MASTER

On line design rule checker

Orbons, L.

Award date:
1984

5814

On Line Design Rule Checker

by L.Orbons
research group ES
Eindhoven University of Technology
10 - 7 - 1984

Supervisors:

Prof. dr.-ing. J.A.G. Jess
Ir. M. v.d. Woude
Ir. A. Bidlot

CONTENTS

SUMMARY 1

CONTENTS 2

1 INTRODUCTION..................................4

1.1 The design cycle........................4

1.2 The environment.........................6

2 DRC BASED ON DEVICE RECOGNITION..............9

2.1 Introduction............................9

2.2 The layout defined.....................11

2.3 Design rules...........................14

3 DRC SYSTEM OVERVIEW.........................19

3.1 The on-line approach...................19

3.2 Device checking........................24

4 CONTOUR ANALYSIS............................25

4.1 Minimum width checking...............25

4.2 Width check algorithm................29

4.3 Minimum spacing checking.............32

4.4 Spacing check algorithm..............33

5 PROGRAM DESCRIPTION.........................34

5.1 The database.........................34

5.1.1 Rectangle storage................34

5.1.2 Bin structure....................35

5.2 Operations on the database...........37

5.2.1 Insert operation.................37

5.2.2 Delete operation.................39

5.2.3 Select operation.................40

5.3 The Scanline Algorithm...............43

5.3.1 Insert operation.................45

5.3.2 Delete operation.................50

5.4 The check routines...................52

5.4.1 Width check......................52

5.4.2 Spacing check....................53

5.5 Commands..............................55
5.6 Communication with layout editor.....57
5.7 Design rules.........................58
6 CONCLUSIONS AND FUTURE DEVELOPMENTS.........60
7 REFERENCES..................................61

1 INTRODUCTION

In this report an on-line design rule checker for width and spacing checks on contours is described.
This design rule checker is intended for the interactive layout editors developed at our research group.

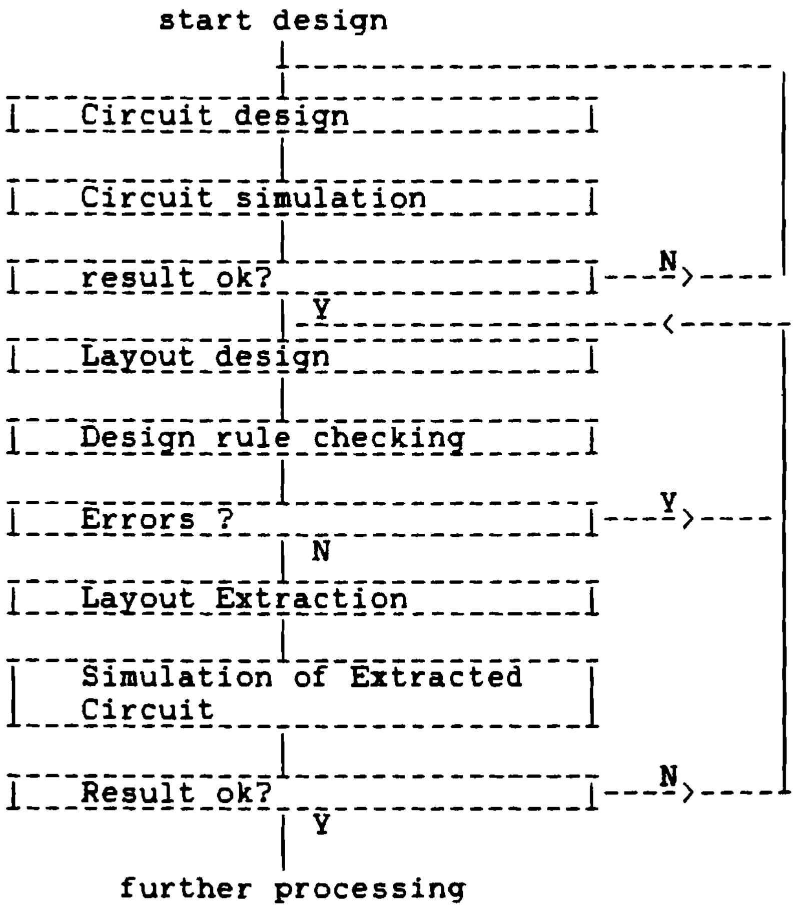
1.1 the design cycle

In this chapter a part of a possible design cycle of a (V)LSI chip will be described briefly. In this context the role of the design rule checker and the extractor will be discussed.

The design of a (V)LSI chip can be split in several stages. First of all a circuit description is made. Next the circuit is analysed using a circuit simulator. If necessary corrections are made in the circuit description, until the results of the simulation are satisfactory.
After these steps we may commence with the layout design. After finishing the layout, a design rule checker will inspect the layout for possible design rule violations. As soon as the design rule checker doesn't report any violations anymore the layout representation is correct. Unfortunately correct means only correct for design-rule violations. It is quite likely that due to parasitic effects, wrong dimensioned transistors or even completely forgotten parts of the circuit the behaviour of the realised circuit is somewhat different as desired.
Therefore it is necessary to have an extractor which generates a circuit description from the layout. This

extracted circuit description is then simulated. The results of the extracted circuit simulation can be compared with the results of the original circuit.
The proces described above is repeated until the behaviour of the extracted circuit is satisfactory.
In figure 1 an overview of this proces is given. For convenience the steps that lead to the circuit description and the steps after the construction of the layout are left out.
This report will deal mainly with the last section in this part of the design cycle, the design rule checker and the extractor.

fig. 1 overview of the design cycle



Most design-rule-checkers and extractors in use today process the layout, or parts thereof if a hierarchical method is used, after it has been completed.
This means that the correction of errors in the layout may cause several significant changes in the layout and also that we have to go through the whole process of design rule checking, extraction and simulation once again.
Since the layout generation is often an interactive process, it would be desirable that design rule checking and extraction would also work on an on-line basis.
The major advantage of this approach is that the designer immediately gets feedback on errors he made, which may save him a lot of time.
Since many basic operations in the design-rule checker and the extractor are the same (in both cases the devices in the layout must be analysed) it is attractive to combine both steps.
In the following the attention will be focussed mainly on the on-line design rule checking.

1.2 The environment

Our group has two hierarchical interactive layout editors available. One symbolic and one geometric layout editor called ISLE (Interactive Symbolic Layout Editor) and CM (Colour Mask). The next step in the development of a layout design system is the development of an on-line design rule checker and extractor.
The design-rule checker and extractor are intended as programs which run parallel to the layout editors. These programs should be able to communicate with ISLE and CM. Since both editors use the same database, this shouldn't cause severe problems.

For a usefull implementation of a design rule checker or extractor it is necessary that certain preconditions are fulfilled.
For the design rule checker this means the following. First of all the set of design rules should be consistent. Next the design rule checker has to be incremental. So only the changes in the layout are evaluated.
A special problem is that, in the process of creating a device, there may temporarily exist design rule violations in the layout. The desing rule checker shouldn't report this kind of errors.
Of course the design rule checker should be able to remember where what kind of errors occurred.
If the connectivity data is available, it becomes attractive to check the connectivity also on an on-line basis.

On line extraction of a layout also gives rise to several typical problems. First the extractor has to recognise devices, lookup the model belonging to that device and calculate the parameters of the model. Next this model is added to the extracted circuit description.
The accuracy of the extractor is another point of interest. The more accurate the description is, the more time is spent in calculating the model parameters and the simulation after completing the circuit description will also take much more time.
Special care is needed for the actions that have to be taken if a device or a part of a devices is deleted. This goes as well for the extractor and design rule checker.
Another common problem is the time needed for the checks. Both the extractor and design rule checker should be fast and may not delay the designer significantly.

The hierarchy in the design has not been mentioned before but it is quite obvious that a hierarchical approach is necessary for succesfull on-line design rule checking and extraction.

2 DRC BASED ON DEVICE RECOGNITION

2.1 introduction

The architecture of a design rule checker depends heavily on the way the design rules are described.
Usually design rules are defined in terms of overlap of and spacing between masks and also by a specification of devices, using the same kind of description.
In such cases design rule checking consists of applying all these rules one by one on the whole layout, making it necessary to scan the layout several times.
Due to time considerations, this is not acceptable for an on-line design rule checker. A different approach is necessary.

In this new approach each legal device should be uniquely specified by describing the relations between the masks present in the device. In the following sections some of these relations will be described.
When the design rule checker encounters a part of a layout, it first locates the devices present in that part. The next step then consists of checking the devices seperately. These checks are performed by comparing the device specification with the actual situation in the layout.
The specification of design rules will be much more elaborate then before, since all legal devices must be specified.

If a design rule checker is able to recognise devices, the step to extraction is not so large anymore. Actually the design rule checker performs the first step in an extraction

process, the location and recognition of devices.
Another important aspect is that of technology independence. If our design rule checker is, to a certain extent, technology independent, it should be possible to adapt the design rule checker to a change in the technology easily.
In the following paragraphs the subjects mentioned here will be treated in some more detail.

2.2 The layout defined

In this section a general definition of a layout will be given. We will start with giving definitions for the basic elements in a layout.

window:

a window is a rectangle. A window is characterised by its position and its dimensions. A window can e.g. be the surrounding box of a compound

Next we define a layer or a mask.

layer :

a region having the same dimensions and position as the window. A unique number and/or name is assigned to a layer.

Thus every window can be accompanied by several different layers.

contour:

a polygon consisting of one or more connected paraxial rectangles in one particular layer. Two rectangles are connected if they have at least one point in common. A contour is a connected region, which in general may contain holes. A contour usually doesn't contain all rectangles of a layer in the window.

constraints:

two layers have constraints with each other if there are restrictions concerning overlap and/or spacing between them. In N-MOS technology e.g. metal and polysilicon have no constraints with each other.

common area:

A common area is the intersection of two overlapping polygons A common area is a connected region.

maxset :

A maxset is a set of one or more contours, sharing a common area. All contours in a maxset belong to different layers. In the common area the number of overlapping or touching contours reaches a local maximum.
Each contour in a maxset has constraints with at least one other contour in the maxset.
If the common area is formed by say the contours A B and C then only ABC forms a maxset, AB, AC or BC are, in this case, no maxsets.

From the definition of a maxset it follows that a maxset may contain one or more common areas.

basic maxset:

A basic maxset is a maxset which contains exactly one common area.

device kernel :

A device kernel is a basic maxset, the common area is characterised by a number of overlapping or touching layers. In the common area the layer density reaches a (local) maximum.
In the following we will restrict ourselves to rectangular device kernels only.

device perifery :

The device perifery consists of all rectangles touching or overlapping the device kernel.

Provisionally we define a device as follows,

device :

A device consists of one device kernel and a device perifery.
Note : not all possible kernels form a device.

Now we can define a layout as follows

layout:

a set of devices in a window

2.3 Design rules

We consider the following design rules:

minimum width rules:

each contour should satisfy a minimum width criterion. Consider a pair of points (P1,P2) on the border of a contour, where P1 is a cornerpoint and P2 is an arbitrary other point of the border.
In this case the border of a contour doesn't belong to the interior of the contour.
There occurs a width violation in the contour if there exists such a pair (P1,P2) for which the following statement holds:

> The shortest path between P1 and P2 lies completely in the interior of the contour and the length of that path is less then a minimum length d.

The minimum width d depends on the layer in which the contour is situated.

minimum spacing rules:

each single contour and each pair of contours must satisfy a minimum spacing criterion.
Consider a pair of points (K1,K2) where K1 is a corner in contour 1 and K2 is an arbitrary point on the border of contour 2, or contour 1 if only one contour is evaluated.
Here the border of the contour belongs to the interior of the contour.
There occurs a spacing violation between two contours if

there exists a pair (K1,K2) for which the following statement holds:

> The shortest path between K1 and K2 lies completely on the exterior of both contours and has a length which is less then a minimum lenght s.

If K1 and K2 belong to the same contour, there occurs a spacing violation if the shortest path between K1 and K2 lies completely on the exterior of the contour and has a length less then or equal to the minimum length s.
The minimum spacing s depends on the layers involved.
Touching or overlapping is not considered as a spacing violation.
Note : spacing rules are applied between at most two contours. This implies that the occurrence of a third contour in the neighbourhood or perhaps overlapping the two others, should not influence the required spacing between the previous two.

When checking a device for correctness, we consider two kind of area's which have to be analysed.

1 overlap area's
2 common area's

Where the overlap area of contour A on contour B is the area covered by contour A with the restriction that the common area('s) of both contours don't belong to the overlap area.
The overlap area of contour B on contour A is the area covered by contour B with the restriction that the common area('s) of both contour don't belong to the overlap area.

Now the following select operations can be defined:

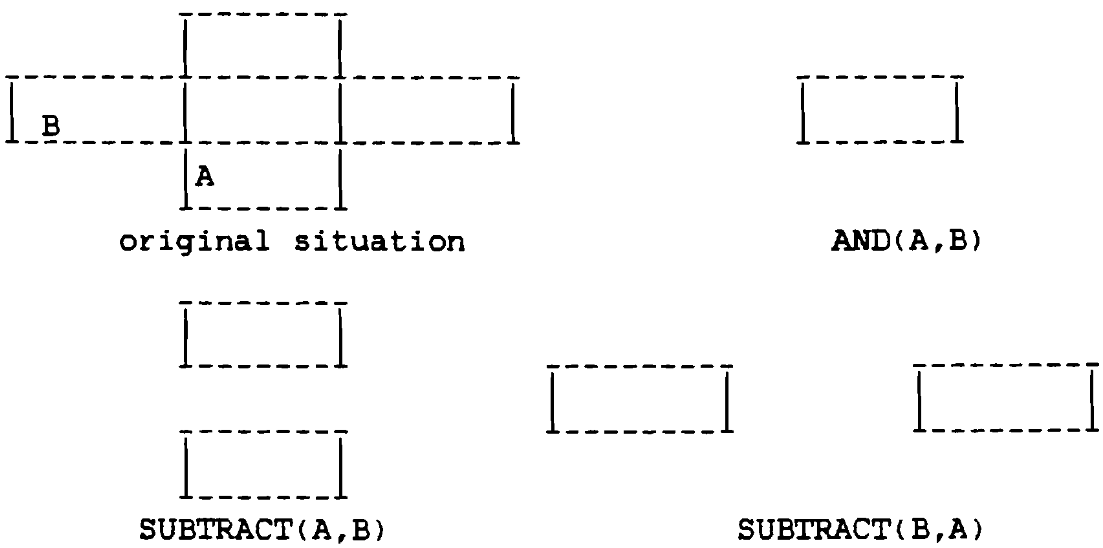
boolean AND operation

The boolean AND operation is applied on two contours. AND(contour A,contour B) yields the common area's of contour A and B.

the SUBTRACT operation

The SUBTRACT operation is also applied to two contours. SUBTRACT(contour A,contour B) yields the overlap area of contour A on contour B (i.e. the parts of A not covered by B).

fig 2 example



original situation

AND(A,B)

SUBTRACT(A,B)

SUBTRACT(B,A)

Now we define:

Select operations:

The boolean AND and the SUBTRACT operation on contours.

Device region:

A device region is a set of polygons, which is created by applying one of the select operations on two or more contours belonging to a device.

Each device region must satisfy certain conditions. In our layout we consider the following region conditions:

1 minimum dimension conditions,e.g. minimum width and minimum spacing
2 shape conditions, e.g. the shape of a region must be a rectangle
3 conditions concerning the number of unconnected polygons in a region (the cardinality of a region)

Now we define:

SHAPE operation:

The SHAPE operation checks whether the shape of a contour agrees with the defined shape type. e.g. SHAPE (contour) = rectangle;

CARDINALITY operation:

The CARDINALITY operation calculates the number of unconnected polygons in a device region. e.g. in fig. 2 the cardinality of the region created by SUBTRACT(A,B) is 2.

Now we update the definition of a device as follows:

device:

A device consists of a device kernel and a device perifery. A device can be specified by describing some of the device regions and the conditions that must be fulfilled in these regions.

The result of the previous definitions is that there are now two independant classes of design rules.
The first class of design rules, the minimum spacing and minimum width rules are context independent and are applied to single contours only.
The second class contains design rules which are related to the legal devices and are therefore context dependent.
Since all contours satisfy the appropriate minimum spacing criterion, all devices sastisfy this criterion.
Therefore eventual design rule errors can then only occur in the devices themselves.

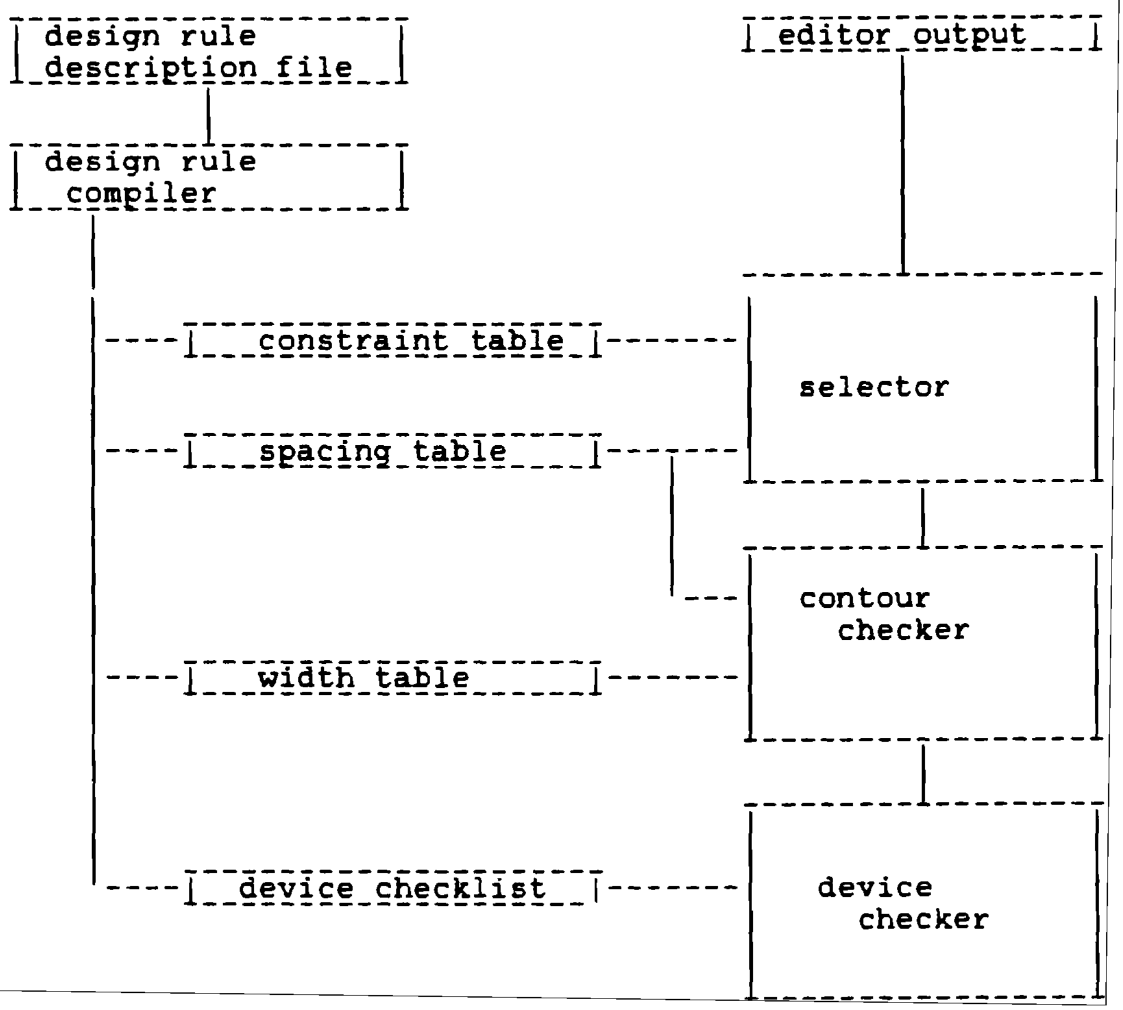
3 DRC SYSTEM OVERVIEW

3.1 The on line approach

The design rule checker which will be described here processes rectangles. These rectangles are orthogonall with respect to the x and y axis.
After inserting or deleting a rectangle we have to inspect the neighbourhood of the rectangle and select rectangles which are close to the inserted or deleted rectangle.
Rectangles which overlap or touch the inserted or deleted rectangle are always selected.
If the distance between the inserted or deleted rectangle and another rectangle is less then the minimum spacing allowed between the layers in which they are situated then that rectangle is also selected.
When design rule violations are detected the program must issue appropriate error messages and also keep an administration of the errors found in the layout.
When looking at the design rule checker part in fig. 3 we see that it consists of three parts.
The first part, the selector, selects a group of rectangles which overlap, touch or lie within the minimum spacing area of the inserted or deleted rectangle.
Next the selector groups the rectangles in contours. On their turn the contours are assigned to basic maxsets of overlapping contours.
After the selection part, the contours are passed to the

contourchecker where the following checks are performed. First each contour is checked for minimal width violations. After that, the contours are checked for minimum clearance violations.
Finally we will check the basic maxsets.
The first thing that is done by the device checker, is analysing which masks are present in the maxset. When this combination correponds with a legal device, several operations are carried out on the constituing contours to determine whether there are design rule violations in the maxset.
When looking at fig. 3 we see that the devices are specified in the design rule description file.
The design rule compiler processes this file and generates several tables which serve as reference for the actual design rule checker.

fig. 3 drc system configuration



In pseudo pascal the proces looks as follows:

```
program design rule checker {rectangle,mode}

begin

  if mode = insert
  then insert rectangle in datastructure;

  select group of rectangles within minimum
   distance of the inserted or deleted rectangle;
  assign({group}--> {contours,maxsets})

  if mode = delete then
  begin
    for each maxset do
      if maxset in errorlist(s) then delete maxset from
        errorlist(s);
    delete rectangle from datastructure;
    delete rectangle from group;
    delete contours;
    delete maxsets;
    partition({group}--> {contours,maxsets});
  end;
  checkcontours(rectangle,mode);

  if no error detected then
    checkdevice(contours,maxsets);
end {on line design rule checker};
```

```
procedure checkcontours( inrectangle, mode);

begin
  rectanglelist:=list of all rectangles
    which overlap the inrectangle;
  if mode=insert then
  begin
    assign {rectanglelist} ---> {contours};
    layer:= layerno of inrectangle;

    for each contour in layer do
    begin
      check minwidth of contour;
      for mask:=1 to nmask do
        if (layer and mask have constraints)
        and (a contour2 with layerno=mask is present)
        and (minimumspacing > 0)
        then check minimum spacing between contours;
  end
  else   {mode=delete}
  begin
    while rectanglelist <> empty do
    begin
      take nextrectangle from list;
      checkcontours(nextrectangle,insert);
    end;
  end;
end {checkcontours};
```

3.2 Device checking

As pointed out before, we can describe devices by using the WIDTH, SPACING, AND, CARDINALITY, SUBTRACT and SHAPE operations.
Another important property of a device is that it consists of a combination of different layers or masks.

Thus it must be feasible to describe each device by the occurrence of a number of masks, a number of instructions, and a number of conditions which must be fulfilled after executing the instructions.

The framework of a program that can carry out these tasks could be:

```
procedure checkdevice(contours,maxset);
begin
  if maxset on legal device list
  then
  begin
    carry out instructions;
    if unfulfilled conditions
    then
    begin
      put maxset on violation list;
      issue error message;
    end;
  end;
end {checkdevice};
```

4 CONTOUR ANALYSIS

4.1 Minimum width checking

Each contour must satisfy a minimum width criterion. A contour is a polygon consisting of paraxial rectangles. These rectangles satisfy the minimum width criterion.

When analyzing a polygon we will only consider its border. The border of a polygon is described by a number of edges.

Since the polygon consists of only paraxial rectangles, we can describe the polygon by its vertical edges. There are two kinds of edges. The in-edges and the out-edges. At an in-edge we find the interior of a polygon on the right side of the edge, and at an out-edge the interior lies on the left side of the edge. (see fig. 4.)

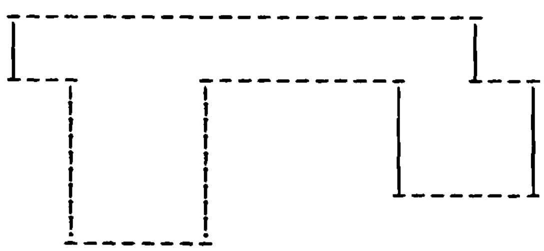


fig 4a. polygon representation

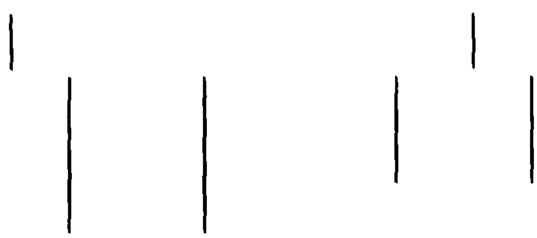


fig 4b. edge representation

In the following we assume that all contours consist of paraxial rectangles which satisfy the minimum width criterion.

From the definition of the minimum width criterion it follows:

1 An in-edge of a polygon satisfies the minimum width criterion if the interior of the polygon reaches to at least a distance d on the right side of the in-edge.

2 An out-edge of a polygon satisfies the minimum width criterion if the interior of the polygon reaches to at least a distance d on the left side of the out-edge.

3 A convex corner satisfies the minimum width criterion if we can place a rectangle with minimal dimensions in the corner in such a way that the rectangle is covered completely by the interior of the contour

4 A concave corner satisfies the minimum width criterion if we can place a rectangle with minimal dimensions in that corner in such a way that the rectangle is always completely covered by the interior of the contour.

5 A contour satisfies the minimum width criterion if all concave corners satisfy the minimum width criterion.

Although only pairs of concave corners may give rise to violations, not all pairs of concave corners can have width violations, therefore we will make a distinction between the corners and their combinations

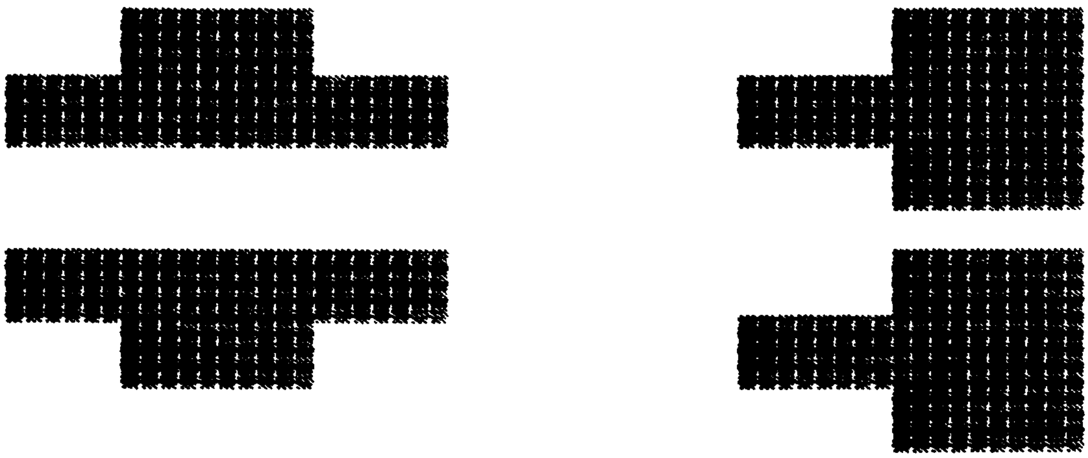
Definitions

1 A concave in-corner is a concave corner which occurs at an in-edge

2 A concave out-corner is a concave corner which occurs at an out-edge.

Since all contours consist of rectangles which satisfy the minimum width criterion there cannot occur width violations between concave in-corners, as well as between concave out-corners.

fig.5a configurations where no width violation occurs



. 5b configurations with concave in,- and concave-out corners with potential width violations.

From this the following theorem follows:

Theorem

A contour satisfies the minimum width criterion if all concave corners occurring at out-edges are at least a distance d separated from the nearest concave in-corner on the left side of the out-corner.

4.2 The width check algorithm

The strategy used to find the minimum width violations is as follows.
We will sort the vertical edges of the rectangles according to their x-coordinates in non-descending order. When two edges have the same x-coordinate, the edges are sorted on their ymin coordinate, also in non-descending order.
In-edges with the same x-coordinate as an out-edge are placed before the out-edge.
We will examine the vertical edges of the contour, by using a scanline algorithm. If we detect a concave corner at an out-edge, the contour will be examined, at that point, for width violations.
The scanline is swept accross the contour from left to right. At each moment a scanline element list is maintained. It describes a cross section of the contour at a certain x-position. Each time an edge of a rectangle is encountered, the scanline is updated.
A scanline is an ordered list of scanline elements see fig.6. A scanline element represents a rectangular slice of the contour. Each scanline element contains information of the border of the contour. A scanline element has an origin, which gives the x coordinate where the element was created and the slice begins. Further a scanline element contains ymin and ymax coordinates, which give the range in which the element is situated. Finally a scanline element contains a density field, the density gives the number of overlapping rectangles between ymin and ymax at the present location of the scanline.
In the scanline, the scanline elements are ordered according to their ymin value. Scanline elements do not overlap each

other, but touching is allowed.

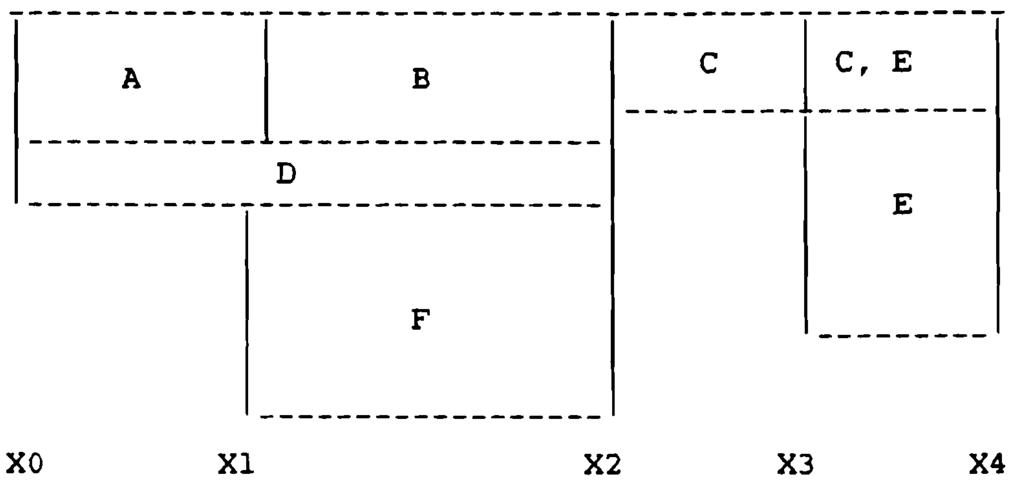


fig. 6.a contour split in rectangles

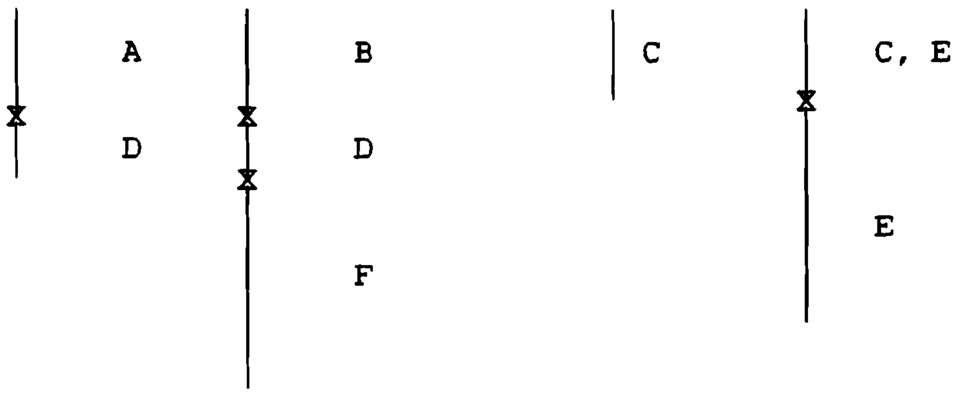


fig. 6.b. scanning the rectangles in the contour

In pseudo pascal this algorithm works as follows:

```
procedure widthchecker {inputlist};
{inputlist is a list of vertical edges,
 lexically sorted according to x, in/out,y}
begin
  while edges on input list do
  begin
    take edge from inputlist;
    update scanline elementlist;
    {a scanline element consists of xorigin,
     ymin,ymax, density; the scanline element
     list is sorted according to ymin}
    while scanline elements with density=0 do
    begin
      delete scanline element;
      if concave corners detected then
      {a concave corner occurs if ymax/ymin of the
       deleted scanl. elem.= ymin/ymax of
       the next/previous scanl.el.}
      begin
        if concave corner at in-edge within
           minimum width
        then report width violation;
      end;
    end;
  end;
end {widthchecker};
```

4.3 Minimum spacing checking

Minimum spacing checks have to be performed each time a rectangle is inserted or deleted.
We have to check for spacing violations in the contour to which the new rectangle belongs and also between the other selected contours and the contour containing the new rectangle.
In solving this problem we can use a similar scanline algorithm as for the width checker.
Instead of analysing the scanline when deleting scanline elements, we will now analyse the scanline when inserting and deleting scanline elements. This check consists of "walking" along the scanline and checking the distance between two scanline elements. If a scanline element is deleted, we also have to look forward in the x-direction for possible spacing violations.
A violation occurs in a situation in which the distance between the two different contours is less then a predefined minimum. Note that overlap is not a spacing violation, because overlapping contours may form devices and the device checking is done at a later stage by the device checker.
The vertical edges of the rectangles, ordered in the same way as in the widthchecker, are analysed. When a scanline element is deleted, we look ahead in the edge list, and check for edges which ly to close to the deleted scanline element.

4.3 Spacing check algorithm

In pseudo pascal this looks as follows:

```
procedure spacingchecker{inputlist};
begin
  while edges on inputlist do
  begin
    take edge from inputlist;
    update scanline;
    if (a new scanline element is created)
       or  (a scanline element is deleted)
    then
    begin
      check spacing in scanline;
      if    spacing violation
      then  report violation;
    end;
    if a scanline element is deleted then
    begin
      look ahead in edgelist for spacing violation;
      if    spacing violation
      then  report violation;
    end;
  end;
end {spacingchecker};
```

5 PROGRAM DESCRIPTION

5.1 The database

5.1.1 Rectangle storage

The database of the design rule checker consists of a rectangle list and a bin structure. For the rectangle list the following format is used:

```
Type  Quartet   = array[1..4] of integer;
      linkrect  = ^surrect;
      surrect   = record
                    id         :integer;
                    layer      :integer;
                    contourno  :integer;
                    corners    :quartet;
                    next       :linkrect;
                  end;
var   startrect :linkrect;
```

Here id is a number which uniquely identifies a particular rectangle, layer contains the number of the layer in which the rectangle is situated, contourno contains an optional contourno and the array corners contains information about the position of the rectangle.
The coordinates are stored in the following way:
Corners[1] contains xmin, Corners[2] contains xmax, Corners[3] contains ymin and Corners[4] contains ymax.
The rectangle list is not ordered. The pointer to the first record in the rectangle list is stored in startrect.

5.1.2 The bin structure

A bin structure is created by dividing the x-axis of the layout in a number of intervals or bins.
In this case for each bin an administration is kept of the rectangles which lie within that bin. In the bin structure used here a rectangle is represented by two vertical edges, an in,- and an out edge respectively.
For each bin this administration consists of an ordered list of in-edges of rectangles which cross the bin and an ordered list of in and out-edges of the rectangles which start or end in the bin.

```
const  nbins    = 25;

type   linkbin =^binel;

       binel    = record
         in       :boolean;
         idpoint :linkrect;
         next     :linkbin;
       end;

       bins     = array[1..nbins] of linkbin;
var    transbins, deltabins :bins;
```

Here nbins stands for the number of bins, binel stands for bin-element.
The in-field in the bin-element, when true, indicates that we are dealing with an in or out edge of a rectangle.
Idpoint is the pointer to the rectangle in the rectangle

list.
In each bin we have two linked lists of binel: the transbin list and the deltabin list.
The transbinlist is an ordered list of bin-elements, representing the edges of rectangles which cross the bin.
The deltabin-list is a similar ordered list, but now the edges of the rectangles which start or end in a bin are stored.
Both list contain bin-elements, or better edges, ordered in descending order on their x-coordinate.
When two edges have the same x-coordinate, the out-edge, if present, is placed before the in-edge. See fig. 7.

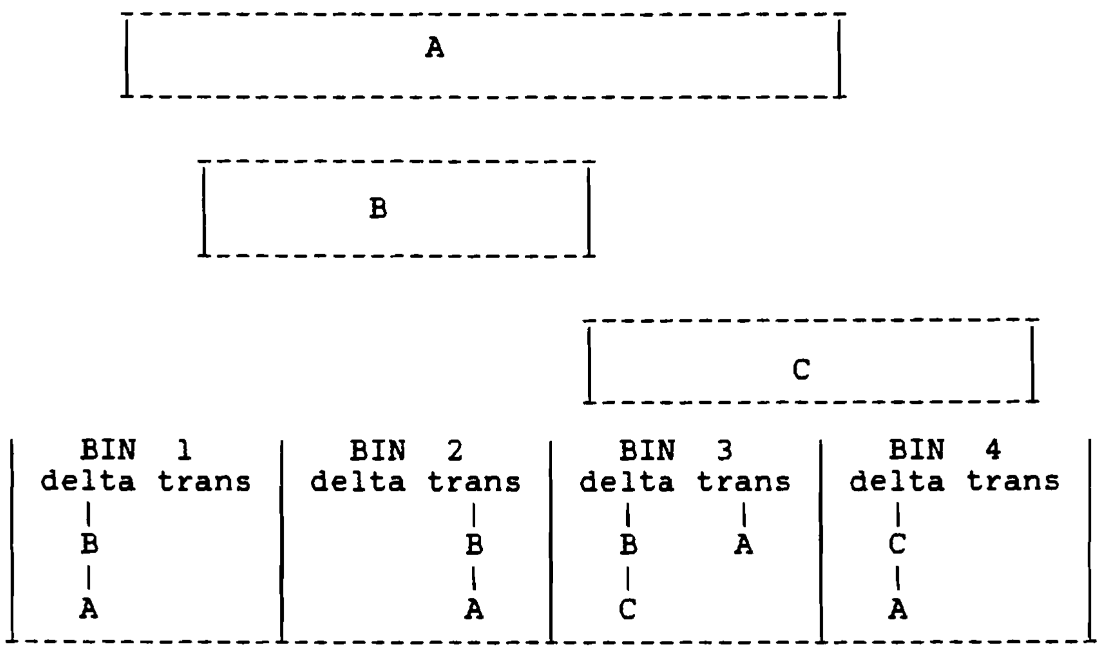


fig. 7 bin structure

5.2 Operations on the database

We consider three basic operations on the binstructure.

1 The INSERT operation
2 The DELETE operation
3 The SELECT operation

5.2.1 The insert operation

The insert operation is used to insert a rectangle in the datastructure. The following actions are taken.

1 The rectangle is inserted in the rectangle list
2 The bins in which the rectangle will be stored are calculated
3 The vertical edges of the rectangle are stored in the bin structure.

The insert operation is performed by

```
Procedure Insertrect(id,maskno  :integer; xycoor :quartet);
```

This routine calls a number of other routines which do the actual work. Here id is the identification number of the inserted rectangle.

Action 1 is performed by

```
Procedure Placerect( ids, maskno, freefield :integer;
                     xycoor :quartet;
                     var rectlistpoint, idsurrect
                     :linkrect);
```

Here rectlistpoint is the pointer to the first record of the list in which the rectangle is inserted. Idsurrect is the pointer to the record where the information of the inserted rectangle will be stored.

Action 2 is performed by

```
Procedure Binrange ( var xycoor:quartet;
                     var minbin,maxbin, maskrect:integer;
                     option: rangemode);
```

After the execution of this routine, the firstbin where the rectangle falls into is returned in minbin and the last in maxbin.
Option can be normalrange or extendrange.
In the latter case the rectangle from which the binrange is calculated is enlarged by the maximum clearance of the layer with number maskrect.

Action 3 is performed by:

```
Procedure Inclbin( var binar:bins; binnum:integer;
                   idsurrect:linkrect;
                   in:boolean);
```

Here binnum gives the number of the bin in which the edge is placed. Up indicates whether we are dealing with an in (left) or out (right) edge. Idsurrect points to the record in which the information of the rectangle is stored. Binar may be any variable of type bins, in this program TRANSBINS or DELTABINS is used.
Procedure Inclbin (include in bins) inserts an edge of a rectangle in the bin with number binnum. This procedure is

also responsible for the ordering of the edges in the bin.

5.2.2 The delete operation

This operation deletes a rectangle from the datstructure. Actually the ID field in the corresponding rectangle is set to zero. This implies that it is also necessary to clean-in the database every now and then.
For the delete operation the following actions are taken:

1 The bins in which the rectangle is stored are calculated
2 In deltabins[minbin] the bin which points to the deleted rectangle is located and the ID-field of that rectangle is set to zero.
3 The edges of the rectangle are removed from the bin-structure and the rectangle is removed from the rectangle list.

The delete operation is performed by:

```
Procedure Deleterect( xycoor :quartet; ids :integer);
```

This procedure calls a number of other routines which do the actual work.
Action 1 is performed by procedure binrange, see previous section for a description of this routine.

Action 2 &3 are performed by:

```
Procedure Bindelete( outlist :linkbin; ids:integer);
```

Here outlist is intended as the pointer to the bin in which the in-edge of the to be deleted rectangle has been stored.

Action 3 is performed by

```
Procedure Cleaners( var binar, deltabins :bins;
                    var startrect :linkrect);
```

This procedure calls two other procedures:

```
Procedure Dustman( var binar :bins); and
Procedure Rectdustman(var startrect :linkrect);
```

Dustman deletes all bins in binar which point to a rectangle with an id-field equal to zero. Binar may be either transbins or deltabins.
Rectdustman deletes all rectangles with a zero in the id-field from the rectangle list. Here startrect points to the first rectangle in the rectangle list.

5.2.3 The select operation

The select operation is used to select rectangles which overlap or lie within a certain distance of a rectangular box. The distance depends on the layers involved.
Input for this operation is a box, the output consists of a sorted list of edges. These edges belong to rectangles which lie within a certain distance from the box. The edges in the edgelist are sorted in non-descending order on xmin.
In performing this operation the following actions are carried out:

1 The bins in which the box falls are calculated.
2 Rectangles which lie in these bins are selected.
3 The edges of the selected rectangles which lie within a minimum distance of the box are placed on the output list.

The select operation is carried out by :

```
Procedure Rectselect( var outlist :linkbin;
                      xycoor  :quartet;   maskno  :integer)
```

Here xycoor contains the coordinats of the box, maskno contains the number of the layer of the box.
After execution of this routine outlist contains the pointer to the first record of the list of selected edges. These edges are ordered on xmin in non-descending order.
Rectselect uses the following procedures. For action one procedure binrange is used with option=extendrange.

Action 2 is performed by:

```
Procedure Makelist( var rectlist :bins;
                    minbin,maxbin :integer);
```

This procedure copies the startpointer of the bins in which the box lies, to rectlist.
The last action is carried out by

```
Procedure Sellisty( var rectlist :bins;
                    var outlist :linkbin;
                     xycoor  :quartet;  maskwind  :integer);
```

This procedure scans the bins copied in rectlist and selects the necessary edges in outlist.

The actual checking is performed by

```
Procedure Checkrect( point :linkbin; xycoor :quartet;
                     var outlist :linkbin;
                     maskwind :integer);
```

This procedure appends an edge of rectangle point".idpoint to the outlist, if this rectangle lies close enough to the rectangle given by xycoor.

5.3 The scanline algorithm

A scanline algorithm is used to find contours, to check spacing and width and to construct maxsets.
A scanline may be considered as an ordered list of scanline elements. The scanline represents a rectangular slice of a contour. Each scanline element contains information of the border of the contour. A scanline element has an origin which gives the x-coordinate where the element was created and the slice begins.
Further a scanline element contains ymin and ymax coordinates which give the range in which the element is situated.
Finally a scanline element contains a list of rectangles which are present in the scanline element.
In the scanline, the scanline elements are ordered according to their ymin value. Scanline elements do not overlap each other, but touching is allowed.
In pascal this looks as follows:

```
type  linkscanel = "scanel;{pointer to a scanline element}
      linkedge   = "edge;

      scanel     = record
                     ymin     :integer;
                     ymax     :integer;
                     edgelist :linkbin;
                     next     :linkscanel;
                   end;
```

```
      edge        = record
                      ymin     :integer;
                      ymax     :integer;
                      idpoint  :linkrect;
                      next     :linkedge;
                    end;

var   scanline    : linkscanel;
```

Roughly the scanline algorithm looks as follows

```
Procedure Scan( inlist :linkbin;
                maskno,masknol :integer;
                option :scanoption);
begin
  while edges on inlist do
  begin
    take next edge from inlist;
    if edge is an in-edge
    then insert edge in scanline
    else delete edge from scanline;
  end;
end;
```

The scanline algorithm works with edges, but the input list consists of elements of type linkbin. This means that a conversion must take place. This conversion is taken care of by :

```
Procedure  Scanadjust(  inbin:linkbin;  var  edge:linkedge);
```

There are now two operations that can be performed on the scanline, an insert and a delete operation.

5.3.1 The insert operation

For the insertion of an edge in a scanline the following recursive routine is used:

```
Procedure Scanins(var prepoint,startp,
                  scanline :linkscanel;
                  var scanout,scanend:linkbin;
                  var edgein :linkedge);
begin
  determine where and how the input edge
        overlaps the scanline;

  case kind of overlap of

  no overlap: insert edge in scanline;
  overlap   : begin
                split, if necessary a scanline element;
                update the splitted scanline element;
                newedge = edgein-(part of edgein which is
                          already inserted)
                {split edgein}
                Scanins( scanline, newedge);
              end
  end {case};
end;
```

Here startp points to the scanline element under concideration and prepoint to the scanline element before startp.
After the insert operation is completed, scanout points to a list of edges which overlap edgein.
Note: this list may contain edges from different scanline elements. In order to make a distinction between the edges of the different scanline elements, the up-field of the linkbin record is set to mark the beginning of a new group of edges belonging to the same scanline element.
This algorithm is a simplification of the real situation. Actually 11 different cases of overlap are considered.

```
edgein          <---------------->
                      a

scanline                <--------------------->
                                  b

edgein          <----------------->
                      a
new scanline            <---------X---------->
                            b          b

split edgein    <-------X--------->
                    a        a

scanline                <---------X---------->
                             b          b

insert edges
new scanline    <-------X---------X---------->
                    a      a,b          b
```

fig. 8. example of the scan algorithm.

In scanins the following routines are used:
Splitting of a scanline element:

```
Procedure Splitscanle( scanle :linkscanel;
                       edgein :linkedge;
                       kind   :integer);
```

This routine creates a new scanline element and inserts it immediately after the scanline element pointed to by scanle. The edgelist of the old scanline element is copied to the new one. The parameter kind influences the borders of the scanline elements as listed in the following table.

kind	old scanel		new scanel	
	ymin	ymax	ymin	ymax
5	no change	no change	edgein"ymin	edgein"ymin
6	no change	edgein"ymax	edgein"ymax	old scanel"ymax
7	no change	edgein"ymin	edgein"ymin	old scanel"ymax

```
Procedure Splitedge( edgein :linkedge;
                     scanle :linkscanel;
                     kind   :integer;
                     var edgeout1,edgeout2 :linkedge);
```

This procedure splits the input edge in two other edges, edgeout1 and edgeout2. The idfield of edgein and thein field are copied to edgeout1 and edgeout2. The ymin and ymax field are adjusted. This adjustment depends on kind as follows from the next table.

kind	edgeout1		edgeout2	
	ymin	ymax	ymin	ymax
5	edgein"ymin	scanle"ymin	scanle"ymin	edgein"ymax
others	edgein"ymin	scanle"ymax	scanle"ymax	edgein"ymax

The updating of a scanline element is done by two routines:

Procedure Modify(point:linkscanel; edgein:linkedge);

This routine copies the ymin and ymax values of edgein to the scanline element indicated by point. Also the edge is inserted in the edgelist of the scanline element.

Procedure Modifysmall(point :linkscanel; edgein:linkedge);

This procedure insert the edge in the edgelist of the scanline element.

Function Overlap(prepoint,startp, scanline :linkscanel;
edgein :linkedge) :integer;

This routine returns the kind of overlap of the input edge withthe scanline. Startp points to the first scanline element that must be evaluated.

5.3.2 The delete operation

When an out or out edge is encountered the scanline must also be updated. This means that one or both of the following actions must be carried out.

1 delete the edge from the edgelist from one or more scanline elements.

2 delte the scanline element(s) which have an empty edgelist.

This delete operation works as follows

```
Procedure delete(edge, scanline);
begin
  locate the scanline element for which
            scanline"ymin= edgein"ymin;

  while scanle".ymax <= edgein"ymax do
  begin
    split edgein
    Update the scanline element;
    if scanle".edgelist = empty
    then delete scanle;
    take next scanle;
  end;
end;
```

The delete operation is carried out by:

```
Procedure Delete( var edgein :linkedge;
                  var scanline :linkscanel);
```

The splitting of edgein is done by procedure splitedge (see before) where kind=6.
The actual opdate and delete is done by

```
Procedure Demodify(var scanline, scanle, prepoint
                   :linkscanel; var edgein :linkedge);
```

Here scanline points to the first scanline element. Scanle points to the scanline element under consideration and prepoint points to the scanline element before the one which is considered. Edgein is the edge which will be deleted from the scanline.

5.4 The Check routines

The width and spacing checks are integrated in the scanline algorithm. Each time the contents of the scanline changes, the width and spacingcheck routines inspect the scanline.

5.4.1 The width check routine

The width checking is performed by

```
Procedure  Checkwidth(  xycorners:quartet;  maskno:integer);
```

Here xycorners specifies the area in which a widthcheck is performed, maskno gives the layerno of the rectangle.
In Checkwidth the following actions are taken (see also the width check algorithm. First we select the relevant edges from the bin-structure. Next the edges with layer=maskno are selected by using

```
Function  Newlist(  edgelist:linkbin;  var  outlist:linkbin;
                    maskno,maskno2 :integer) :boolean;
```

The edges on edgelist which have as layerno maskno or maskno2 are selected and inserted in outlist. Function Newlist returns the value true if the edgelist contains edges with layer=maskno and edges with layer=maskno2.
Otherwise Newlist=false.

Next the procedure Scan with option=width is activated.

Option=width causes the following actions to be taken in the scanline algorithm.
When a scanline element is deleted the corners on each side of the deleted scanline element are inspected on convexity. If concave corners are found, these corners are checked for width violations.

```
Procedure Convexcorner( var botconvex, topconvex :boolean;
                        breakmin, breakmax, maskno,
                        xscan :integer; top :boolean);
```

Here scanline points to the first element in the scanline. Breakmin and breakmax give the range around the concave corner, in which a width violation could occur. Maskno gives the layerno of the contour which is beeing checked. The boolean top, when true indicates that we are dealing with a concave top corner. This is of importance for the error messages to be issued
Widthcheck tries to find scanline elements in the range breakmin-breakmax from which the origin lies closer then the minimum width from the concave corner.

5.4.2 The spacing checks

Because we use a scanline algorithm, two different kind of checks are necessary. The first check is applied on the scanline itself each time an edge is inserted or deleted. This check is performed by:

```
Procedure Spacingcheck( scanline :linkscanel;
                        maska, maskb,  xmin, xmax :integer);
```

Here scanline give the beginnin of the scanline. Maska and maskb indicate between which masks the spacing is checked, xmin and xmax, indicate between which x-coordinates the scanline element lies.
The routine checks whether the "gaps" in the scanline are large enough.

The second check is applied when a scanline element is deleted. In this case we look forward in the edgelist whether there is an edge which is to close to the deleted scanline element. This is done by

```
Procedure Lookforward( inlist :linkbin; xscan, breakmin,
                       breakmax, minspac,
                       maska, maskb :integer;
                       edgep :linkbin);
```

Here inlist is the inputlist of edges, xscan gives the current x-position of the scanlinea, breakmin and breakmax give the y-coordinates of the area to be inspected and xscan and xscan+minspac give the x-coordinates of the area to be inspected. Maska and maskb give the masks between which the spacing must be checked. Edgep points to the edge to be deleted.

5.5 Commands

The design rule checker recognizes the following commands:

command	action
0	end of checking, exit drc
1	set the window in which the editing takes place
2	insert a rectangle in the datastructure
3	delete a rectangle from the datastructure
4	analyze the error lists
5	check group of rectangles
6	check the complete layout in the window
7	activate the checker
8	deactivate the checker

Note: If the checker is activated then the inserted or deleted rectangle (command 2 or 3) and its neighbourhood is also checked for design rule violations. If the checker isn't activated, then these rectangles are placed on a waitlist, by issuing command 5 the rectangles on the waitlist are checked for design rule violations.

The commands are carried out by:

```
Procedure  Widthspacecom( icommand,ids,maskno :integer;
                          xycoor :quartet);
```

By this procedure the following procedures are called:

```
Procedure Checkwidth( xycorners:quartet;
                      maskno: integer);
```

This procedure checks the neighbourhood of the rectangle give by xycorners and maskno for width violations.

```
Procedure Checkspacing( xycorners:quartet;
                        maskno :integer);
```

This procedure checks the neighbourhood of the rectangle given by xycorners and maskno for spacing violations.

```
Procedure Checkwaitlist( waitlist :linkrect);
```

This procedure checks the neighbourhood of all rectangles on the waitlist for width and spacing violations.

```
Procedure Checkarea( inlist :linkbin;
                     option :scanoption);
```

This procedure checks the item indicated by option (width, spacing or scanl) of all rectangles addressed by the edges on the inlist.

```
Procedure Checkerror( option :scanoption;
                      errorlist :linkrect);
```

This procedure checks for design rule violations in all rectangles on the error list.

5.6 Communication with the layout editor

The communication between the design rule checker and the layout editor takes place by using an eventflag and a buffer belonging to that flag.
The checker (receiver) clears the eventflag and waits for the layout editor (transmitter) which sets the flag and fills the buffer with the command.
After the setting of the flag, the checker will read the command, execute it and clear the eventflag, thus enabeling the transmitter to send another command.
For the receiver the following routines are used:

```
Procedure Clref(var evflag ,status :integer); extern
(fortran);
```

This is a system routine which clears an eventflag.

```
Procedure Waitfr(var evflag,status :integer); extern
(fortran);
```

This system routine waits for the eventflag to be set.

```
Procedure Accep(var buf :buffer); extern (fortran);
```

This is a user routine, resident in file Accep.ftn. This routine reads the contents of the buffer.

Finally we have

```
Procedure Wsreceiv( var icommand, ids, layer :integer;
                    xycoor :quartet);
```

This routine reads the command from the buffer belonging to eventflag=50.

Here buffer =array[1..15] of integer. The commands are stored in the following format.

Buffer[1]= unused, occupied by system
Buffer[2]:= icommand, Buffer[3]:= ids,
Buffer[4]:= layer,
Buffer[5] ... Buffer[8] := xmin, xmax, ymin, ymax.

Apart from the previous routines the transmitter also uses

```
Procedure Send( var task :taskname;
                 var buf  :buffer2; var  evflag  :integer);
extern (fortran);
```

This system routine fills the buffer belonging to the eventflag and sets the eventflag. Here buffer= array[1..13] of integer. Buffer[1] and buffer[2] are used by the system.

5.7 Design rules

For the specification of the design rules the following arrays are used:

```
type cstraintar = array[1..nmask, 1..nmask] of integer;
     minwidthar = array[1..nmask] of integer;

var  constar,illoverl :cstraintar;
     minwidth         :minwidthar;
```

Where constar[maska,maskb] and constar[maskb,maska] contain the minimum spacing required between maska and maskb.

Illoverl[maska,maskb] <>0 indicates that overlap between maska and maskb is illegal.
Minwidth[maska] contains the minimum width of a contour in maska.

These arrays are stored in the file cnstr.nms in the following format:

First the array constar, which is a now a 7x7 array of integer;
Then the array minwidth which is a 1x7 array of integer;
Finally the array illoverl which is a 7x7 array of integer.

Note a minus sign in the specification of constar indicates that the involved layers have no constraints with each other.

6 CONCLUSIONS AND FUTURE DEVELOPMENTS

The design rule checker developed in this project is capable of executing width and space checks on an on-line basis. For the width check it is necessary that all rectangles in the layout satisfy the minimum width criterion.
The checker doesn't check the devices in a layout.
The design rules are stored in the file cnstr.nms. This file contains several tables in which the minimum spacing and width values are stored. Note the design-rule compiler has not yet been developed nor is there a format for the specification of the devices in the design rule file.
The next step in the completion of this design rule checker is to develop a format in which all devices can be uniquely specified.
The next problem that must be solved is the construction of the design rule compiler and its interface with the design rule checker.
Finally the device checker can be constructed. Here special attention for the specification of the devices is required.
Since the design rule checker is capable of recognizing devices, the step to layout extraction is not so large anymore.
It becomes attractive to extend the device description with a device model so that the parameters of this model can be calculated as well on an on line basis.
Another step further, the design-rule checker could have information about the connectivity of the layout, making it possible to verify the circuit description.

7 REFERENCES

Baird H.S.
Fast algorithms for LSI artwork analysis
Journal of Design Automation and Fault Tolerance Computing
Vol2, no2, may 78

Berkel v. K
Automatic Integrated Circuit Layout Verification: A literature study
dept. of electrical engineering
University of technology Delft
sept. 80

McCaw C.R.
Unified shapes checker - a checking tool for LSI
Proceedings on design Aut. Conf. 79

Delhy C.A.C.A.
ISLE an Interactive Symbolic Layout Editor
dept. of electrical engineering University of technology Eindhoven
oct. 82

Dobes I. and Byrd R.
The automatic recognition of silicon gate transistor geometries
Proceedings on design Aut. Conf. 76

Haken D.
A geometric design rule checker
Computer Science dept. Carnegie-Mellon University
jun. 80

Hofmann M. and Lauther U.
HEX: an instruction driven approach to feature extraction
20th design Aut. Conf. 83

Lauther U.
An O(N log N) algorithm for boolean mask operations
18th design Aut. Conf. 81

Lyon R.F.
Simplified design rules for VLSI layouts
Lambda first quarter 81

Marek-Sadowska M. and Maly W.
A hierarchical layout description for
artwork analysis of VLSI IC

Theeuwen F.
Design guide
Dept. of electrical engineering University
of technology Eindhoven
august 83

Whitney T.
A hierarchical Design-rule checking algorithm
Lambda first quarter 81

Wilcox, Rombeek, Caughney
Design rule verification based on one dimensional
scans
Proc. on design aut. conf. 78

Wilmore J.A.
Efficient boolean operations on IC masks
18th design aut. conf. 81