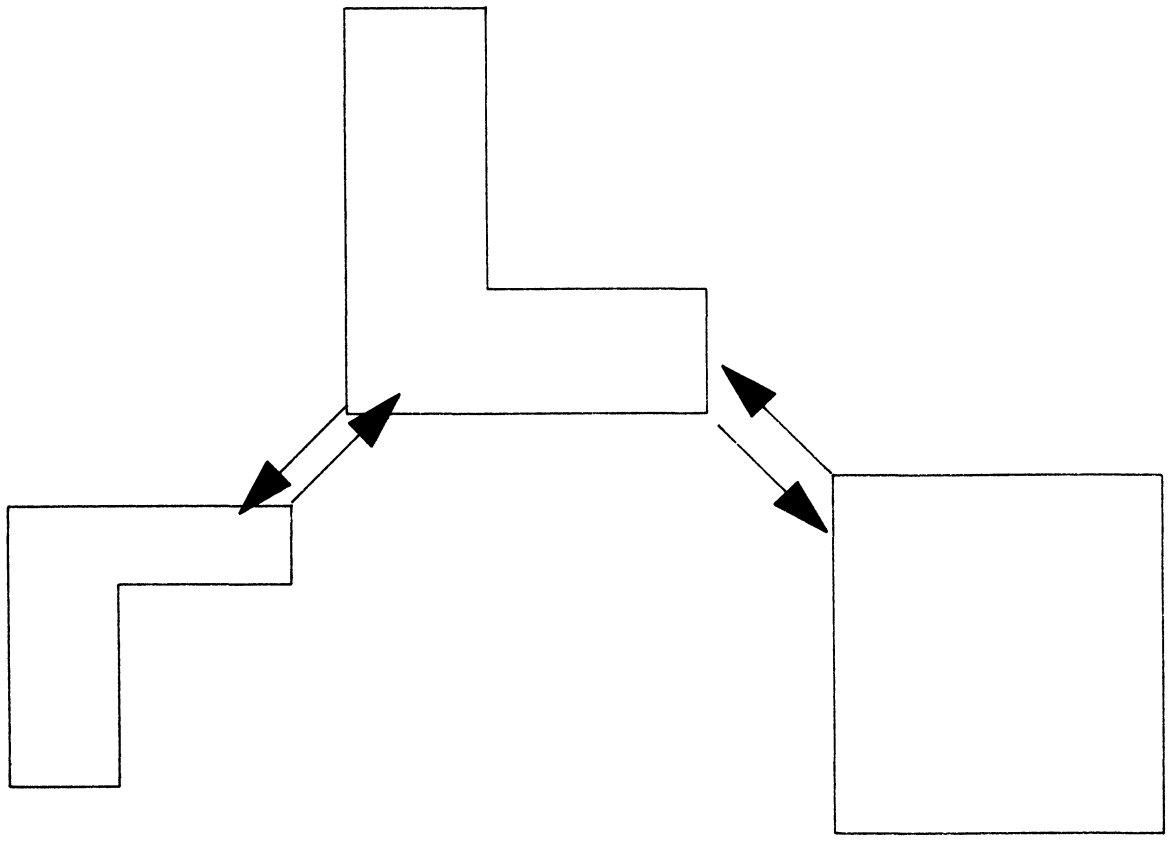
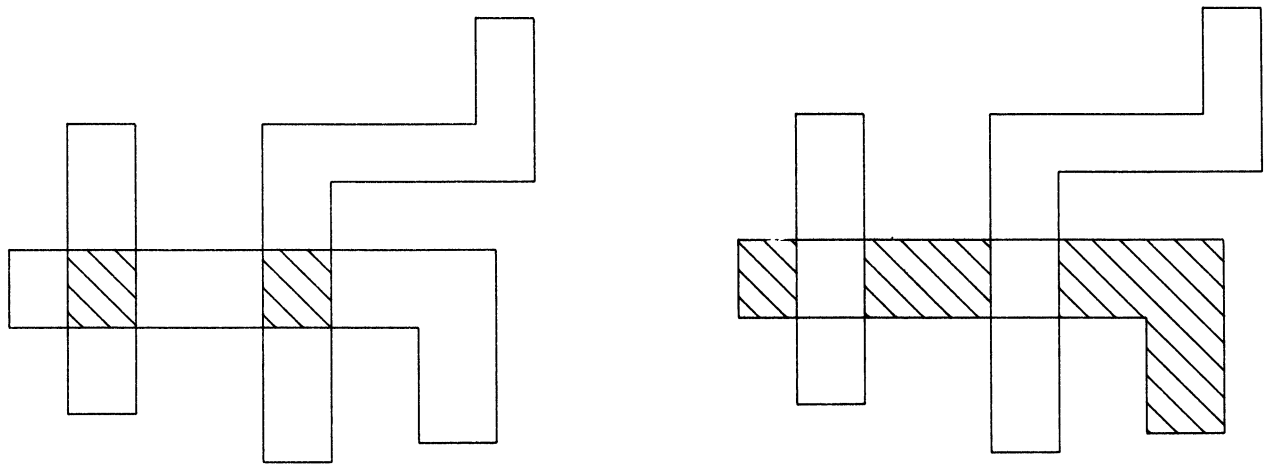


FIGURE 3 Diagonal Spacing Check.



(a) overlap region hatched (b) extension region hatched

FIGURE 4 Overlap and Extension Checks.

DRC IN A MESH CONNECTED PROCESSOR ARCHITECTURE

The overall layout is partitioned into sub-layouts, each of which is given to a processor in the mesh. All the processors form appropriate quadtrees of their sub-layouts and perform DRC on them in parallel. For those LGs that are totally enclosed within a processor boundary, DRC can be easily performed in parallel without any communication between the processors. Whereas for an LG which spans two or more processor boundaries, DRC can be performed only by a sequential traversal of its boundary. Hence there is no advantage in processing such LGs using multiple processors. In fact use of multiple processors for such LGs involves communication and synchronization overheads. The method we adopt to perform DRC of LGs in parallel is similar to that of parallel boundary following of regions in an image [7]. Hence we split such LGs into multiple geometries each of which is fully enclosed within different processor boundaries. After we have split every LG as mentioned above, DRC can be performed on the various clipped LGs in parallel. This DRC does not check the design rules fully for the clipped-LGs. Hence we have to perform certain extra checks for all the clipped LGs that fall into one of the following two categories: (1) An LG is divided at the processor boundary into clipped-LGs which are allotted to two or more processors, and none of the processors have the actual dimensions of the edges of the original LG. (2) An LG is within a design rule interaction distance (DRID[3]) of the processor boundary such that while performing the design rule checks processor boundary is encountered thereby precluding any further traversal to complete the check. The processing of such clipped LGs is carried out in two phases, one that performs checks along orthogonal directions and the other that performs diagonal checks. The algorithm for parallel-DRC is given below in Pascal like pseudo-code.

```
Algorithm Parallel-DRC;
begin
  Partition Layout;
  for all processors do in parallel
  begin
  { CDC_List → Clipped DRC completion list and
    UDC_List → Unclipped DRC completion list
      CI_List → Combined Intersection list of all the intersection points across any processor
                boundary}
```

```
perform Local_DRC;
{ Identify if there are LGs meeting Corners of sub-layout,
  Create CDC_Lists and UDC_Lists for all boundaries }
Communicate CDC_Lists with neighboring PEs;
Form CI_Lists corresponding to every Boundary;
for all elements in a CDC_Lists do
  begin
    Orthogonal Checks;
    Diagonal Checks;
  end; {this may result in some more entries in UDC_Lists}
Communicate UDC_Lists to neighboring PEs;
{hence receive appropriate UDC_Lists from neighboring PEs}
For all elements in a UDC_List do
    begin
      Orthogonal Checks:
      Diagonal Checks; { these diagonal checks may require communication with some
              neighboring processor; enter these elements in secondary-UDC_Lists}
    end;
  Communicate secondary-UDC_Lists to neighboring PEs;
  {hence receive appropriate secondary-UDC_Lists from neighboring PEs}
  Process secondary-UDC_Lists; {similar to UDC_Lists}
  Communicate information of LGs intersecting sub-layout corners to neighbors;
  Perform orthogonal and diagonal checks for corner cases;
end;
end.
```

The various steps in above algorithm are elucidated further below.

Layout Partitioning

To process the layout in parallel using P processors, we have to divide it into P sub-layouts and perform DRC on each of these sublayouts in parallel. The mapping of the sub-layouts onto the processors is shown in Figure 5. Here we show how the partitioning step itself can be performed in parallel. Assuming that the layout comprises N LGs, N/P LGs can be arbitrarily allotted to every processor. Each processor in turn processes the LGs allotted to it, and identifies the sub-layout to which that LG belongs. In case the LG belongs to two or more sub-layouts the LG is divided into the required number of clipped-LGs and the processors to which the clipped-LGs are allotted is decided. The final allotment of LGs and clipped-LGs is thus decided. Appropriate LGs and clipped-LGs can now be distributed to the appropriate processors such that each processor gets one sub-layout.

Local_DRC

In this procedure design rule checks are performed on all the LGs and clipped LGs present in every processor. This Local_DRC is very similar to the sequential DRC. The sub-layout is initially enveloped on all its sides by white nodes as described in [8]. The boundary of every LG of the sub-layout is travelled using the BFA. The BFA identifies the initial Northern most Black-white node pair (P,Q) on the LG. It then traces the boundary of the LG by determining the black-white node pairs such that the side which is common to the blocks of image corresponding to the nodes P and Q is an edge or part of an edge of an LG. It may be noted that no boundary codes are generated during the boundary traversal of any LG. Instead we have to calculate the length of every edge of the LG to check whether it is more than a certain limit specified by the design rules. Further, at all vertices certain extra orthogonal and diagonal checks have to be performed to verify the spacing (specified by the spacing design rule) between mask openings corresponding to any two LGs and to verify the width of any mask opening.

However, such checks cannot be successfully completed if the LG is clipped or is within the DRID of the sub-layout boundary. To process such LGs the processors have to communicate amongst themselves to perform design rule checks on the LGs. To be able to do so, the procedure Local_DRC identifies those points where a processor boundary is encountered while performing the orthogonal or the diagonal

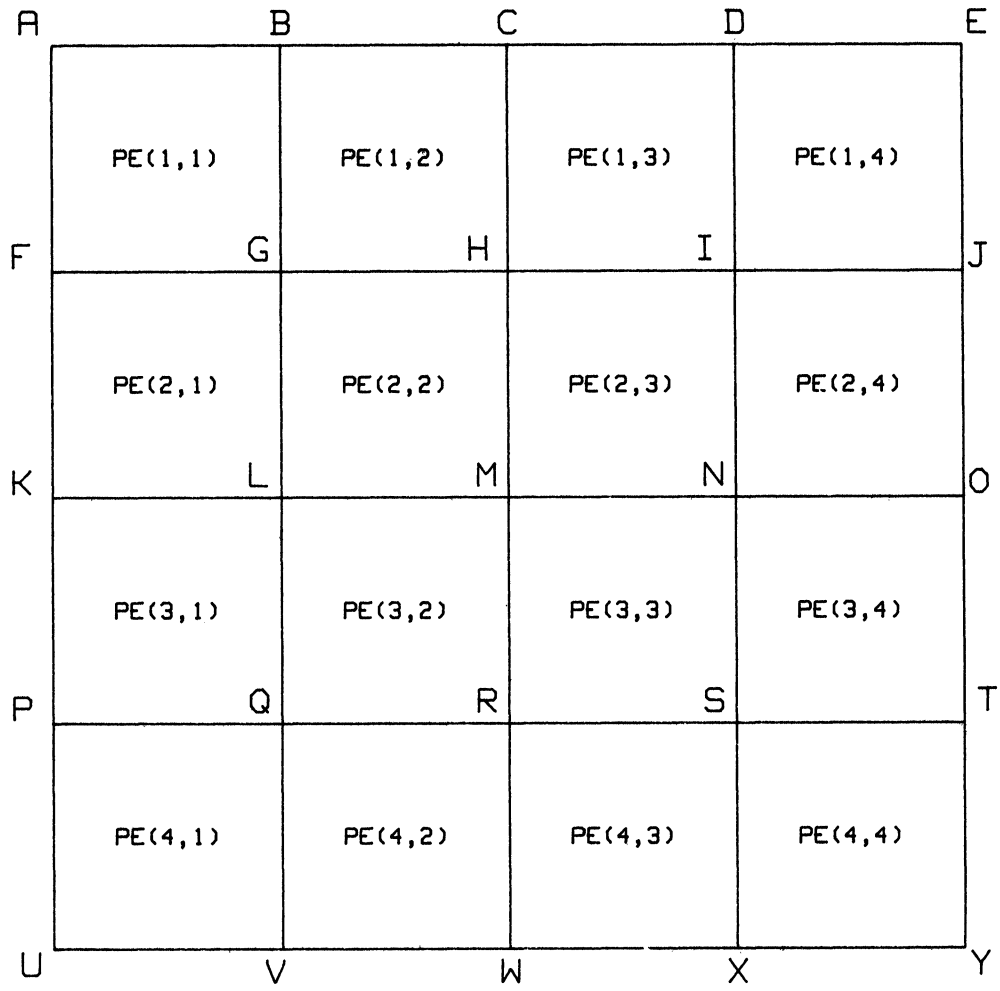


FIGURE 5 Mapping of the Sub-layouts on to 2-D Mesh of Processors.

checks. The intersection points along a processor boundary d, encountered while making orthogonal or diagonal checks for clipped LGs, are stored in lists called the Clipped_DRC_Completion list (CDC_List[d]) and those encountered during checks on LGs within a DRID from the processor boundary are stored in Unclipped_DRC_Completion lists (UDC_List[d]). Thus with every side, d, of a sub-layout we associate a CDC_List(d) and a UDC_List[d].

All the enveloping white nodes are distinguished from the sub-layout nodes by their K_values. This facilitates identification of clipped-LGs. Every side of the sub-layout has a starting point (called begin_point) and an ending point (called end_point), corresponding to a clockwise traversal of the sub-layout boundary. For example, the North side of a sub-layout has the NW corner as its Begin_point and the NE corner as the End_point, and the East side has the NE corner as its begin_point and SE corner as its end_point (see Figure 6). With every entry in the

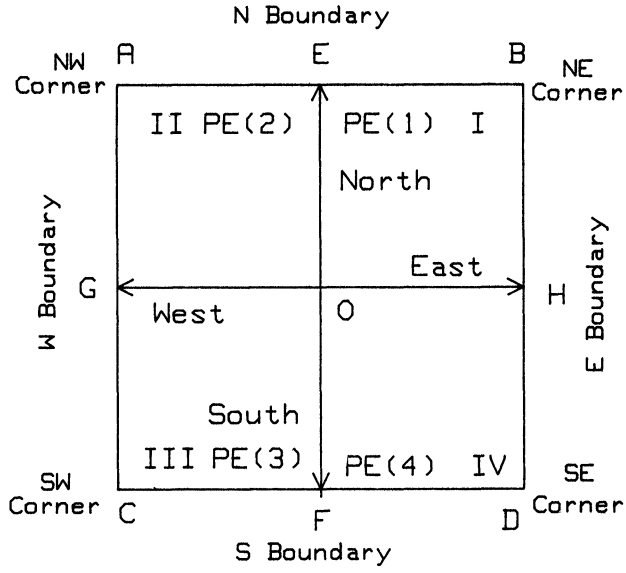


FIGURE 6 Quadrant to processor mapping to explain corner cases.

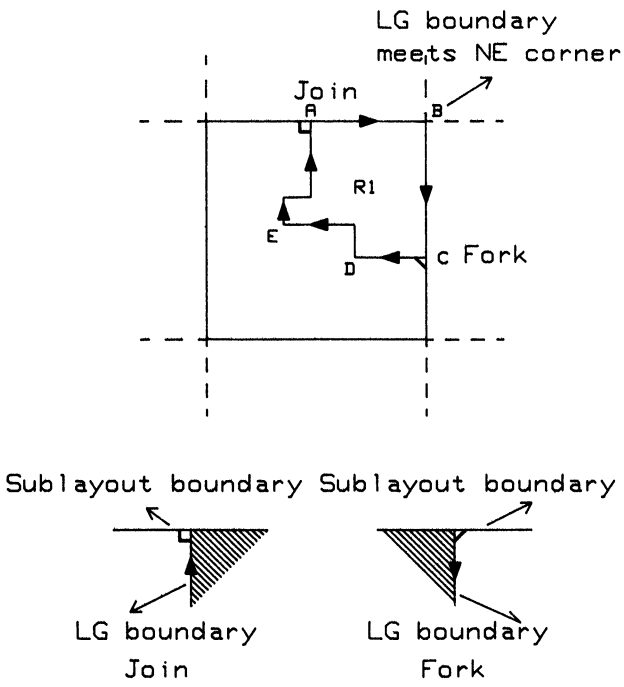


FIGURE 7 An LG illustrating join and fork.

CDC_List[d] and UDC_List[d], we also enter the distance of the corresponding intersection point from the starting point of side d.

DRC on LGs Near Processor Boundary

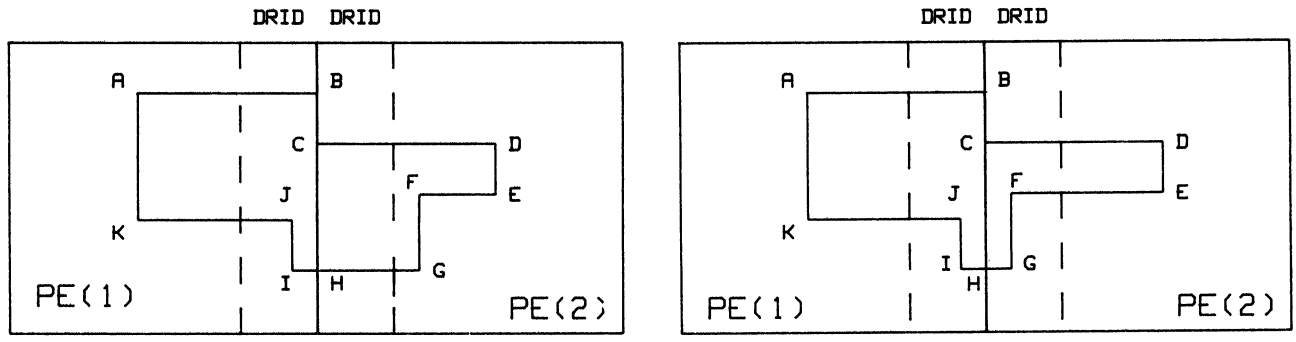
The lists corresponding to the boundary d, have to be sent to the processor's neighbor along that boundary and the corresponding list has to be received from that neighbor. This interchange of the lists corresponding to boundary d, allows any two processors to complete the orthogonal and diagonal checks on the LGs that have been clipped while partitioning the layout along boundary d and on LGs that are within the design rule interaction distance from the boundary d.

Completing checks on Clipped-LGs To make an entry in the CDC_List corresponding to any side d, we have to identify if the LG boundary intersects that side, and if so, then we have to identify if the LG boundary joins the processor boundary or forks away from it (Figure 7). This information is stored in the booleans JOIN and FORK respectively. For LG boundaries that meet corners, we require certain extra checks which are explained in the section Processing LGs Crossing at Corners.

In the CDC_List[d] we store all points where the boundary d is encountered while performing orthogonal checks on the various LGs. In case the edge of a clipped-LG which meets the layout boundary, exceeds the minimum distance required by the design rule, there is no error associated with that edge even if it extends past the neighboring processors boundary. But, if an edge of a clipped-LG meets a processor boundary and has a length which is less than that specified by the design rule, we are not sure if there is an error associated with that edge (Figure 8). This is so because that edge may extend into the sub-layout of the neighboring processor and the total length of that edge (associated with the complete LG of which the clipped-LG forms a part) may be greater than that specified by the design rule.

Thus in every node of the CDC_List we need to store

1. the position of the intersection point (in terms of its distance from the begin point of that Boundary) and
2. a boolean ERROR, that indicates if the length of the edge intersecting the processor boundary



(a) no width error associated with segment IH (b) width error associated with IH

FIGURE 8 A clipped LG illustrating that PE(1) must communicate with PE(2) to determine the error associated with IH.

is below the permitted width of the mask opening.

3. if ERROR = TRUE then we also store the length of the corresponding edge in LENGTH.

Processing CDC_lists It may be noted that every boundary of a sub-layout is processed by two processors. Thus there are two CDC_Lists corresponding to a sub-layout boundary (each being present in a different processor). Thus the processor boundary HM in Figure 5 is processed by PE(2, 2) and PE(2, 3). The processor boundary HM forms the side_of(E) of the processor PE(2, 2) and the side_of(opposite_dir (E)) = side_of(W) of PE(2, 3). A procedure to process the two CDC_Lists corresponding to the sides of every sub-layout, completes the design rule check on the clipped-LGs along that boundary.

To perform the above tasks we need a list which contains all the intersection points of LG boundaries with the processor boundary for each of its sides. Such a list is obtained by merging the two CDC_Lists associated with a particular boundary. For this, the two processors which share that side of the sub-layout boundary exchange their CDC_Lists (associated with that side) and merge it (independently). Thus a combined intersection list (CI_List) associated with that side is available in both the processors. These lists are processed by the two processors independently.

For instance, consider two consecutive intersection points (P_1, P_2) corresponding to node pairs ($T1$, $T2$) from CI_List as shown in Figure 9, case IV. Let $E1$ and $E2$ be the edges incident on the points P_1 and P_2 respectively. If the two points are not coincident (i.e. $d1 < d2$) then together with $E1$ and $E2$ there is another edge existing between P_1 and P_2. It is possible that there is a design rule violation along either the edges $E1$ and $E2$ or the edge between P_1 and P_2. If P_1 and P_2 are coincident points ($d1 = d2$) then the two edges $E1$ and $E2$ are colinear and can possibly be combined to form a third edge E. There can be an error along the edge E only if there is an error on $E1$ and $E2$ both and the combined length of $E1$ and $E2$ (i.e. the length of E) is also in error.

Given a pair of intersection points (P_1, P_2), we can identify all errors associated with that pair. Thus we have to process all possible node pairs of the combined intersection list (CI_List) corresponding to pairs of consecutive (or coincident) intersection points present on the sub-layout boundary. Whenever there is no ambiguity, we shall refer to the terms intersection points and nodes in the CI_List corresponding to those intersection points interchangeably. This can best be performed by sorting the CI_List in the increasing order of the distances of the intersection points from the begin_point (associated with the side) in both the processors.

Assume that the PE(2,2) shown in Figure 5 is processing the intersection points on the processor boundary corresponding to E side of the sub-layout. PE(2, 2) therefore receives the CDC_List from PE(2, 3) corresponding to the W side of its (of PE(2, 3)'s) sub-layout. PE(2, 2) combines the following two lists: (1) Its own CDC_List associated with side E and (2) the CDC_List of PE(2, 3) associated with side W. This results in the CI_list of PE(2,2) associated with side E. PE(2, 2) then sorts this CI_list in the increasing order of the distance of the intersection points from the begin_point of side E (i.e. point NE). We describe how the node pairs ($T1$, $T2$) from the CI_Lists are processed below.

Resolving errors across processor boundaries The node pairs from the CI_List ($T1$, $T2$) are taken one by one and processed as follows. Let the PE(2, 2) refer to itself as this_PE and the neighboring PE i.e. PE(2, 3) as other_PE. Hence every pair of the ($T1$, $T2$) is characterised by the following features:

1. Whether $T1$ belongs to this_PE or other_PE,
2. Whether $T2$ belongs to this_PE or other_PE,
3. Whether $T1$ joins or forks at the intersection with the sub-layout boundary.
4. Whether $T2$ joins or forks at the intersection with the sub-layout boundary.

Thus, for different characterizations of the above features, design rule errors associated with two consecutive elements ($T1$, $T2$) of the CI_list can be uniquely determined. Since each of the features can be characterised by boolean values, there exist in all 16 combinations of the four features, some of which are invalid for actual VLSI layouts. Actions for individual cases are enumerated in algorithm interpret _CDC_List in the Appendix B and illustrated in Figure 9. Here we give details of two possible cases. Let $d1$ and $d2$ denote the distance of $T1$ and $T2$ from the begin_point of side E of this PE respectively and let $E1$ ($E2$) be the edge associated with the intersection point corresponding to $T1$ ($T2$). Since the intersection points of the CI_list are sorted, either $d2 > d1$ or $d2 = d1$.

a) First we assume that $T1$ belongs to this PE, $T2$ belongs to the other PE, $T1$ joins the sub-layout boundary whereas $T2$ forks and $d2 > d1$. This situation corresponds to the case XIII(a) shown in Fig-

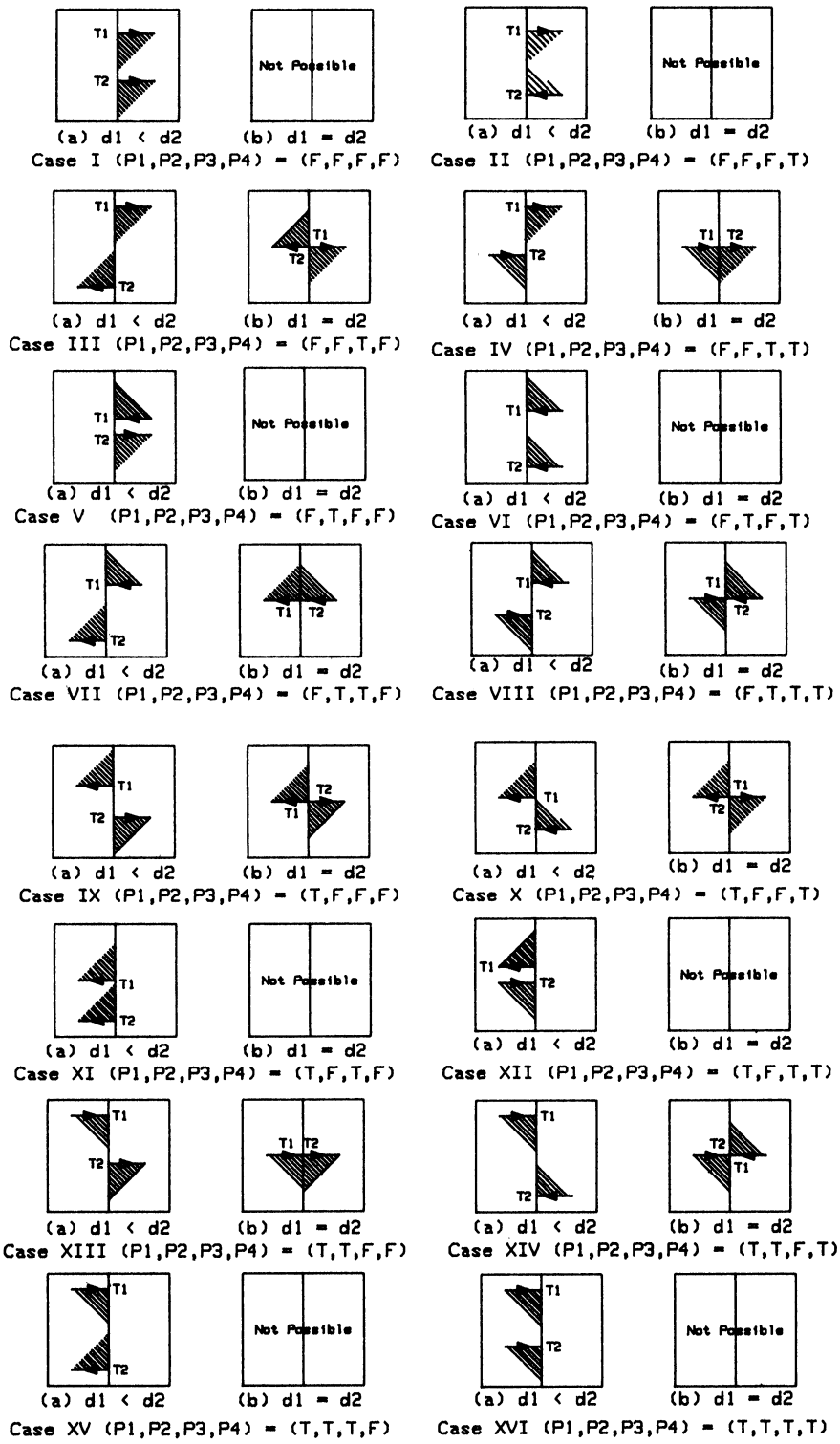


FIGURE 9 Interpretation of Regions for Intersection Pair (T1, T2).

ure 9. The errors associated with this case can be reported as follows:

1. if there is an error associated with $E1$ then report it.
2. if there is an error associated with $E2$ then report it.
3. If ($d2$-$d1$) is less than that specified by the design rule then it is an error that has to be reported.

For the case $d2 = d1$ (case XIII(b)), the edge $E1$ continues as the edge $E2$ in the other PE. Hence there is only one possible error associated with this pair i.e. if $E1$ and $E2$ both are in error and the sum of the length of $E1$, $E2$ is also an error then report error. It may be noted that in this case if any one of the edge is not in error then there is no error associated with the combined edge.

b) Next we consider the case when $T1$ and $T2$ both are in this PE and both join the sub-layout boundary and $d2 > d1$. It can be seen from Figure 9. case XVI(a) that such a case is not possible since two regions cannot overlap or in other words LG boundaries by definition cannot occur embedded within other regions. Similarly the corresponding case when $d2 = d1$ (case XVI(b)) also cannot occur.

It may be noted that while processing CDC_Lists we may be required to perform diagonal checks which are composed of two orthogonal checks in directions perpendicular to each other. If the intersection point corresponding to the node of the CDC_Lists being processed is within the design rule interaction distance from a corner of the sub-layout and the orthogonal check requires us to travel in a direction towards a processor boundary, the check can be completed only by the neighboring processor. This can easily be achieved by making appropriate entries in the UDC_List corresponding to the PE boundary encountered.

Completing checks on Unclipped-LGs For LGs that are within design rule interaction distance from the processor boundary certain extra orthogonal and diagonal checks have to be performed at the corners of the LGs to verify the spacing (specified by the spacing design rule) between mask openings corresponding to any two LGs and to verify the width of any mask opening. It may be recalled that in order to perform orthogonal checks we have to travel a distance d in two orthogonal directions from the corners of an LG. Similarly for diagonal checks, we have to travel distance $d1$ along a certain direction followed by a distance $d2$ in a direction perpendicular to the previous direction. In case the point from where we start an orthogonal or diagonal check is within the DRID of the processor boundary (i.e. either $d1$ or $d2$ mentioned above) it is possible that we meet the processor boundary before we are able to complete the diagonal check. These orthogonal and diagonal checks can only be completed by the other processor. The other processor may be required to make the following two types of traversals:

1. travel a certain distance x, starting from a point P on the processor boundary such that it moves inwards and away from P.
2. travel a distance x as in (1) followed by a distance y in a direction normal to the previous direction in either the clockwise sense or in the counter-clockwise sense.

For all such cases, where we are required to cross the processor boundary, we store the above information (as required in case 1 or 2) in a list of nodes called Unclipped_DRC_Completion list (UDC_List). Thus the other processor has to be informed of following information: (1) from which point it has to start travelling certain distance, (2) how much distance it has to travel and (3) in which direction it has to travel.

These UDC_Lists are communicated to the appropriate neighbors. Hence the corresponding UDC_Lists are received by every PE from its neighbors. The intersection point at the boundary has to be located in the quadtree. From that point onwards a certain distance has to be travelled (as specified in the UDC_List node) and the required orthogonal or diagonal check has to be made.

If the intersection point corresponding to the node of the UDC_Lists being processed is within the DRID from a corner of the sub-layout and the orthogonal check requires us to travel in a direction towards a processor boundary, the check can be completed only by the neighboring processor. To perform such a check we form a secondary-UDC_List and make required entries in the secondary-UDC list corresponding to the boundary encountered.

This step is followed by a communication step to send the secondary-UDC_Lists to the neighboring processors and design checks are carried out in a manner identical to the processing of UDC_Lists. It can be seen that the checks performed while processing secondary-UDC_Lists do not require further communication between processors.

Processing LGs crossing at corners In the section Completing Checks on Clipped-LGs above we mentioned that we identify if any LG meets any corner or not. In Figure 6 we assume that a layout has to

be allotted to four processors and hence is partitioned into four sub-layouts. Assuming that line EF forms the Y-axis of a coordinate system, line GH forms the X-axis and point O forms the origin then we can associate a sub-layout corresponding to every quadrant of the coordinate system. It can be seen from the Figure 6 that the quadrants I, II, III and IV correspond to the sub-layouts EBOH, AEOG, GODF and OHCF respectively. We assume that the sub-layout corresponding to the quadrants are assigned to the processors PE[1], PE[2], PE[3] and PE[4] respectively. The point O forms one corner of each of these sub-layouts e.g. it forms the SW corner of the sub-layout allotted to PE[1], and SE corner of the sub-layout allotted to PE[2] etc. We assume an array B[1 . . 4] of booleans where B[i] indicates if there is an LG in PE[i] such that O coincides with one of its corners. The array B indicates if there is any LG that either touches or includes point O and if it does, then B indicates the number of parts into which that LG gets divided (depending on the number of elements in B that are TRUE) and the processors to which these parts of the LG get allotted (depending on which elements of B are TRUE). Figure 10 gives certain representative corner cases and their boolean assignments.

Below we present a scheme to compute the array B. All the four PEs initialize all the elements of array B to FALSE. Then the PE[i] sets the B[i] = TRUE if there is an LG which has a corner coinciding with point O. This computation can be performed while carrying out the procedure Local_DRC. After this step all the processors communicate with their neighbor along the Y-axis (i.e. PE[1] communicates with PE[4] and PE[2] communicates with PE[3]) and perform OR operation on the B vectors associated with

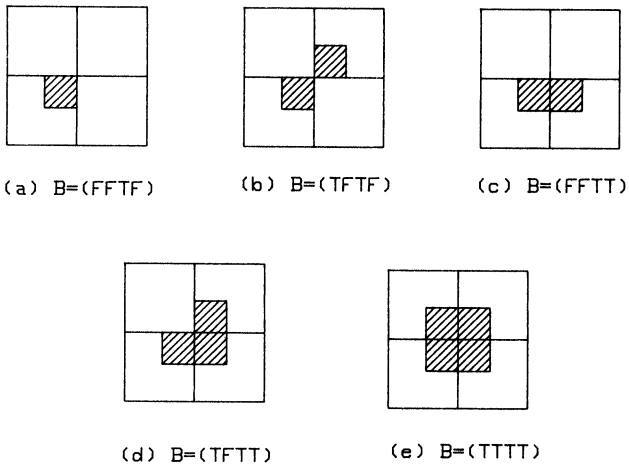


FIGURE 10 Examples of LGs intersecting corners and the corresponding boolean assignment of B.

corner O. Next the processors communicate along the X-axis and perform OR operations on the resulting B vectors. The B vector after the above steps stores the complete information regarding the corner at point O and the same value of B vector is present in all processors. The above procedure can be generalised to any corner. After the above operations every processor surrounding a corner receives details of the corner and hence the required orthogonal or diagonal checks can be made by the processors. Details of the corresponding procedure are given in procedure Process_Corner_case in Appendix B.

ANALYSIS OF THE PARALLEL ALGORITHM

In this section we present the complexity analysis of the various steps of the parallel DRC on the mesh connected processors. We assume the layout can be represented using $m \times m$ pixels where $m = 2^q$ and each pixel denotes an area of lambda $\times$ lambda. We assume that there are $n \times n$ processors in the mesh, $n = 2^p$. The time complexity of the individual steps of the parallel DRC is given below.

1) Partitioning the layout: Assume the total number of LGs to be N and the number of processors to be P. Since every processor processes N/P LGs, on an average it can be assumed that the time required for the partitioning step is $O(N/P)$ assuming that an LG can get clipped into atmost constant number of clipped-LGs.

2) Local DRC: In this procedure we need to compute the distances of all the intersection points between an LG-boundary and sub-layout boundary from the corresponding begin_points. Every such distance computation requires time proportional to the effective width of the K_values of the nodes of the sub-quadtree allotted to one processor which is $O(q - p)$. Since there may be $4*2^{q-p}$ such intersection points in the worst case we require $O(2^{q-p}(q - p))$ time for distance computation. Besides, for performing the boundary following of the LGs we need $O(\Sigma p_i)$ time where Σp_i is the sum of the perimeters of the LGs and clipped-LGs present within the sub-layout allotted to every processor. Hence the overall time required for the procedure Local_DRC is $O(2^{q-p}(q - p) + \Sigma p_i)$. If we assume that the number of LGs per unit area is constant and the average perimeter of the LGs is constant (i.e. it is independent of the total number of LGs in the layout), there exists a linear relation between the total number of LGs and the area of the sub-layout. Thus $O(\Sigma p_i)$ can also be written as $O(2^{2(q-p)})$ since the area of the sub-layout is $2^{2(q-p)}$ and the number of LGs is propor-

tional to the area of the sub-layout. Thus the time required for Local_DRC becomes $O(2^{q-p}(q-p) + 2^{2(q-p)}) = O(2^{2(q-p)})$.

3) Process UDC_Lists and CDC_Lists: The time required for this procedure is dominated by the time required to sort the elements present in the combined intersection list. Thus if 'r' represents the number of LG boundary crossings across any boundary, the time required for the sorting step is $O(r\log r)$ and the time required for the remaining steps is $O(r)$. Since the number of LG boundary crossings at any boundary can be no more than 2^{q-p}, the time required for the overall procedure is $O(r\log r) = O(2^{q-p}(q-p))$.

It may be noted that the sorting step dominates the time required for processing the CDC_lists and UDC_Lists. It is known a priori that in the worst case there may be 2^{q-p} elements in the CDC_Lists (UDC_Lists) where each key (being the distance of the intersection point from the begin_point of the side under consideration) of the elements to be sorted is an integer less than 2^{q-p} and greater than zero. Also no two keys have the same value since at a point only one LG boundary may intersect a sub-layout boundary. For sorting such elements a hash based sorting is well suited. A simple hashing function where the ith key is hashed onto the ith memory location can be adopted. This sorting scheme requires an additional memory of $O(2^{q-p})$. Hence with negligible extra storage the hashing scheme enables us to sort the r elements in $O(r)$ time. This results in an $O(2^{q-p})$ time for the overall procedure.

Hence the overall time complexity of the parallel DRC is $O(2^{2(q-p)})$. Thus for a layout of size $m \times m$ pixels and a processor mesh of $n \times n$ processors, the time required for parallel DRC is $O((m/n)^2)$. It can be seen that if we process the overall layout sequentially, the time required is same as that of performing BFA on all LGs. This time is $O(\Sigma p_i)$ where the summation of the perimeters is carried out over the entire layout. Hence we require time proportional to the area of the layout i.e. $O(m^2)$. Hence it can be seen that the speedup obtained is $O(n^2)$ which is same as $O(P)$ for $P = n \times n$ processors.

CONCLUSIONS

In this paper we provided a parallel algorithm to perform Design Rule Checking of Layout geometries in a VLSI layout. The algorithm assumes the parallel architecture to be a two-dimensional mesh of processors. The algorithm is based on a linear quadtree representation of the layout. Through a complexity analysis it is shown that a linear speedup with respect to the numer of processors is possible. Though the algorithm for parallel DRC was presented based on linear quadtree representation of a layout, this by no means is necessary and the algorithm can be developed for a pointer based quadtree as well. The assumption of an $n \times n$ processor mesh can be relaxed and readily extended to a mesh $n1 \times n2$ processors where $n1 \neq n2$. It is possible to make either of $n1$ or $n2$ to be equal to one resulting in a linear array. The algorithm can be easily adapted to interconnection topologies of higher connectivity such as higher dimensional meshes or hypercube since it is possible to embed two-dimensional meshes onto these topologies.

Acknowledgements

We gratefully acknowledge the contribution of V. Rajaraman, Chairman, Centre for CAD and Super Computer Education and Research Centre, Indian Institute of Science, for initiating this research and for his valuable assistance and support throughout. We also acknowledge the help rendered by Y.V.S.P. Rao and K. Sriram for the preparation of the manuscript and Ms. Premlata for her efforts in simulating the algorithm.

References

[1] N.B. Bhat and S.K. Nandy, "Special Purpose Architecture for Accelerating Bitmap DRC," *Proceedings of the 26th ACM/IEEE Design Automation Conference,* 1989.

[2] G.E. Bier and A.R. Pleszkun, "An Algorithm for Design Rule Checking on a multiprocessor," *Proceedings of the 22nd ACM/IEEE Design Automation Conference,* July 1985.

[3] F. Gregoretti and Z. Segall, "Analysis and evaluation of VLSI Design Rule Checking Implementation in a Multiprocessor." *Proceedings of the 13th Annual International Conference on Parallel Processing,* August 1984.

[4] S.K. Nandy, "Geometrical Design Rule Check of VLSI layouts in Distributed Computing Environment," *International Journal of Computer Aided VLSI Design* (to appear).

[5] S.K. Nandy and L.M. Patnaik, "Linear Time Geometrical Design Rule Checker based on Quadtree representation of VLSI Mask Layouts," *Computer-Aided Design,* Butterworth & Co. (Publishers) Vol. 18, No. 7, September 1986, pp. 380–388.

[6] S.K. Nandy, R. Moona, and S. Rajagopalan, "Linear Quadtree Algorithms on the Hypercube," *Proceedings of the 17th Annual International Conference on Parallel Processing,* 1988.

[7] R.B. Panwar and S.K. Nandy, "Parallel Architecture for Boundary Following of Regions of an Image Stored as Linear Quadtree," *Proceedings of the 26th Annual Allerton Conference on Communication, Control and Computing,* 1988.

[8] C.R. Dyer, A. Rosenfeld, and Hanan Samet, "Region Representations Boundary Codes from Quadtree:" *Communications of the ACM,* Vol. 23, No. 3, pp. 171–179, March 1980.

Biographies

S.K. NANDY obtained his B.Sc (Hons) Physics degree from Indian Institute of Technology Kharagpur in 1977, and subsequently obtained B.Engg. (Hons) in Electronics and Communications Engineering in 1980, MSc. (Eng.) in Computer Science and Engineering in 1987, Doctor of Philosophy in Computer Science and Engineering in 1989 from the Indian Institute of Science, Bangalore, INDIA. His research interests are in the areas of VLSI

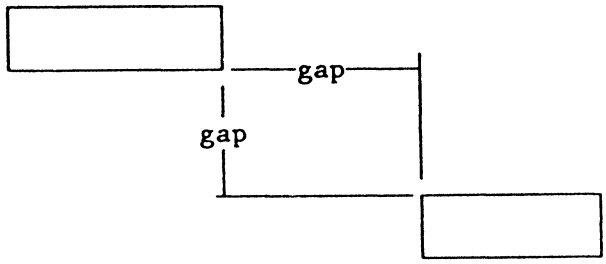
CAD, Parallel and Distributed Architectures, Dataflow Computing, Architectural Synthesis of Low Latency, High Throughput systems on silicon, High Level Synthesis, etc. He has several research papers presented at International Conferences and Journals. He is presently an Assistant Professor at the Computer Aided Design Laboratory and the Department of Electrical Communications Engineering of the Indian Institute of Science, Bangalore, INDIA.

R.B. PANWAR obtained his Bachelor's degree in Engineering from the University of Nagpur, INDIA in 1986. Subsequently, he obtained the Master's degree in Computer Science from the Department of Computer Science and Automation of the Indian Institute of Science, Bangalore, INDIA. He is presently a research scholar at the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, USA. His research interests span parallel algorithms for linear algebra, actors and VLSI theory and design.

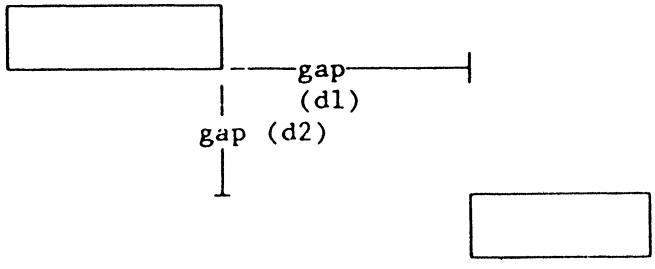
APPENDIX A

Design rule checking is a process of determining whether certain interrelationships and constraints are maintained between the various layout geometries in a layout. The various design checks that are carried out on LGs in a layout can be categorised as follows*:

Check #1. minimum width rectangular constraint
Check #2. maximum width rectangular constraint
Check #3. minimum gap circular or square constraint

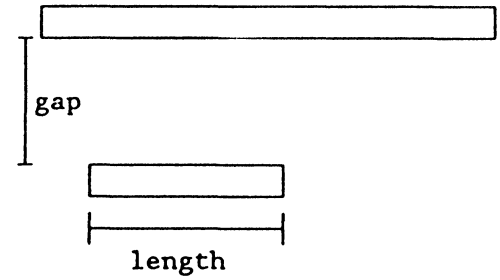


Check #4. minimum gap rectangular constraint

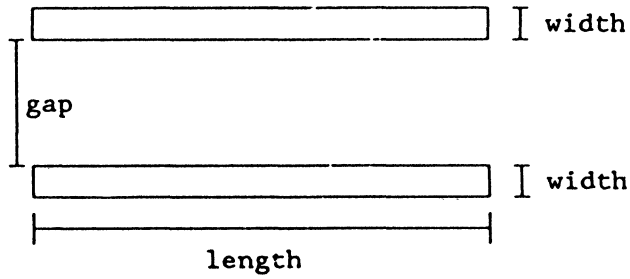


*All checks specified are based on personal communications with Dr. T. G. R. Van Leuken, Delft University, The Netherlands.

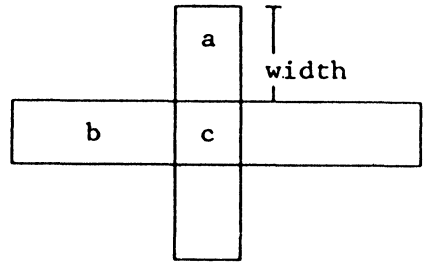
Check #5. gap depending on length of parallel track if length < n then gap > a else gap > h



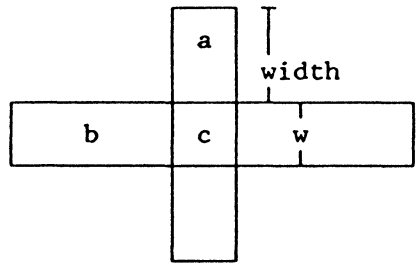
Check #6. gap depending on width of parallel tracks if width < b then gap > a else gap > b



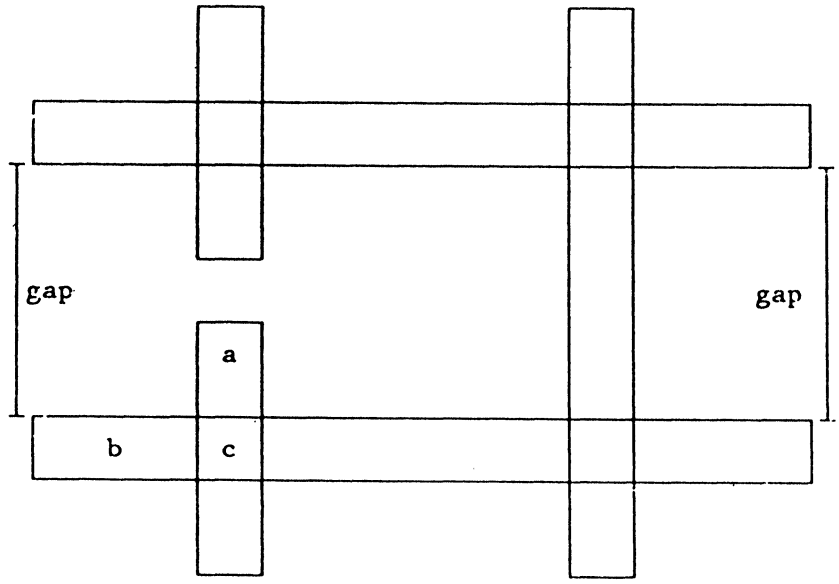
Check #7. width depending on adjacent mask (combination) if mask c is present then width > A else width > B



Check #8. width depending on adjacent mask (combination) width if mask c is present and w > n then width > A else width > B



Check #9. gap depending on electrical voltage if electrically related then gap > A else gap > B



These checks can be easily carried out by following the boundaries of LGs in different layers of the layout and they can be classified as orthogonal checks and diagonal checks to meet the rectangular and square constraints.

APPENDIX B

In the following we provide a formal description of the algorithm for parallel DRC in pseudo Pascal. We define the following structures and functions to be used in the description of the algorithms to follow.

```
type
  corner_type = (NE, NW, SW, SE);
type
  CDC_node = record
    crosses_at_corner              : Boolean;
    corner_name                    : corner_type;
    JOIN, FORK                     : boolean;
    PE_id.                         : integer;
    distance {from begin_point}    : integer;
    Fill_later                     : boolean;
    ERROR_SO_FAR                   : boolean;
    ERROR_LENGTH :                 : integer:
    Next_node                      : pointer to the next CDC_node;
  end;
type
  UDC_node = record
    distance {from begin point}    : integer;
    check                          : (width, spacing);
    two_traversals                 : boolean;
    case two_traversals of
      false : begin
                {only one traversal}
                d1                 : direction;
                {travel in the direction d1}
                l1                 : integer;
                {distance l1 to be travelled in direction d1}
              end;
```

```
        true : begin
                    d1, d2              : direction;
                    {first travel along d1 and then along d2}
                    l1, l2              : integer;
                    {travel 11 distance along d1 and l2 distance along d2}
                  end;
      Next_node                         : pointer to next UDC_node;
    end;
type status = array[NE..SE] of boolean;
var
    corner_status                       : array[NE..SE] of status;
      {e.g. corner_status[NW] gives
      the status of the NW corner of this PE}
var
    CDC_list                            : array[N..W] of CDC_node;
      { CDC_list is Clipped_DRC_Completion list }
    UDC_list                            :array[N..W] of UDC_node;
      { UDC_list is Unclipped_DRC_Completion list }
    secondary_UDC_list                  :array[N..W] of UDC_node;
      { secondary_UDC_list is Unclipped_DRC_Completion list which is
formed while processing UDC_list; it contains UDC_nodes with
two_traversals = false only }
Procedure Partition_Layout;
begin
    {Assume input available as a CIF file, N rectangles and P
    processors}
    Randomly allot N/P rectangles to each processor;
    {so as to partition the layout in parallel}
    { Let every processor contain Q[1..P], an array of queues where
    the ith element of Q will contain all the rectangles allotted to
    the processor i}
    for each processor do in parallel
      begin
        for every rectangle do
          begin
            if the complete rectangle can be allotted
                to the ith processor
              then add the rectangle to Q[i]
            else
              if the rectangle spans k processors (j1, j2, . . . , jk)
                then form k clipped rectangles out of the
                given rectangle and add to appropriate queues;
            end;
        end;
      combine the Qs present in different processors to obtain single
      queue for every processor;
    end; { of Partition_Layout }
Function Boundary_white(Q:node):boolean;
begin
    If Q is a white node that belongs to the enveloping nodes
        then Boundary_white: = true
    else Boundary_white: = false
end; { of Boundary_white }
Procedure K_to_coord(k : K_value; var x,y :integer);
```

```
  { K_value is an array [1..max_level], (x,y) return the co-ordinates
  of the center of the block of image corresponding to the node with
  K_value k; Let the size of overall image be 2Lx2L and the center of
  the image is point (0,0) }
begin
  x:=0; y:=0; l:=L;
  for i:=1 to max_level do
    begin
      case k[i] of
        1 : x:=x-1/2; y:=y+l/2;
        2 : x:=x+1/2; y:=y+l/2;
        3 : x:=x-1/2; y:=y-l/2;
        4 : x:=x+1/2; y:=y-l/2;
      end;
      l:=l/2;
    end;
end;
Function convex : boolean;
  {this function returns TRUE if the given corner point is convex and
  returns FALSE if it is concave}
begin
  if previous_white = current_white then convex := false;
  else
    if previous_black = current_black then convex := true;
  end;
procedure orthogonal_spacing_check(Z : direction; minimum_spacing :
                                        integer);
begin
  {perform orthogonal spacing check from the current corner point in
  the direction Z}
  travel in the Z direction starting from the current black node;
  length := 0; let NODE be the neighboring white node of
  current_black, in direction Z; black_node_encountered := false;
  while (black_node_encountered) and (length < minimum_spacing ) do
  begin
    length := length + edge_length_of(NODE);
    NODE := neighboring node of NODE in direction Z;
    if NODE is Boundary_white then
    begin
      create a UDC_node;
      with UDC_Node do
        begin
          check := spacing;
          two_traversal := false;
          d1 := Z;
          l1 := (minimum_spacing - length);
          distance {from begin point} := distance from begin point
            of the boundary encountered to the point
            of intersection along the boundary;
        end;
      end;
      if NODE is black then black_node_encountered := true;
    end;
```

```
    if black_node_encountered and length < minimum_spacing then
      spacing ERROR;
end; { of orthogonal_spacing_check }
procedure orthogonal_width_check(Z : direction; minimum_width :
                                        integer);
begin
  { perform orthogonal width check from the current corner point in
    the direction Z }
  { procedure analogous to orthogonal_spacing_check except that we
    travel black nodes in direction Z until a white node is
    encountered or distance traversed is greater than minimum width;
    also, if any UDC_Node is created we make the corresponding
    UDC_Node.check := width }
end;
  procedure diagonal_spacing_check(X,Y : direction);
  begin
    { perform diagonal spacing check from the current corner point in
    the direction (X,Y) }
    { this corresponds to (a) orthogonal spacing check of E1 along X
    followed by an orthogonal spacing check of E2 along Y (b) orthogonal
    spacing check of E2 along Y followed by an orthogonal spacing check
    of E1 along X }
      travel in the X direction starting from the current black node;
      length := 0;
      let NODE be the neighboring white node of
      current_black, in direction Z;
      black_node_encountered := false;
      while (black_node_encountered) and (length < minimum spacing ) do
      begin
        length := length + edge_length_of(NODE);
        NODE := neighboring node of NODE in direction Z;
        if NODE is Boundary_white then
        begin
          create a UDC_node;
          with UDC_Node do
          begin
            check := spacing;
            two_traversal := true;
            d1 := X; d2 := Y;
            l1 := (E1 - length); l2 := E2;
            distance {from begin point} := distance from begin point
              of the boundary encountered to the point
              of intersection along the boundary;
          end;
        end;
        if NODE is black then black_node_encountered := true;
      end;
      if black_node_encountered and length < E1 then
          spacing ERROR
      else
      begin
        travel in the Y direction starting from the current black node;
        length := 0;
        let NODE be the neighboring white node of
```

```
        NODE (last processed), in direction Z;
      black_node_encountered := false;
      while (black_node_encountered) and (length < E2) do
        begin
          length := length + edge_length_of(NODE);
          NODE := neighboring node of NODE in direction Z;
          if NODE is Boundary_white then
            begin
              create a UDC_node;
              with UDC_Node do
                begin
                  check := spacing;
                  two_traversal := false;
                  d1 := Y;
                  l1 := (E2 - length);
                  distance {from begin point} :=
                    distance from begin point of the
                    boundary encountered to the point of
                      intersection along the boundary;
                end;
            end;
          if NODE is black then black_node_encountered := true;
        end;
      if black_node_encountered and length < E2 then
          spacing ERROR;
    end;
    {So far we have completed orthogonal spacing check of E1 along X
    followed by an orthogonal spacing check of E2 along Y. Now perform
    orthogonal spacing check of E2 along Y followed by an orthogonal
    spacing check of E1 along X similarly}
  end; { of diagonal_spacing check }
procedure diagonal_width_check(X,Y : direction);
begin
  { perform diagonal width check from the current corner point in
    the direction (X,Y) }
  { this corresponds to
    (a) orthogonal width check of E1 along X
    followed by an orthogonal width check of E2 along Y
    (b) orthogonal width check of E2 along Y followed by
    an orthogonal width check of E1 along X }
end;
Procedure orthogonal_&_diagonal_checks;
begin
    Let X := previous_d;
      Y := opposite_of(current_d);
    if {the corner point is} convex then
      begin
        Orthogonal_spacing_check( X direction);
        Orthogonal_spacing_check( Y direction);
        {this completes the orthogonal check}
        diagonal_spacing_check (X,Y);
        {this completes the diagonal spacing checks at the corner}
        diagonal_width_check (opposite(X), opposite(Y));
        {this completes the diagonal width checks at the corner}
```

```
      end
    else
      begin
        {the corner is covcave}
        Orthogonal_width_check(X direction);
        Orthogonal_width_check(Y direction);
        {this completes the orthogonal check}
        diagonal_width_check (X,Y);
        {this completes the diagonal spacing checks at the corner}
        diagonal_spacing_check (opposite(X), opposite(Y));
        {this completes the diagonal width checks at the corner}
      end;
    {if the sub-layout boundary is encountered while making the above
    traversals then a new UDC_node is entered in the UDC_list
    indicating the additional length to be traversed (by the
    neighboring processor) to detect an error if any }
end; { of orthogonal_&_diagonal_checks }
procedure Process_Corner_case(b : status);
begin
  case b[NE], b[NW], b[SW], b[SE] of
    { each of the check indicated below has to start from point 0 and
      proceed as specified below}
    { let X := opposite (previous_d) and Y := current_d }
    { the direction (X,Y) indicates a direction bisecting the angle
      formed by X and Y}
    { the function opposite gives the opposite of any direction }
    (F, F, F, F) : no operation;
    (F, F, F, T) : perform orthogonal check in X;
    (F, F, T, F) : let T1 be the first node of CDC_List(Y);
                     if T1.ERROR_SO_FAR then report error;
    (F, F, T, T) : let T1, T2 be the first two nodes of CDC_List(X);
                   If T1.ERROR_SO_FAR and T2.ERROR_SO_FAR and
                   (T1.ERROR_LENGTH + T1.ERROR_LENGTH) < design rule
                   length then report error;
                   advance pointers T1 and T2 by one CDC_node;
    (F, T, F, F) : orthogonal check in direction Y;
    (F, T, F, T) : orthogonal check in X, orthogonal check in Y;
    (F, T, T, F) : no operation;
    (F, T, T, T) : perform orthogonal check in X;
                   perform orthogonal check in Y;
                   perform diagonal check in (X,Y);
    (T, F, F, F) : perform orthogonal check in X;
                   perform orthogonal check in Y;
                   perform diagonal check in (X,Y);
    (T, F, F, T) : no operation;
    (T, F, T, F) : no operation; {action for (T1, T2) will be taken as
                                  in process_CDC_List};
    (T, F, T, T) : orthogonal check in direction Y;
    (T, T, F, F) : no operation;
    (T, T, F, T) : diagonal check in direction (X,Y);
    (T, T, T, F) : orthogonal check in direction X;
    (T, T, T, T) : no operation;
                   only advance pointers T1 and T2 by one CDC_node;
  end;{ end of the case statement}
  end; {end of Process_Corner_case}
```

```
procedure process_UDC_list;
begin
  for dir := N to W do
    for each UDC_node of the UDC_list[dir] do
      begin
        with the UDC_node do   .
          begin
            locate the intersection point
            {using the distance variable in the UDC_node}
            if not two_tranversal then
              begin
                if check = spacing then
                  travel white nodes for a distance l1 along
                    direction d1 and report spacing error if any
                  else
                    travel black nodes for a distance l1 along
                    direction d1 and report width error if any;
              end
            else
              begin
                if check = spacing then
                  begin
                    travel white nodes for a distance l1 along
                    direction d1 followed by distance l2 along
                    d2 and report spacing error if any ;
                    If processor boundary encountered
                      while travelling distance l2 then
                      create a suitable UDC_node and
                      enter it in secondary-UDC_list and
                      set UDC_Node.check := spacing;
                  end
                else perform width check similarly;
              end;
          end; {of with}
end; {of process_UDC_list}
procedure process_secondary-UDC_list;
begin
  for dir := N to W do
    for each UDC_node of the secondary-UDC_list[dir] do
      travel distance l1 along direction d1 and report error (spacing
      or width error as appropriate) if any;
end;
procedure local_DRC ;
  Previous_white:node; Previous_d,current_d:N..W;
begin
  Envelope_local_image;
  Find northernmost black node current_black;
  Identify a white node current_white adjacent to current_black;
  current_d:=E; {direction of region boundary
                 between node pair (current_black, current_white)}
  Previous_d:= current_d;
  Previous_white:=current_white;
  previous_black:= current_black;
  length_traversed := minimum (edge_length_of(current_black),
                               edge_length_of(current_white));
```

```
While not boundary of region covered do
  begin
    Find next (current_black, current_white) pair;
    if change in direction then
      begin
        If not Boundary_white(Previous_white) and
          not boundary_white(current_white) then
          begin
            if length_traversed < design rule distance
              {width check corresponding to the layer in
              consideration} then
                if FILL_Later then
                  begin
                    with the last CDC_node do
                      begin
                        ERROR_SO_FAR := true;
                        ERROR_LENGTH := length_traversed;
                      end
                    length_traversed := minimum
                      (edge_length_of(current_black),
                      edge_length_of(current_white));
                  end
                else report design rule error;
                {corner of an LG reached}
                orthogonal_&_diagonal_checks;
                current_d: = new_direction;
                length_traversed := minimum
                  (edge_length_of(current_black),
                  edge_length_of(current_white));
            end;
          end
        else
          length_traversed := length_traversed +
            minimum(edge_length_of(current_black),
            edge_length_of(current_white));
        If not Boundary_white(Previous_white) and
          boundary_white(Q) then
          begin {Start of boundary_code along
                sub-layout boundary}
            create a node for CDC_List(previous_d);
            with CDC_node do
              begin
                crosses_at_corner: = false;
                JOIN: = true;
                FORK: = false;
                pe_id: = this PE;
                if length_traversed < design rule distance then
                  begin
                    ERROR_SO_FAR := true;
                    ERROR_LENGTH := length_traversed;
                    length_traversed: =
                    minimum(edge_length_of(current_black),
                    edge_length_of(current_white));
                  end;
                distance: = distance(intersectionpoint,
```

```
                    begin_point of Boundary_of(previous_d));
                    {using procedure K_to_coord}
                  end;
            end;
      If (Boundary_white(Previous_white) and
          not Boundary_white(Q)) then
        begin {end of Boundary_code along sub-layout boundary}
          if length_traversed < design rule distance then
            report error;{of the previous edge}
          create a node for CDC_List(previous_d);
          with CDC_node do
            begin
              crosses_at_corner: = false;
              JOINT: = false;
              FORK: = true;
              pe_id: = this PE;
              FILL_LATER : = true;
              {the boolean ERROR_SO_FAR and ERROR_LENGTH
              will be filled later when the boundary encounters
              the next change of direction}
              distance: = distance(Intersectionpoint,
              begin_point of Boundary_of(previous_d);
              {using procedure K_to_coord}
            end;
        end;
    If (Boundary_white(Previous_white) and
        Boundary_white(Q)) then
        begin {boundary code meets a corner}
          let X: = previous_d;
              Y: = opposite(current_d);
          corner_status[corner_of(X,Y)]
            [corner_of(opposite(X),opposite(Y))]: = true;
        end;
        Previous_d: = current_d;
        previous_black : = current_black;
        previous_white : = current_white;
    end; {end of while }
end; { of procedure local_DRC }
    procedure process_corner_booleans;
      var corner_status : array[NE..SE] of status;
      {e.g. corner_status[NW] gives the status of the NW corner of this PE}
    begin
      for every processor do in parallel
        begin
          for dir = [N, E, S, W] do
            begin
              case dir of
                N : begin
                      send corner_status[NE] to neighbor PE
                      in the N direction;
                      correspondingly receive corner_status[NE]
                      of the neighbor PE in the S direction;
                      store the received corner_status in Ctemp;
                      OR the four bits of corner_status[SE] of
                      this PE with the four bits of Ctemp;
```

```
                    send corner_status[NW] to neighbor PE
                    in the N direction;
                    correspondingly receive corner_status[NW]
                    of the neighbor PE in the S direction;
                    store the received corner_status in Ctemp;
                    or the four bits of corner_status[SW] of
                    this PE with the four bits of Ctemp;
                  end;
                E : begin
                    send corner_status[NE] to neighbor PE
                    in the E direction;
                    correspondingly receive corner_status[NE]
                    of the neighbor PE in the E direction;
                    store the received corner_status in Ctemp;
                    OR the four bits of corner_status[NW] of
                    this PE with the four bits of Ctemp;
                    send corner_status[SE] to neighbor PE
                    in the E direction;
                    correspondingly receive corner_status[SE]
                    of the neighbor PE in the E direction;
                    store the received corner_status in Ctemp;
                    or the four bits of corner_status[SW] of
                    this PE with the four bits of Ctemp;
                  end;
                S : steps similar to the procedure above;
                W : steps similar to the procedure above;
              end;
            end;
        end;
    end; { of process_corner_booleans }
```

Below we explain the procedure Interpret_CDC_List(T1,T2) with regard to Fig. 9, wherein the sub-layout boundary between T1 and T2 are interpreted. T1 and T2 are two consecutive nodes from templist and are the intersection points of the LG boundary with the sub-layout boundary encountered during the clockwise traversal of the sub-layout boundary. Let d1 and d2 denote distance(T1,origin) and distance(T2,origin) such that $d1 \leq d2$. We merge the two lists using the distance(T, begin_point) as the first key and the boolean not(This_PE) as the second key assuming FALSE < TRUE. The interpretation corresponding to pair (T1, T2) is unique for any particular boolean assignment for the following predicates associated with T1 and T2, viz.

a). P1 := (T1.Pe_id = this_PE)
b). P2 := (T1.JOIN = TRUE)
c). P3 := (T2.Pe_id = this_PE)
d). P4 := (T2.JOIN = TRUE)

Actions for the individual cases corresponding to the boolean assignments of P1, P2, P3 and P4 are given below. The procedure below is exhuastive and may perform redundant checks also. This is mainly for sake of clarity and readability. Cases that do not perform redundant checks can be developed similarly.

```
Procedure Interpret_CDC_List (var T1,T2 : pointer to CDC_node);
Begin
  Case (P1,P2,P3,P4) of
    { all orthogonal checks along the boundary i.e. along X or Y direction are not necessary since these
    cases appear as appropriate intersection points and will be detected while checking d2-d1 distance;}
    { in all the if statements the part corresponding to then statement is performed when d1<d2 and the
    else part is performed if d1 = d2}
```

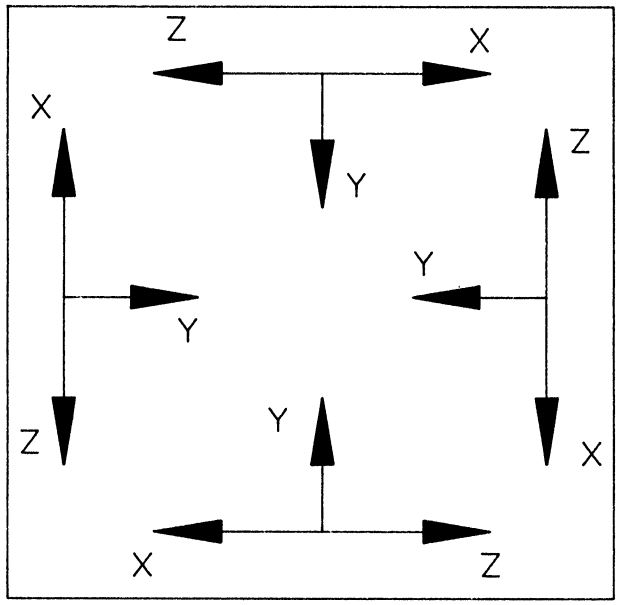


FIGURE 11 Conventions for directions X, Y, Z in procedure Interpret_CDC_List.

```
{ Use of directions X, Y, Z is with respect to Fig. 11, wherein direction X is same as that of the
direction of clockwise traversal along the sub-layout boundary, direction Z is the direction opposite
of X and Y is the direction orthogonal to X and Z and moves into the sub-layout away from the
associated boundary. e.g. for the E boundary, the direction X is S, Z is N and Y is W.}
{ in all the pairs (T1,T2) we take action corresponding to the intersection point T1 only; this is so,
since before the next iteration of the loop (that calls this procedure) we update T1, T2 by executing
T1: = T2, T2: = T2.next and the present T2 becomes T1 while the next pair is being processed and
the corresponding action is taken}
{ a check (orthogonal or diagonal) which starts at a white node checks for spacing whereas that
which starts at a black node checks for width; hence we do not specify if the check corresponds to
that of width check or spacing check}
(F,F,F,F) : if d1 < d2 then case not possible {(i)a }
           else case not possible; { CASE (i)b }
(F,F,F,T) : if d1 < d2 then {(ii)a}
          begin
            Orthogonal checks :
              In direction Y starting from point T1;
            Diagonal checks :
              In direction (X,Z) starting from T2;
          end
              else case not possible; {(ii)b}
(F,F,T,F) : if d1 < d2 then
            Begin
              if (d2-d1) < design rule then report error;
              Orthogonal checks in direction Y and Z from T1;
              Diagonal Checks in direction (X,Z) from T1;{(iii)a}
              end
            else
              begin
                report error;
                diagonal check in direction (X,Z) from T1;
              end; {(iii)b}
(F,F,T,T) : if d1 < d2 then
```

```
                Begin
                  if (d2-d1) < design rule then report error;
                  Orthogonal checks in direction Y and Z from T1;
                  Diagonal Checks in direction (Y,Z) from T1;{(iv)a}
                end
              else
                if (T1.Error_length + T2.Error_length) < design
                    rule specification then report error;{(iv)b}
(F,T,F,F) : if d1 < d2 then
                begin
                  orthogonal check in direction Y from T1;
                  diagonal checkin direction (X,Y) from T1;
                end {(v)a}
              else case not possible; {(v)b }
(F,T,F,T) : if d1 < d2 then case not possible; {(vi)a }
              else case not possible; {(vi)b}
(F,T,T,F) : if d1 < d2 then
                begin
                  if d2-d1 < design rule then report error;
                  orthogonal check in Y and Z directions from T1;
                  diagonal check in (Y,Z) direction from T1;
                end {(vii)a}
              else Enter_pe_point(T_new); { (vii)b}
(F,T,T,T) : if d1 < d2 then
                begin
                  if d2-d1 < design rule then report error;
                  orthogonal check in direction Y from T1;
                  diagonal check in direction (X,Y) from T1; {(viii)a}
                end
              else
                begin
                  report error;
                  perform diagonal check in direction (X,Y) from T1;
                end; {(vii)b}
(T,F,F,F) : if d1 < d2 then
                begin
                  if d2 - d1 < design rule then report error;
                  diagonal check in direction (Y,Z) from T1;
                end { (ix)a}
              else
                begin
                  report error;
                  diagonal check in direction (Y,Z) from T1;
                end; { (ix)b}
(T,F,F,T) : if d1 < d2 then
                begin
                  if d2 - d1 < design rule then report error;
                  if T1.Error_so_far then report error;
                  diagonal check in direction (Y,Z) from T1;
                end {(x)a}
              else
                begin
                  report error;
                  diagonal check in direction (Y,Z) from T1;
                end; { (x)b}
```

```
  (T,F,T,F) : if d1 < d2 then case not possible; { (xi)a}
              else case not possible; { (xi)b}
  (T,F,T,T) : if d1 < d2 then
                begin
                  if d2 - d1 < design rule then report error;
                  diagonal check in direction (Y,Z) from T1;
                  if T1.Error_so_far then report error;
                end { (xii)a}
              else case not possible; { (xii)b}
  (T,T,F,F) : if d1 < d2 then
                begin
                  if d2-d1 < design rule then report error;
                  if T1.Error_so_far then report error;
                  diagonal check in direction (X,Y) from T1;
                end {(xiii)a}
              else
                if (T1.Error_length + T2.Error_length) < design
                  rule specification then report error;{(xiii)b}
(T,T,F,T) : if d1 < d2 then
              begin
                if (d2-d1) < design rule then report error;
                if T1.ERROR_SO_FAR then report error;
                Diagonal Checks in direction (X,Y) from T1;
              end
            else; /* no operation */ { (xiv)b}
  T,T,T,F) : if d1 < d2 then
               begin
                 if (d2-d1) < design rule then report error;
                 if T1.ERROR_SO_FAR then report error; {(xv)a}
                 Diagonal check in (X,Y) direction from T1;
               end
             else Error condition; {(xv)b}
  (T,T,T,T) : if d1 < d2 then case not possible; {(xvi)a}
              else case not possible; { (xvi)b}
  end; {of case}
end; {end of Interprete_CDC_List }
  Begin { of Parallel_DRC }
    Partition_Layout;
    for all processors do in parallel
      begin
        perform local_DRC;
        for side:=N..W do sort(CDC_List[side]);
        for side:=N..W do sort(UDC_List[side]);
        for side:=N..W do
          begin
            process corner_booleans;
            send(CDC_list[opposite_dir[side],
                 neighbor_PE[opposite_dir(side)]);
            templist:=receive(CDC_list[opposite_dir(side)],
                 neighbor_PE[side]);
            templist:=reverse(templist);
            { replace the distance d of every node by (D-d),
              where D=distance(begin_point, end_point) of any
              Boundary }
            templist:=merge(CDC_list[side],templist);
```

```
            Let (T1,T2) be the first node pair of templist;
            process_corner_case(begin_corner(side));
            While (T1,T2) is not the last node pair of templist do
              begin
                Interpret_CDC_List(T1,T2);
                Advance T1 by one node of templist;
                Advance T2 by one node of templist;
              end;
            send(UDC_list[opposite_dir[side],
                neighbor_PE[opposite_dir(side)]);
            templist: = receive(UDC_list[opposite_dir(side)],
                neighbor_PE[side]);
            templist: = reverse(templist);
            { replace the distance d of every node by (D-d),
              where D = distance(begin_point,end_point) of any
              Boundary}
            process_UDC_List;
            send(secondary-UDC_list[opposite_dir[side],
                neighbor_PE[opposite_dir(side)]);
            templist: =
            receive(secondary-UDC_list[opposite_dir(side)],
              neighbor_PE[side]);
          templist: = reverse(templist);
          { replace the distance d of every node by (D-d),
            where D = distance(begin_point, end_point) of any
            Boundary }
          process_secondary_UDC_List;
        end;
    end;
end. { of Parallel_DRC }
```