A Systolic Design-Rule Checker

RAJIV KANE AND SARTAJ SAHNI, SENIOR MEMBER, IEEE

Abstract—**We develop a systolic design-rule checker (SDRC) for rectilinear geometries. This SDRC reports all width and spacing violations. It is expected to result in a significant speed up of the design-rule check phase of chip design.**

Keywords and Phrases: **Design-Rule Checks, feature width, spacing, rectilinear geometries, systolic systems.**

I. Introduction

RAPID ADVANCES in technology are making it possible to fabricate circuits of an ever increasing complexity. This increase in circuit complexity poses a severe challenge to the algorithms presently in use in design automation tools. One of the ways to meet the challenge is to develop new computer architectures capable of running these design automation algorithms efficiently. Another approach is to develop faster algorithms.

Several new architectures and corresponding algorithms have recently been proposed for design automation. Blank *et al.* [2] describe a bit-map processor architecture suitable for Boolean operations, wire routing using Lee's algorithm, and for some design-rule check (DRC) functions such as shrink and expand. Mudge *et al.* [7] describe a Cytocomputer architecture adapted for DRC and Lee-type wire routing. Yet another DRC architecture is described in [11]. Some other references for special purpose architectures and associated algorithms for wire routing are [3] and [8]. A parallel-processing approach for logic module placement has been developed by Ueda *et al.* [13]. Simulation has also been the focus of several new architectural studies. The most popular such development is the Yorktown Simulation Engine [10], [3], [5]. Another logic simulation machine is described by Abramovici *et al.* [1].

In this paper, we shall be concerned with the design of a systolic system for design-rule checks. Our design differs from all earlier work on special-purpose architectures for design automation in that ours is the first systolic design. Of course, systolic designs have been studied for quite some time. A valuable reference is [6]. Our systolic system for DRC's differs from earlier work on hardware assisted DRC's in that it is edge-based rather than bit-map-based. Consequently, it has the potential of being much faster and less expensive than earlier designs.

Manuscript received August 1, 1983; revised July 15, 1986. This research was supported in part by the Office of Naval Research under Contract N00014-80-C-0650 and in part by the Microelectronics and Information Sciences Center at the University of Minnesota.

R. Kane is with Daisy Systems, San Jose, CA 95127.

S. Sahni is with the University of Minnesota, Minneapolis, MN 55455.

IEEE Log Number 861127.

Specifically, our systolic design-rule checker (SDRC) checks for spacing and width errors. The design may be extended to include other design-rule checks. Our design points out the potential for systolic systems in design automation applications.

II. Polygons and Errors

In arriving at our SDRC, we made several assumptions on the nature of the polygons to be handled and also on the type of errors to be checked for. First, we assume that polygons are composed of horizontal and vertical edges only. Hence, only right-angled bends are permitted. Polygons may contain holes. These holes are also restricted to be polygons with right-angled bends. Fig. 1 shows two example polygons that satisfy these restrictions.

This restriction on the edges composing a polygon allows a compact representation of each polygon. This representation consists of the following.

1) *Polygon number:* Each polygon is assigned a unique number. Holes within a polygon are assigned the same number as the enclosing polygon.

2) *A sequence of polygon vertices:* This sequence begins at the lowermost left-hand vertex of the polygon and is obtained by traversing the polygon so that its interior lies to the left of the edge being traversed. Since all edges are either horizontal or vertical, the polygon vertices (except the first) may be described by providing a single coordinate. Thus, the polygon of Fig. 1(a) is represented as

$$\text{p, n, } x_1, y_1, x_2, y_3, x_4, y_5, x_6, y_7, x_8, y_1.$$

The first symbol p identifies this as an enclosing polygon; n is the polygon number. In case of a hole, an h is used in place of the p. Holes are traversed such that the interior is to the left of each edge traversed. The representation for the polygon and holes of Fig. 1(b) is

$$\text{p, n, } x_1, y_1, x_2, y_3, x_4, y_5, x_6, y_7, x_8, y_9, x_{10}, y_{11}, x_{12}, y_1$$

$$\text{h, n, } x_{13}, y_{13}, x_{14}, y_{15}, x_{16}, y_{17}, x_{18}, y_{19}, x_{20}, y_{13}$$

$$\text{h, n, } x_{21}, y_{21}, x_{22}, y_{23}, x_{24}, y_{25}, x_{26}, y_{21}.$$

The SDRC assumes that the polygons are well formed. Specifically, open polygons (Fig. 2(a)), polygons with shared edges (Fig. 2(b)), polygon overlaps (Fig. 2(c)), and polygons sharing an edge with a hole (Fig. 2(d)) are not permitted. While this assumption of well-formedness is not essential to our discussion, it enables us to concentrate on spacing and width issues. A minor modification to our design allows the SDRC to check for the above

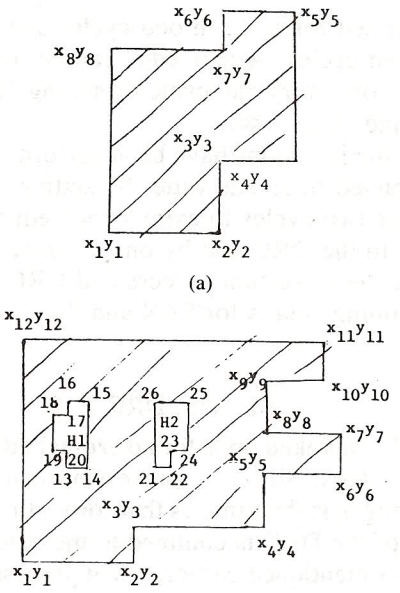


Fig. 1. Examples of polygons. (a) No holes. (b) Two holes H1 and H2. Shaded area is the interior of the polygon.

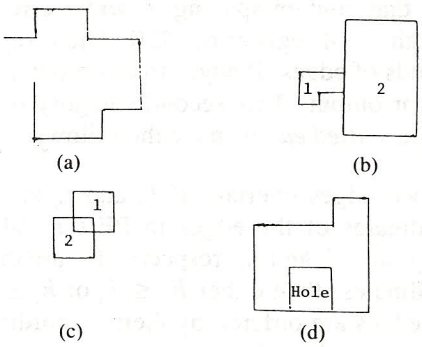


Fig. 2. Malformed polygons.

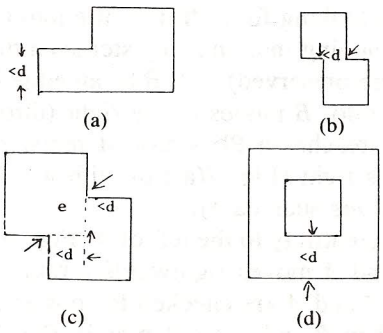


Fig. 3. Polygons with width errors.

malformations. Also, these inconsistencies need to be explicitly checked before one can apply bit-map-based width and spacing checks.

Let w denote the minimum allowable feature width. Fig. 3 gives examples of polygons with width error. Many designers do not regard Fig. 3(c) as an error unless the distance e is less than w. Our SDRC is easily changed to account for this variation. The only change needed is to compare w with e rather than d.

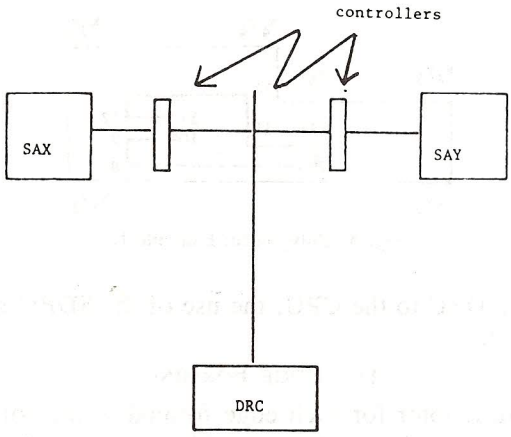


Fig. 4. SDRC architecture.

III. SDRC Architecture

The SDRC is a hardware device that may be attached to a computer system as a peripheral or directly to the CPU as in the case of a floating-point processor. A block diagram of the SDRC appears in Fig. 4. The major components of an SDRC are two systolic sort arrays (SAX and SAY), controllers for these sort arrays, and a systolic design-rule checker (DRC). Note that we use SDRC to denote the entire systolic design-rule check system of Fig. 7 and DRC to refer to a component of SDRC that performs the actual design-rule checks. This component is also systolic in nature. When design-rule checks are to be performed, the CPU sends the compact descriptions of the polygons to the SDRC. This description is transformed into explicit edges by the controllers for SAX and SAY. Horizontal edges are created by the controller for SAX and inserted into SAX. Vertical edges are formed by the controller for SAY and inserted into SAY. The sort arrays sort the edges into lexical order. Thus, the SAX sorts edges by y-coordinates and within y-coordinates by x-coordinates. Recall that we have assumed that there are no overlapping edges. So, even though every horizontal edge has two x-coordinates, there is a unique lexical ordering for the horizontal edges. Similarly, there is a unique ordering for the vertical edges.

As we shall see in the next section, the SAX and SAY are simply systolic priority queues. Consequently, as soon as the edges have been formed and entered into the SAX and SAY, they may be transmitted in lexical order to the DRC. First, SAX sends its edges to the DRC, which examines them for width violations in the y-direction and spacing violations in the x-direction. All detected errors are transmitted back to SAX. Next, SAY transmits its edges to the DRC, which examines them for width errors in the x-direction and spacing errors in the y-direction. These errors are sent back to SAY. The errors collected in SAX and SAY may then be communicated back to the CPU.

Clearly, by using two DRC's, the horizontal and vertical edge processing may be effectively overlapped. Further, by providing a data path for the errors to go directly

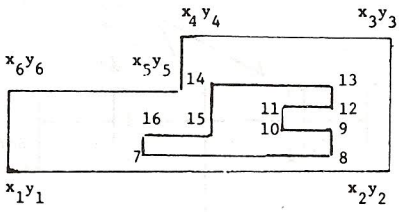


Fig. 5. Polygon for Example 1.

from the DRC to the CPU, the use of the SDRC may be pipelined.

IV. Edge Forming

The descriptor for each edge formed in the sort array controllers consists of the five fields: y, x_l, x_r, $p\#$, and ud. The terminology used is with respect to the horizontal edges. y is the y-coordinate for the edge, x_l the left x-coordinate, x_r the right coordinate, $p\#$ the polygon number, and ud (up–down) is 0 if the interior of the polygon is above this edge and 1 otherwise. In case the DRC sends errors back to the SAX (rather than directly to CPU), then each edge descriptor will have two additional bits to record the error. For vertical edges, x is the x-coordinate of the edge; y_b and y_t are, respectively, the bottom and top y-coordinates; $p\#$ is the polygon number; and lr (left–right) is 0 if the polygon interior is to the left of the edge and is 1 otherwise. The $p\#$ field is used only to identify polygons with errors. This field may be omitted and the detected errors can be associated with polygons by performing a search at the end.

Example 1: The edge descriptors for the horizontal edges of the polygon of Fig. 5 are: $y_1, x_1, x_2, 1, 0$; $y_7, x_7, x_8, 1, 1$; $y_{16}, x_{16}, x_{15}, 1, 0$; $y_{10}, x_{10}, x_9, 1, 0$; $y_{11}, x_{11}, x_{12}, 1, 0$; $y_6, x_6, x_5, 1, 1$; $y_{14}, x_{14}, x_{13}, 1, 0$; $y_4, x_4, x_3, 1, 1$.

The descriptors for the vertical edges are: $x_1, y_1, y_6, 1, 1$; $x_7, y_7, y_{16}, 1, 1$; $x_5, y_5, y_4, 1, 1$; $x_{15}, y_{15}, y_{14}, 1, 0$; $x_{10}, y_{10}, y_{11}, 1, 1$; $x_{12}, y_{12}, y_{13}, 1, 1$; $x_8, y_8, y_9, 1, 1$; $x_2, y_2, y_3, 1, 0$.

The transformation from the compact polygon representation to the edge descriptors is relatively straightforward. □

V. The Sort Arrays

While many sorting algorithms have been considered for hardware implementation [12], priority queues appear to be best suited for our sort application. Two systolic implementations of priority queues appear in the literature. One is due to Leiserson [9], and the other due to Guibas and Liang [4]. While the design of [4] is simpler than that of [9], it permits an insert/delete every four cycles as opposed to once every two cycles for the design of [9].

The systolic priority queue of [9] is a linear array of processors (PE's) each having two registers A and B. Each register in the priority queue is large enough to hold an edge descriptor. The array of processors pulsates in regular cycles with instructions:

1. $B_i \leftarrow B_{i-1}$
2. Order A_{i-1}, A_i, B_{i-1} so that $A_{i-1} \leq A_i \leq B_i$

being performed for odd i in odd cycles and for even i ($i \neq 0$) in even cycles. A new edge can be inserted in the array just before every odd cycle by setting B_0 to the edge descriptor and A_0 to $-\infty$.

When all the insertions have been performed, the edges can be extracted in lexical order by setting A_0 and B_0 to $+\infty$. It takes two cycles to extract each edge. The edges can be sent to the DRC one by one as extracted, thereby overlapping the extraction process and DRC operation.

The remaining details for SAX and SAY may be found in [9].

VI. The DRC

The DRC is invoked once for horizontal edges and once for vertical edges. Since the processing that occurs with horizontal edges is the same as that for vertical edges, our discussion of the DRC is confined to the case of horizontal edges. As mentioned earlier, when processing the horizontal edges, the DRC checks for width violations in the y-direction and spacing violations in the x-direction.

The DRC (Fig. 6) is a linear systolic array. Edges enter the DRC through the B register of PE 0. Output edges, i.e., edges that contain spacing or width errors, exit the DRC through the A register of PE 0. The A registers contain two kinds of edges: i) edges to be output and ii) edges not ready for output. This second category of A register edges, called *settled edges*, have the following properties.

(a) No two edges overlap. If L_i and L_j are the left x-coordinates of the edges in PE's i and j, respectively, and R_i and R_j, respectively, are their right x-coordinates, then either $R_j \leq L_i$ or $R_i \leq L_j$.
(b) The edges are ordered by their x-coordinates. So if $i < j$, then $R_i \leq L_j$.

The B registers contain edges that are either ready to be output or are looking for a PE to settle into (this can only be done by moving into an A register such that properties (a) and (b) are preserved). Let B be an edge looking for a PE to settle into. B moves to the right (through B registers) until it reaches a PE whose A register edge A lies entirely to its right (Fig. 7(a)) or with which it overlaps (Fig. 7(b) is *one* such case).

In case B is entirely to the left of A (Fig. 7(a)), B settles in this PE and A moves rightwards through the B registers. Edges B and A are checked for possible spacing violations. When B and A overlap as in Fig. 7(b), edge A is split into two segments a and b. Edge B is split into the two segments c and d. If the vertical distance between segments b and d is less than w, then a width error is to be reported. This error is recorded in segments a and d. Segments b and c are discarded as they are not needed to detect any further width errors in the y-direction or spacing errors in the x-direction. Segment d continues to move through the B registers. Observe that segment a isn't ready for output as it is needed to detect further width and spacing errors. Further, observe that by splitting the edges A and B and discarding segments b and c, we ensure that

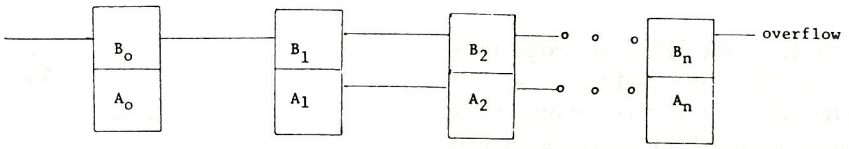


Fig. 6. DRC schematic.

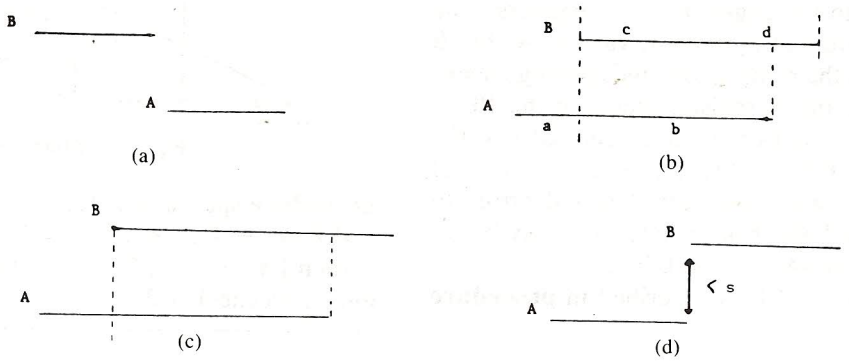


Fig. 7. Relationships between A and B edges.

when the segment d eventually settles, it does not overlap with other settled edges.

When the situation depicted in Fig. 7(c) arises, we need to check if the vertical distance between the B and A edges is at least s. If it is, then the remaining edges passing over A are too far from A to result in errors. So, edge A may be discarded if no errors have been detected so far with respect to it. If they have, then the status of this edge is changed to "ready for output." In case the vertical separation between B and A is less than s, then a vertical spacing error between B and A exists. This is recorded and the edge A is declared ready for output. Observe that, in this case, a y-direction spacing error has been detected. While we earlier stated that only x-direction spacing errors will be detected in this pass, our implementation at times records y-direction errors too. This is essential to catch the spacing error of Fig. 7(d).

With this introduction, we present the details of the DRC. We begin with the description of the registers A and B that each PE has. In describing the fields of a register, we shall use the notation $A[i].x$ to mean field x of register A of PE i. Each register in the DRC has all the fields necessary to describe an edge. In addition, the following fields are also present.

PR This is a 2-bit priority field used to control the flow of data in the A and B registers. The four possible values assignable to PR have the following interpretation:

PR = 11: This signifies an empty register. If ud = 0, then this is an empty register to the right of the rightmost edge in the DRC. If ud = 1, then this is an empty register to the left of the rightmost edge in the DRC.

PR = 10: The register contains an edge that has yet to settle in its place. This value is possible only for B register edges.

PR = 01: This value is possible only for an A register edge. It denotes an edge that has settled.

PR = 00: Denotes an edge for which an error has been detected.

we A 1-bit width error field. It is set to 1 if a width error involving this edge has been detected.

se A 1-bit error field that is set to 1 when a spacing error involving this edge is detected.

rightok A 1-bit field. This is used only for edges with ud = 0. Let $X, Y \in \{A, B\}$. $X[i]$.rightok = 1 iff there is a j such that

$(X[i].P\# = Y[j].P\#$ and $X[j].x_r = Y[j].x_l$ and $Y[j]$.ud = 0).

y_{right} Used in conjunction with rightok. Gives the y-coordinate of the edge that satisfies the condition of rightok

leftok A 1-bit field that is used only for edges with ud = 1. Let $x \in \{A, B\}$. $X[i]$.leftok = 1 iff there is a limb (i.e., an upward growth of a polygon from one of the ends of a horizontal edge with ud = 1) at the left end of the edge.

x_{ext} When leftok = 1, x_{ext} gives the leftmost point of the edge. Since edges may get split during processing, x_{ext} may not equal x_l (x_l will be the current left end of the split edge). Since the rightok and y_{right} fields are used only when ud = 0 while the leftok and x_{ext} fields are used only when ud = 1, these fields may use the same physical register space.

At the start of each cycle of the DRC, an edge is inserted in B_0. This edge has PR = 01, and we = se = 0. Since edges come from the SAX (or SAY) only once every two cycles, the cycle time of the DRC must be at least twice that of the sort arrays. Once the edge enters the DRC at B_0, it moves towards the right until it finds its correct position with respect to the edges in the A registers. The A register edges are ordered by their x_l values. As the B register edges move to the right, width and spacing checks are performed against the A register edges in the PE's. Once all the horizontal edges have been entered into the DRC, we set $B[0].\text{PR} = 11$, $B[0].\text{ud} = 1$, and $A[0].\text{PR} = 11$. This will cause the detected errors to move to the left of the DRC from where they may be removed and sent back to SAX or the CPU.

The basic cycle of the DRC is described in **procedure** cycle (Program 1).

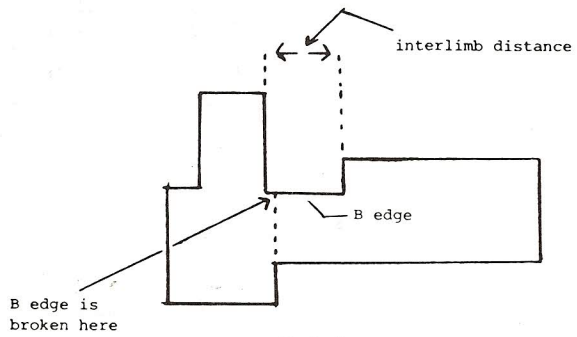


Fig. 8. Interlimb distance.

```
procedure spacecheck 1.2
  if B.x_l − A.x_r < s
  then [A.se ← 1; B.se ← 1] endif
end spacecheck 1.2
```

```
procedure cycle
  {pulsating cycle of the systolic DRC}
  repeat
    { shift B edges right }
    for every PE i, i < n do
      B[i+1] ← B[i]
    B[0] ← new edge
    B[0].leftok ← 0
    A[0].(PR, x_l, x_r, we, se, ud) ← (00, −∞, −∞, 0, 0, 1)
    PROCESS_IN_EACH_PE { described later }
    { shift A edges as needed }
    for every PE i do
      if A[i].PR = A[i+1].PR = 11 and A[i+1].ud = 0
      then { mark i as right of rightmost edge }
           A[i].ud = 0
      end
      for odd i on odd cycles and even i on even cycles do
        if A[i].PR > A[i+1].PR
        then if ((A[i].PR = 00) or (A[i].PR = B[i].PR = 11))
          then A[i] ↔ A[i+1] { interchange edges }
          endif
        endif
    end
  until false { infinite loop }
end cycle
```

A. Procedures Used for Width and Spacing Checks

Before specifying the details of the step 'PROCESS_IN_EACH_PE,' we describe a few procedures used for this purpose.

Spacecheck 1.1: This is used by a PE that contains an edge in its A register that is to the right of the edge in its B register.

```
procedure spacecheck 1.1
  if A.x_l − B.x_r < s
  then [A.se ← 1; B.se ← 1] endif
end spacecheck 1.1
```

Spacecheck 1.2: This is similar to Spacecheck 1.1 except that the B register edge is to the right of the A register edge.

Spacecheck2: This is used to check the interlimb distance in polygons (Fig. 8). As edges progress through the DRC, they may get broken. So, the edge in a register may actually be only a segment of a larger edge. The leftmost point on the original whole edge is "remembered" in the field x_{ext} which takes the place of y_{right} (x_{ext} is used when ud = 1 while y_{right} is used when ud = 0).

```
procedure spacecheck 2
  if B.x_r − B.x_ext < s
  then B.se ← 1 endif
end spacecheck 2
```

Widthcheck1: This is used when the A and B register edges in a PE belong to the same polygon, have some overlap, and A.ud = 0 and B.ud = 1.

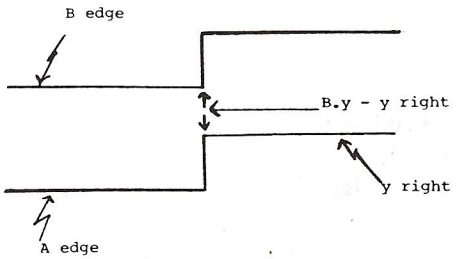


Fig. 9. A and B edges for **widthcheck2**.

procedure widthcheck1
  if $B.y - A.y < d$
  then [A.we $\leftarrow$ 1; B.we $\leftarrow$ 1] **endif**
end widthcheck 1

Widthcheck2: The widthcheck performed by this procedure is shown in Fig. 9. The PE that performs this check has edges in its A and B registers that have the same polygon numbers; A.ud = 0 and B.ud = 1; and A.rightok = 1.

procedure widthcheck2
  if $B.y - A.y_{right} < d$
  then B.we $\leftarrow$ 1 **endif**
end widthcheck2

B. PROCESS_IN_EACH_PE

In this step of the cycle, each PE examines the edges in the A and B registers and performs the checks based on this. In order to understand the edge-processing procedure to be outlined shortly, it is necessary to keep the following in mind.

1) Edges may settle only in A registers. Thus
   B.PR $\neq$ 01 for any PE.
2) Edges that have not yet settled must do so by moving to the right via B registers. So, the case A.PR = 10 is not possible.
3) Settled edges are ordered by their x values left to right in the A registers. The sequence of settled edges (i.e., PR = 01) may be interspersed with error edges (i.e., PR = 00) and empty edges (i.e., PR = 11).
4) A polygon edge may get split during processing. Fig. 10(a) shows a polygon with a hole in it. When edge e is the B edge in the PE that has the edge acd in its A register, the acd edge is split into the three segments a, c, and d. The segments a and c are discarded. In the case of the polygon in Fig. 10(b), the edge e causes the edge ac to be split into segments a and c. The segment a is discarded as no new errors with respect to this segment are possible. All errors detected for the edge are retained by the remaining segment.

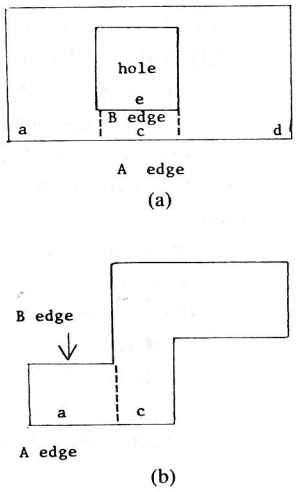


Fig. 10. Examples for edge splitting.

In general, edge splits and discards are carried out so as to ensure that the set of active edges (i.e., PR = 01 or 10) have no overlap of their x-coordinates.

The exact mechanism by which width and spacing errors are detected is best described using algorithmic notation as below. An example illustrating the working of the SDRC appears in Section IV-C.

```
procedure PROCESS_IN_EACH_PE
begin
  case A.PR of
    00: { A edge has an error; do nothing }
    10: { A edge hasn't settled. This is not possible.
          Only B edges may have PR = 10 }
    11: { A register is empty }
  case B.PR of
    00: A ↔ B { Move error edge to empty A register }
    01: { Not possible as edges can settle only in A register }
    10: if A.ud = 0
        then {No edges to the right of PE }
          [B.PR ← 0.1; A ↔ B]
        endif
        { B edge must settle here }
    11: { do nothing }
  end case { B.PR of }
    01: { A edge is in its correct place }
```

```
case B.PR of
  11: { do nothing }
  00 and 01: { not possible }
  10: case A.ud of
    0: { At this point A.PR = 01, B.PR = 10, A.ud = 0.
    The interior of the polygon is above the edge A.}
    {Determine the relationship between the A and B edges }
case
  1: A.x_l ≥ B.x_r:
    if B.ud = 0
    then {B edge is bottom edge of rectangle}
      [if B.x_r = A.x_l
       then { By assumption on the polygons B.p# = A.p# }
        [B.rightok ← 1; B.y_right ← A.y]
      else { B.P# <> A.P# or B and A are
                    from two limbs of the same polygon}
        spacecheck1.1
      endif ]
    endif
    { This is B's place to settle }
    A.PR ← 10; B.PR ← 01; A ↔ B

    { Note that when B.ud = 1, no checks need
      be performed as relevant checks were
      performed when the A edge settled }
  2: A.x_r ≤ B.x_l:
    if B.ud = 0
    then {bottom edge}
      if A.x_r = B.x_l
      then { By assumption on polygons
             B.p# = A.p# }
        [A.rightok ← 1; A.y_right ← B.y]
      else spacecheck1.2
      endif
    else {top edge}
    if A.x_r = B.x_l and not B.leftok
    then {Set leftok and x_ext in case limb test is needed.
        B edge may get split later}
      [B.leftok ← 1; B.x_ext ← B.x_l}
    endif
    if A.rightok
    then {A width check is needed. }
      [if (B.y - A.y < d) and (B.x_l - A.x_r) < d
       then B.we ← 1]
    endif
  endif
3:else: { A and B edges have some overlap and so
          must be part of the same polygon.
          Note that A.x_l ≤ B.x_l < A.x_r. The case B.x_l < A.x_l is not
          possible as this would have caused the B edge to be split
          earlier, leaving B.x_l = A.x_l }

          widthcheck1
  3.1: A.x_l = B.x_l:
    case
      3.1.1: A.x_r = B.x_r:
        if A.rightok
        then
          [widthcheck2
```

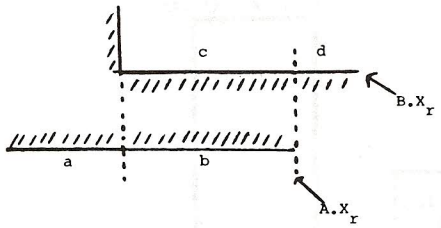


Fig. 11. Edge relationships for case 3.2.3.

```
      if B.leftok
      then spacecheck2]
      endif]
    endif
    { change status of A edge }
    if A.we or A.se
    then A.PR ← 00
    else [A.PR ← 11; A.ud ← 1]
    endif
  3.1.2: A.x_r < B.x_r:
    { split B edge and put left part in A;
      note that if there is a left limb of B,
      B.leftok and B.x_ext were set in case 2.}
    A.ud ← 1; A.y ← B.y; B.x_l ← A.x_r
  3.1.3: A.x_r > B.x_r:
    if A.rightok
    then widthcheck2
    endif
    if B.leftok
    then spacecheck2 endif
    { split A edge }
    A.x_l ← B.x_r
    { This is B's place to settle }
    A ↔ B; A.PR ← 01; B.PR ← 10
end case { 3.1 }
  3.1.2: A.x_r < B.x_r:
    { split B edge and put left part in A;
      note that if there is a left limb of B,
      B.leftok and B.x_ext were set in case 2.}
  case
    3.2.1: A.x_r = B.x_r:
      if A.rightok
      then [widthcheck2; spacecheck2] endif
      { split A edge }
      A.x_r ← B.x_l; A.rightok ← 0
    3.2.2: A.x_r > B.x_r:
      if A.rightok
      then widthcheck2 endif
      spacecheck2 { must be a limb }
      { split A edge discarding the segment A.x_l to Bx_r }
        A.x_l ← B.x_r
    3.2.3: A.x_r < B.x_r: {Fig. 11}
      { split A and B retaining segments a and d (Fig. 11)}
      A.rightok ← 0
      (A.x_l, Bx_l) ← (B.x_l, A.x_r)
  end case { 3.2 }
    end case { A.ud = 0 }
      { Begin last case to consider }
: A.ud = 1:
  { At this time, A.PR = 01, B.PR = 10, and A.ud
    = 1}
  if B.y − A.y ≥ s
  then { remaining edges are too far from A to cause
         errors}
    [if A.we or A.se
     then A.PR ← 00
     else A.PR = 11] endif
  else
  case
    :A.x_l ≥ B.x_r:
      if not [(B.ud = 1 and B.x_r = A.x_l)
        or A.x_l − B.x_r ≥ s]
      then [A.se ← 1; B.se ← 1] endif
      { This is B's place to settle }
      A ↔ B; A.PR ← 01; B. ← 10
    :A.x_r ≤ B.x_l:
      if not [B.ud = 1 and B.x_l = A.x_r
        or B.x_l − B.x_r ≥ s]
      then [A.se ← 1, B.se ← 1] endif
    :else: { Partial overlap. So, B.ud = 0}
        A.se ← 1; B.se ← 1
        case
        : B.x_r < A.x_r: {split A}
          A.x_l ← B.x_r
          A ↔ B; A.PR ← 01; B.PR ← 10
        : B.x_r ≥ A.x_r:
          A.PR ← 00
        { The remaining spacing errors involving the left
          part of the A edge will
          be detected when handling vertical edges }
        end case
      end case { else }
    endif { B.y − A.y ≥ s }
  end case { of A.ud }
end. { of PROCESS_INEACH_PE }
```

C. An Example

We illustrate the working of SDRC by means of an example. For the sake of clarity, the example is chosen to be a simple one. The polygons are transferred to the SDRC, where the SAX and SAY generate the edges. We demonstrate the working of the DRC only for horizontal edges. Thus, the input is taken from the SAX, which consists of edges in the sorted order. Fig. 12 shows the layout to be checked for width and spacing violations. The rectangle surrounding the polygons 3 and 4 is the boundary of the layout. The minimum width required w and minimum spacing required s are both equal to two units. Fig. 13 shows the edges input to the DRC, in the sorted order. For ease of understanding, we will not use x_l, x_r, y, ud, and $p\#$ to describe the edges; rather a name will be assigned to each of the edges. Fig. 13 gives the names of the edges used in the example. The only other fields shown in the registers of the PE's are PR, we, and se. During the operation of the DRC, some of the edges may get split into segments. We provide names for the segments also

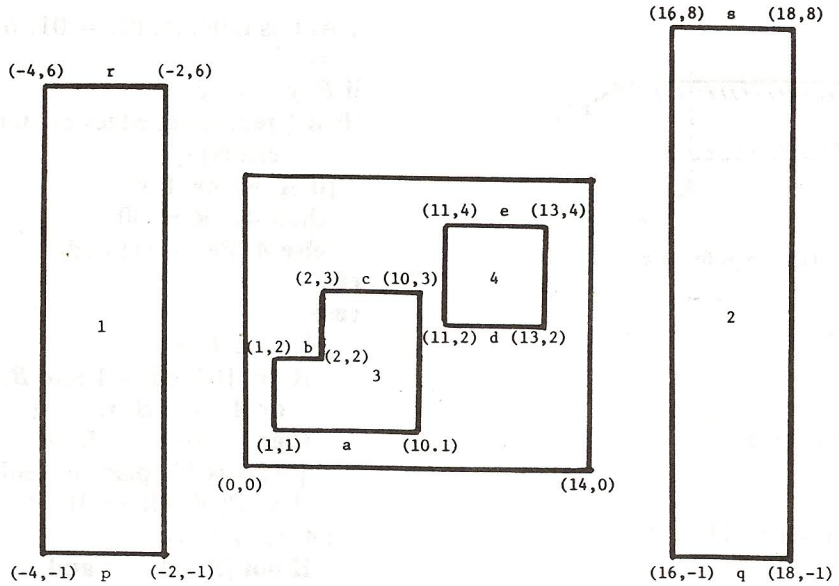


Fig. 12. Example layout.

(Fig. 13). Fig. 14 gives the status of the DRC registers just after the B registers have been transferred to the right. We now describe the processing in the PE's for the edges in the example.

cycle — *processing*

0 Initialization. All the PE's are initialized so that the PR field is set to 11 and ud to 1 (not shown in the figure).

1 Edge p is input. This will settle in PE 0.

2 In PE 0, q is to the right of p. Spacecheck1.2 is called. No error is reported.

3 In PE 0, a is to the right of p. Spacecheck1.2 is called. No errors are detected. (Here onwards we will mention Spacecheck1.1 and Spacecheck1.2 only if they detect errors.) q settles in PE 1.

4 In PE 1, a is to the left of q. a settles in PE 1 (i.e., it goes to the A register) and q moves to register B, from where it is transferred to PE 2 at the end of the cycle.

5 In PE 1, widthcheck1 is called. A width error is detected. Note that a splits. q settles in PE 2.

6 No errors are detected.

7 Note that in PE 1, $b.x_r = c.x_l$, but these belong to the same polygon. Therefore, no error is reported. In PE 2, Spacecheck1.2 is called and a spacing violation is detected. q settles in PE 3.

8 In PE 0, r overlaps p completely; no error is detected. (This is indeed a dummy polygon.) In PE 1, e is sufficiently above edge b, but b has errors associated with it. Thus, $A[1]$.PR is set to 00. We will refer to the contents of this register as error b.

8.1 This shows the status of the PE's after PROCESS_IN_EACH_PE. The A register exchanges have not yet taken place. As this is an even cycle (i.e., cycle number is even), $A[0]$ and $A[1]$ are exchanged.

9 In PE 3, a spacing error is detected. c settles in PE 3. q settles in PE 4. $A[0]$ has the error b. Also note that all the edges have been input. The error b is pulled out of PE 0, and A.PR is set to 11 in the beginning of the next cycle.

10 In PE 3, a spacing error is detected between e and c. Error f moves to PE 0.

11 In PE 2, s is sufficiently above r. Therefore, r is deleted. In PE 4, $A[4]$.PR is set to 00 to report error d. Note that this is an odd cycle; thus, c and error d are exchanged. Edge q settles in PE 5. Error f is taken out.

12 Exchange of A registers takes place in PE's 2 and 3.

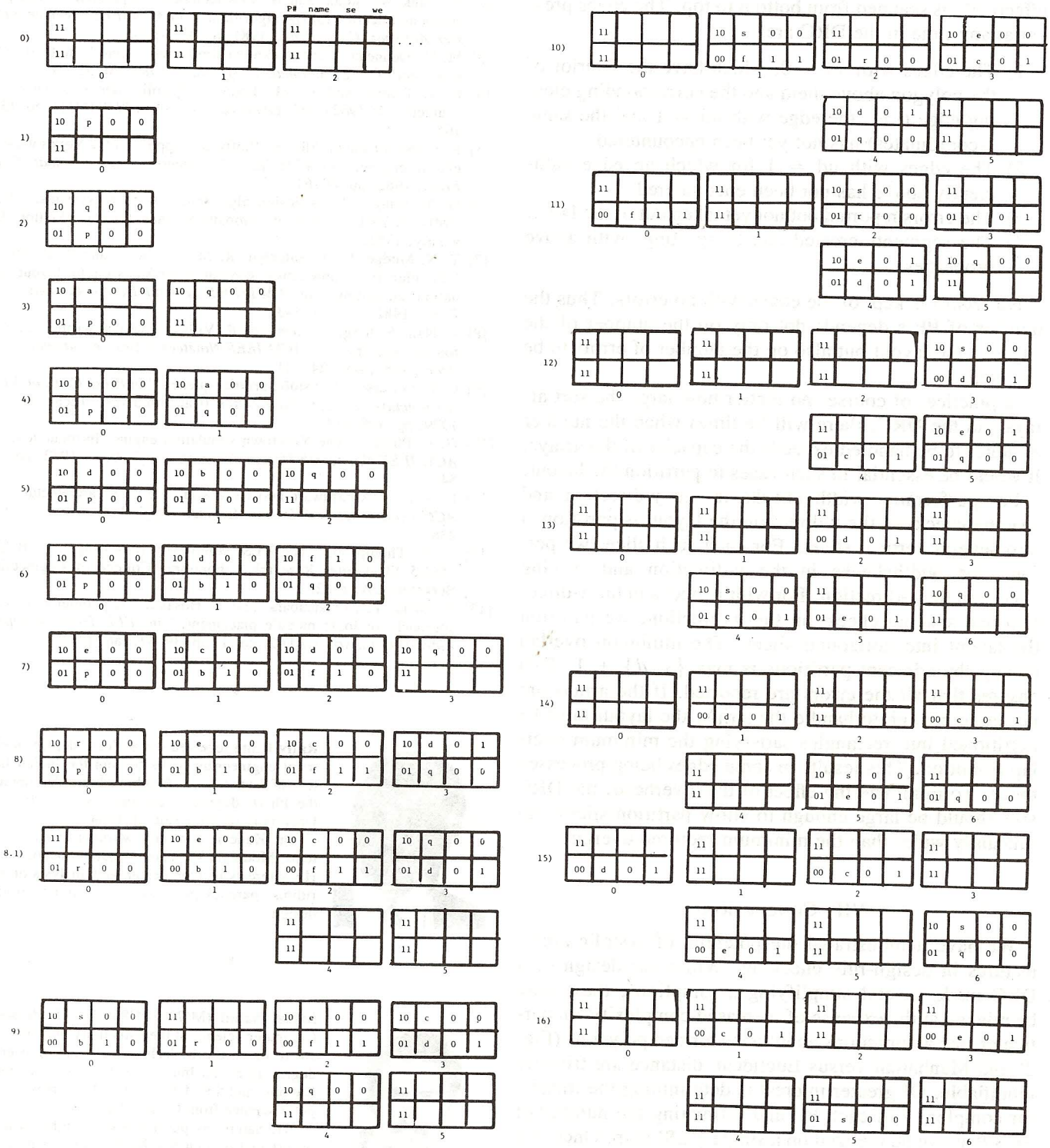
13 In PE 4, s is sufficiently above c, so it is converted to error c. Error d moves to PE 1 and error e to PE 3.

14 In PE 5, edge e is converted to error e. Error d goes to PE 0; error c to PE 2; and error e to PE 4.

15 Error d is pulled out; in PE 6, we retain only the edge s in the A register. The errors and edge s will be pulled out to the left gradually. When edge s appears in PE 0, it indicates the end of processing.

D. Performance

Under the assumption that the sort arrays and DRC are large enough to accommodate all the edges, the sort time and the DRC time is linear in the number of the edges in all the polygons. Furthermore, the time spent extracting the errors from the sort arrays is effectively overlapped with the DRC processing. The edges input to the DRC are sorted on the multikey (y, x_l, ud). Thus, the layout



name	y	x_l	x_r	ud	$p\#$
p	−1	−4	−2	0	1
q	−1	16	18	0	2
a	1	1	10	0	3
b	2	1	2	1	3
d	2	11	13	0	4
c	3	2	10	1	3
e	4	11	13	1	4
r	6	−4	−2	1	1
s	8	16	18	1	2
segment					
f	1	2	10	0	3

Fig. 13. Horizontal edges of layout of Fig. 12.

effectively is scanned from bottom to top. The edges present at any time in the DRC are

1) The edges with ud = 0, which have the interior of the polygon above them and the corresponding closing edge (i.e., the edge with ud = 1 and the same x-coordinates) has not yet been encountered.
2) The edges with ud = 1 for which an edge sufficiently above has not been encountered.
3) The errors reported but not yet taken out of the DRC.
4) The segments created due to splitting, with above properties.

No record is kept of the edges with no errors. Thus the number of PE's depends not only on the number of the edges in the layout but also on the number of errors to be reported.

In practice, of course, no matter how large the sort arrays and the DRC, there will be times when the number of edges to be handled exceeds the capacity of the arrays. It would be essential in such cases to partition the layout.

For performing width checks in the y-direction and spacing checks in the x-direction the layout is partitioned into several vertical slices. For each such slice, we perform the widthchecks in the y-direction and spacing checks in the x-direction. For width checks in the x-direction and spacing checks in the y-direction, we partition the layout into horizontal slices. The minimum overlap among the adjacent partitions is max $\{s, d\} + 1$. This ensures that all the errors are reported. If the arrays are not large enough to handle the strips, the layout must be partitioned into rectangles satisfying the minimum overlap condition. This results in some edges being processed twice. To minimize the effect of this overhead, the DRC size should be large enough to allow partition slices significantly wider than the minimum required overlap.

VII. Conclusions

We have demonstrated the potential of systolic architectures in design-rule checking. While our design of a DRC made several simplifying assumptions, these may be relaxed at the expense of increased complexity. In particular, the assumptions about well-formed polygons (Fig. 2) and Manhattan versus Euclidean distance are trivially modifiable. Of greater interest is determining the transistor complexity of each PE and estimating the number of PE's that can be realized on a single VLSI chip. Once this is known, one can estimate the speed up that one can expect using the SDRC versus a conventional computer.

References

[1] M. Abramovici, Y. H. Levendel, and P. R. Menon, "A logic simulation machine," in *ACM IEEE Nineteenth Design Automat. Conf. Proc.*, 1982, pp. 65–73.
[2] T. Blank, M. Stefik, and W. vanCleemput, "A parallel bit map processor architecture for DA algorithms," in *ACM IEEE Eighteenth Design Automat. Conf. Proc.*, 1981, pp. 837–845.
[3] M. M. Denneau, "The Yorktown simulation engine," in *ACM IEEE Nineteenth Design Automat. Conf. Proc.*, 1982, pp. 55–59.
[4] L. J. Guibas and F. M. Liang, "Systolic stacks, queues and counters," in *1982 Conf. Advanced Research in VLSI* (MIT), pp. 155–164.
[5] E. Kronstadt and G. Pfister, "Software support for the Yorktown simulation engine," in *ACM IEEE Nineteenth Design Automat. Conf. Proc.*, 1982, pp. 60–64.
[6] H. T. Kung, "Let's design algorithms for VLSI systems," Rep. CMU-CS-79-151, Dept. of Computer Science, Carnegie Mellon University, 1982.
[7] T. N. Mudge, R. A. Ratenbar, R. M. Lougheed, and D. E. Atkins, "Cellular image processing techniques for VLSI circuit layout validation and routing," in *ACM IEEE Nineteenth Design Automat. Conf. Proc.*, 1982, pp. 537–543.
[8] R. Nair, S. Jung, S. Liles, and R. Villani, "Global wiring on a wire routing machine," in *ACM IEEE Nineteenth Design Automat. Conf. Proc.*, 1982, pp. 224–231.
[9] C. E. Leiserson, "Systolic priority queues," in *Proc. Conf. on VLSI: Architecture, Design, Fabrication* (California Inst. Technol.), Jan. 1979, pp. 199–214.
[10] G. F. Pfister, "The Yorktown simulation engine, Introduction," in *ACM IEEE Nineteenth Design Automat. Conf. Proc.*, 1982, pp. 51–54.
[11] L. Seiler, "A hardware assisted design rule check architecture," in *ACM IEEE Nineteenth Design Automat. Conf. Proc.*, 1982, pp. 232–238.
[12] C. D. Thompson, "The VLSI complexity of sorting," UCB ERL M82/5 Electronics Research Laboratory, College of Engineering, Berkeley, CA, 1982.
[13] K. Ueda, T. Komatsubara, and T. Hosaka, "A parallel processing approach for logic module placement," in *IEEE Trans. Computer Aided Design*, vol. CAD-2, no. 1, pp. 39–47, Jan. 1983.

*

Rajiv Kane received the B.Tech. degree in electrical engineering in 1979 from the Indian Institute of Technology, Bombay, India. He obtained the Ph.D. degree in computer science from the University of Minnesota in 1984.

Dr. Kane is currently working at Daisy Systems, Mountain View, on layout verification tools. His interests include design and analysis of algorithms, parallel processing, and computer-aided design.

*

Sartaj Sahni (M'79–SM'86) is a Professor of Computer Science at the University of Minnesota. He received the B.Tech. (electrical engineering) degree from the Indian Institute of Technology, Kanpur, and the M.S. and Ph.D. degrees in computer science from Cornell University.

Dr. Sahni has published over eighty research papers and written five books. His research publications are on the design and analysis of efficient algorithms, parallel computing, interconnection networks, and design automation. Dr. Sahni is a coauthor of the texts *Fundamentals of Data Structures*, *Fundamentals of Data Structures in Pascal*, and *Fundamentals of Computer Algorithms*, and author of the texts *Concepts in Discrete Mathematics* and *Software Development in Pascal*. Dr. Sahni is the area editor for Parallel Algorithms and Data Structures for the *Journal of Parallel and Distributed Computing*.