An Hierarchical and Technology Independent Design Rule Checker

T. G. R. van Leuken
J. Liedorp

Circuits and Systems Group
Department of Electrical Engineering
Delft University of Technology
The Netherlands

Abstract

This paper describes a uniform and new approach to a technology independent and hierarchical artwork verification method. It is based upon the 'augmented instance' of a cell and the stateruler scan algorithm. By making an hierarchical instance of a cell, the cell is made independent of other cells. The artwork verification programs based upon the two mentioned concepts exploit the hierarchy and repetition present in the layout description of an integrated circuit. That way the run time and memory requirements are no longer a function of the number of layout primitives in the fully instanced integrated circuit, but only of the number of primitives defined in the original hierarchical layout description. In the method of artwork verification described in this paper the design rules that can be tested upon are based upon the presence of combinations of masks. All combinations of masks can be tested with respect to each other, so the programs for the verification of the artwork are largely technology independent. Aside from handling the verification in an hierarchical manner, the main problem addressed by the method is the efficient handling of the large class of possible design rules. We describe the concepts and their implementation. The results are illustrated by some examples. The techniques presented have been implemented for paraxial geometrics. They also are usable in the general context.

Date: October, 1986
Last revision: May, 2003.

1. Introduction

The growing complexity of the artwork of integrated circuits necessitates the use of structured design methods. An important issue in this respect is the possibility to use hierarchy in the description of an integrated circuit, i.e. a cell in a description may call another cell etc. In most artwork description languages this concept of hierarchy is present. For efficiency reasons it is very desirable that programs that are involved with the design and verification of an integrated circuit can exploit this hierarchy. The following demands are made on our system:

a. The freedom of the designer's methodology should in no way be impaired by the tools (for that reason we will not use the notion 'hierarchical protection frame').

b. Minimal complexity both in the scanning and in the handling of the multitude of design rules must be achieved.

c. The hierarchy must be exploited as much as possible.

In this paper we will show a method for artwork verification that optimally exploits this. We will describe the underlaying principles and describe the way they are used to form an efficient way for artwork verification.

In section 2 we will define the 'augmented instance' of a cell. In doing so we will be able to see an hierarchical cell description as the composition of a number of independent augmented cell instances.
The augmented instances are made independent by requiring that cells are interconnected by means of 'terminals'. We will define a terminal as an area in a cell on a certain mask where primitives of other cells, defined in the same mask, may overlap. A difference between our approach and that of Newell and Fitzpatric [1] is that our approach preserves the cell hierarchy in its original form.

In section 3 we will discuss the conversion of the augmented instances to line segments. The algorithm for doing so will be based upon the stateruler scan algorithm. This algorithm also forms the basis for the artwork verification programs discussed in the next sections.

In section 4 we will discuss the artwork verification programs, using the line segments as input.

In section 5 some results of tests with the programs mentioned will be given. It turns out that the algorithm indeed is linear with respect to the number of primitives in the cell under test, as was already stated in [2].

Section 6 at last will give some conclusions.

2. Augmented Instancing of a cell

The layout description of an integrated circuit is usually available as an hierarchy of cells, in which each cell is made up of primitives (rectangles, wires etc.) and references to other cells. The basic problem we meet when we try to exploit the hierarchy is the influence the cells can have on each other. Consider for instance the cell given in figure 2.1. To interconnect the cells m2 and m3, referenced in cell m1, among themselves as well as with the primitives defined in cell m1, primitives of different cells have to overlap.

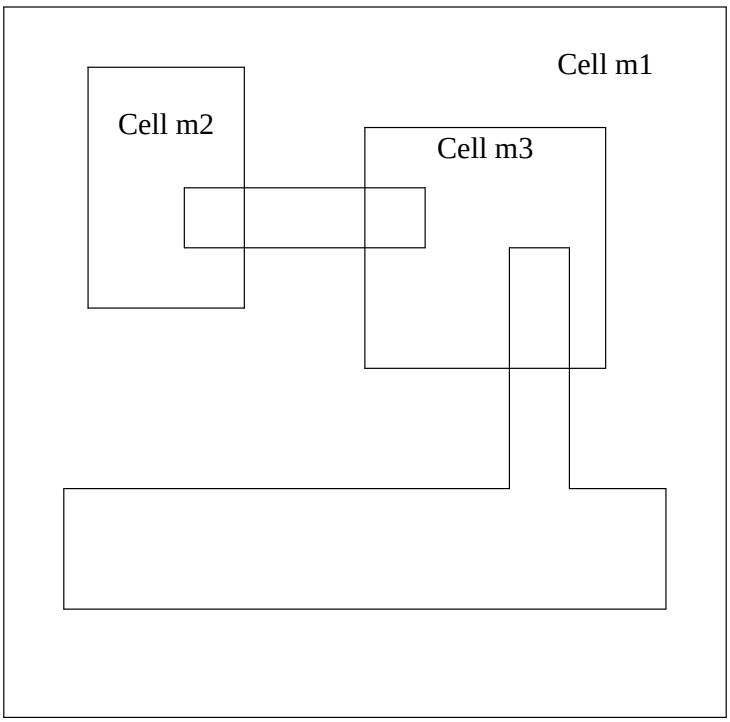

Figure 2.1. Cell interconnection

These primitives necessarily cross an imaginary boundary that can be drawn at some distance around the cell to protect its contents from disturbances from the outside. The problem now is that the analysis of two instances of the same cell can be quite different. For instance in the case of artwork checking, the interfringing primitives may create new errors or, conversely remove errors that were previously present. Also new and unwanted elements may be generated by overlapping primitives originating from different cells.

The solution we propose was developed with existing design practice in mind. It limits the designer in some ways, but leaves him free as much as possible. The restrictions imposed on an individual cell design are:

1. The referenced cells should be free of design rule errors.
2. The implied circuit of the cells referenced may not be changed as a result of interfringing primitives.

To make a consistent design system we will have to insure that these restrictions are obeyed. The first restriction is obeyed almost automatically. Given a design rule checker it suffices to check the cells in the cell hierarchy individually. The second one is more difficult. By checking the places where primitives of different cells have an overlap, and signaling if they do so, the violations of the second rule can be discovered too. This check is carried out in the program nbool(see section 4). In the sequel we shall assume that the mask data is given in the form of an ordered line segment file, one per mask, with extra masks for the intercell terminals (see further). Such files can be obtained by the same type of algorithm as for the design rule checker, see section 3.

To handle the hierarchical design rule checker, we define the 'augmented cell instance' of a cell. It contains the information necessary for the checking of the design rules, of the cells independently from its subcells. It can also handle the extraction of the circuit from the artwork. (see the paper on extract on this volume). Crucial in the definition of the augmented cell instance is the concept of 'active region'. The active regions of a cell surround the places where the implied circuits of the referenced cells can be changed, or the design rules might be violated. We associate an active region with each primitive defined in the cell, as well as with each overlap of cell frames. A cell frame is an orthogonal rectangle that surrounds a cell as close as possible. The active regions associated with the primitives of a cell are determined by growing the dimensions of the primitives with a constant, but mask dependent parameter, the 'expand_offset'. The active regions associated with the overlapping cell frames are determined by growing the dimensions of the overlap region with the maximum of the expand_offsets just mentioned. Furthermore we define a 'checktype' to distinguish primitives originating from different cells. The checktype '0' is associated with primitives defined in the top level cell, all other primitives get a positive integer as checktype.

The program which determines the 'augmented instances' of cells is the program mkbox. This program reads from the database the data of the primitives of the cells involved and the files determining their hierarchy. From this data it makes (for each cell desired) a file consisting of all the rectangles of the artwork forming the 'augmented instance' of that cell. This file also contains the following primitives:

- All rectangles of the top level cell. Those rectangles have a checktype zero
- All rectangles of sub_cells and sub_sub_cells etc.(recursively) that have an overlap with one of the active regions of the rectangles of the top level cell. As their checktype these rectangles have a positive integer characteristic for the cell they originate from.

- All rectangles of sub_cells that have an overlap with the active regions associated with the overlap of sub_cells. As their checktype these rectangles also have a positive integer indicating which cell they originate from.

3. Line Segment Conversion, the Stateruler

The conversion of the orthogonal rectangles to line segments is done based on the stateruler scan algorithm given in figure 3.1. The algorithm is applicable in many situations where the artwork data is used as input. E.g. it can equally be used for (layout to circuit) extraction, design rule checking and the generation of derived geometries (polygons, groups, pattern generation). The complexity of the algorithm is linear in the number of edges in the layout. The method is best illustrated with the simple example of rectangle to vertical line segment conversion.

```
begin

    initialize stateruler
    event_status := select_event(event, event_pos);
    stateruler_pos := event_pos;
    while(event_status is not NIL)
    loop
        repeat [make a stateruler profile]
            insert_event(event, stateruler_pos);
            event_status := select_event(event, event_pos);
        until( event_status is NIL or stateruler_pos < event_pos);
        while( stateruler_pos < event_pos) [analyze stateruler]
            stateruler_pos := analyze_stateruler(stateruler_pos);
        stateruler_pos := event_pos;
    end loop
    while(stateruler_pos < MAX_INTEGER) [make final analysis ]
        stateruler_pos := analyze_stateruler(stateruler_pos);
end
```

Figure 3.1. Stateruler Scan Algorithm

A stateruler contains a vertical cross section of the layout description as a sorted list of fields. The stateruler is made by scanning the layout from left to the right. Each field has its own state, determined by a number of variables. What state variables are used depends on the application the algorithm is applied to (e.g. design rule verification or circuit extraction). For rectangle to line segment conversion the state is determined by the duration, the overlap duration and the checktype. The duration gives the x_value for which the field will cease to exist. Likewise the overlap duration gives the x_value for which the overlap of rectangles in the field will cease to exist. If there is no overlap this variable is undefined. The checktype of a stateruler field depends on the checktype of the rectangles forming the field.

The algorithm proceeds by making and analyzing what we call stateruler profiles. A stateruler profile is made by repeatedly selecting an event and inserting that event in the stateruler. The selection of events is based upon a selection criterion. In the case of rectangle to line segment conversion the events are the rectangles. The selection criterion is the smallest left value of the rectangle and if two rectangles have the same left value,

the smallest bottom value. The procedure 'select_event' returns the event_status. The event_status will be 'NIL' if there are no more events to select.

The insertion of an event is done by comparing the bottom and the top values of the event with the bottom and top values of the fields in the stateruler, updating the state of the existing fields and creating new fields if necessary. In the case of rectangle to line segment conversion the state update is done according to a number of rules derived from the way we want to represent a polygon by vertical line segments. Consider figure 3.2a. The rectangles depicted in this figure must be converted to the line segments given in figure 3.2b.

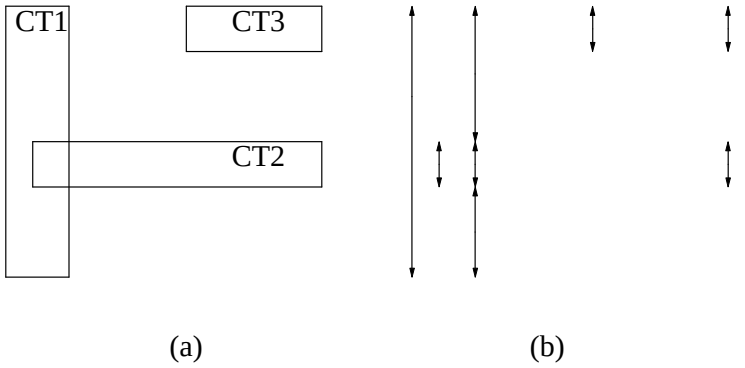

Figure 3.2. Polygon and line segment representation

We distinguish several types of line segments, characterized by the occurrence type. They are:

- START: A start segment will have the interior of the polygon located to the right of it.
- STOP: A stop segment will have the interior of the polygon located to the left of it.
- CHANGE: A change segment indicates a change of checktype. The polygon area to the right will have the checktype of the CHANGE segment.
- START_OV: A start_ov segment indicates the start of an overlap of two areas. The overlap is situated to the right of the segment.
- STOP_OV: A stop_ov segment indicates the cease of an overlap. The overlap will be situated to the left of this segment.

Also combinations of the types mentioned above may occur, so e.g. an segment that is as well START as CHANGE etc.

Whenever two or more rectangles with different checktypes start to overlap we will generate a CHANGE segment. The checktype of such a segment will be made zero. At the end of the overlap another CHANGE segment is generated, restoring the checktype.

In doing so we can assure that these overlap areas are checked because all areas with checktype zero will be fully checked. (see section 5).

Whenever a stateruler has been built the routine ’analyze_stateruler’ is called. In the case of rectangle to line segment conversion this routine generates the line segments based upon the state information present in the individual fields. The generated segments are characterized by six parameters:

- x_value: The x_value of the segment.
- occurrence: the occurrence type as described above.
- y_bottom: The y_bottom value of the segment.
- y_top: The y_top value of the segment.
- connection_type: This variable indicates if the segment has connections to the upper or the lower side or both.
- group_nbr: This variable is an integer indicating what polygon the segment belongs to.
- checktype: This variable indicates the cell the edge belongs to.

The procedure ’analyze_stateruler’ returns the next position for which the stateruler should be analyzed again. It returns MAX_INTEGER if that is not necessary.

The final result will be a set of line segment files which together represent the instanced cell. These files form the basis for subsequent analysis, as discussed in the next two sections.
The program performing this conversion is the program makevln.

4. Design Rule Checking

Design rule checking is needed to see if all rules concerning the dimensions of primitives of a designed integrated circuit obey the rules imposed upon it by the technology the integrated circuit has to be made in. The checks may involve gap or width checks on a combination of masks present or a gap or overlap check between two combinations of masks present. Every technology has its own set of design rules that has to be obeyed. To make the design rule checker as much independent of a special process as possible, it is based upon combinations of masks one may specify for a certain technology.
Generally we may split the problem of design rule checking into two parts:

- A part that for a certain rule selects the items involved.
- A part that does the actual checking upon these items.

For a hierarchical design rule checker the first again may be split in two parts:

- A part that picks from the total amount of data of the artwork of the integrated circuit the minimum part that is needed for the checking of a certain part of the cell. This part has been described in the previous sections.
- A part that selects the data needed for a check of a certain design rule from this data gathered.

This second part is carried out by the program *nbool*, which performs all and, or and negate operations needed to select mask combinations needed for the checker. It does so in one pass, with a linear complexity.

As stated the checks to be performed are width, gap and overlap checks. The width check always concerns one (combination)mask and gap checks may concern one or two (combination)masks. Overlap checks always concerns two (combination)masks. Therefore the checker itself is also divided into two parts:

1. A part that does width and gap checks concerning one (combination)mask.
2. A part that does overlap checks and gap checks concerning two (combination)masks.

So the design rule check is done in three steps:

1. First all the combination masks needed in the check are made by the program *nbool*. The files that have to be made must be given in a file booldata, the format of which is given in appendix A.
2. Then the checks concerning one (combination)mask are carried out by the program *dimcheck*. The checks that have to be carried out must be given in a file *dimcheckdata2*, the format of which is also given in appendix A.
3. At last the checks concerning two (combination)masks are carried out by the program *dubcheck*. The checks that have to be carried out must be given in a file

dubcheckdata, the format of which is also given in appendix A.

Two design rule checkers are in use at the moment:

1. A simple checker *autocheck* which only checks for width and gap errors per mask; so errors stemming from combinations of masks will not be looked for. Autocheck is implemented as a shell_script, which calls the program *dimcheck* with the correct options.
   How *autocheck* is called and what options are possible is given in appendix E.

2. The check program *dimcheck*, which preforms all the design rule checks, so checks in the same mask as well as checks between (combinations of) masks.
   Dimcheck is also implemented as a shell_script. It first calls the program *nbool* which determines the vln_files of the combination_masks needed for the checks. After that the programs *dimcheck* and *dubcheck* are called, which use these files to perform the checks, together with the cell_data of the cell(s) to test and the technology files *dimcheckdata2* and *dubcheckdata* Dimcheck thereby checks for width and gap errors in the same (combination)mask and *dubcheck* checks for gap and overlap errors between two different (combination)masks.
   Schematically this is shown in the next figure:

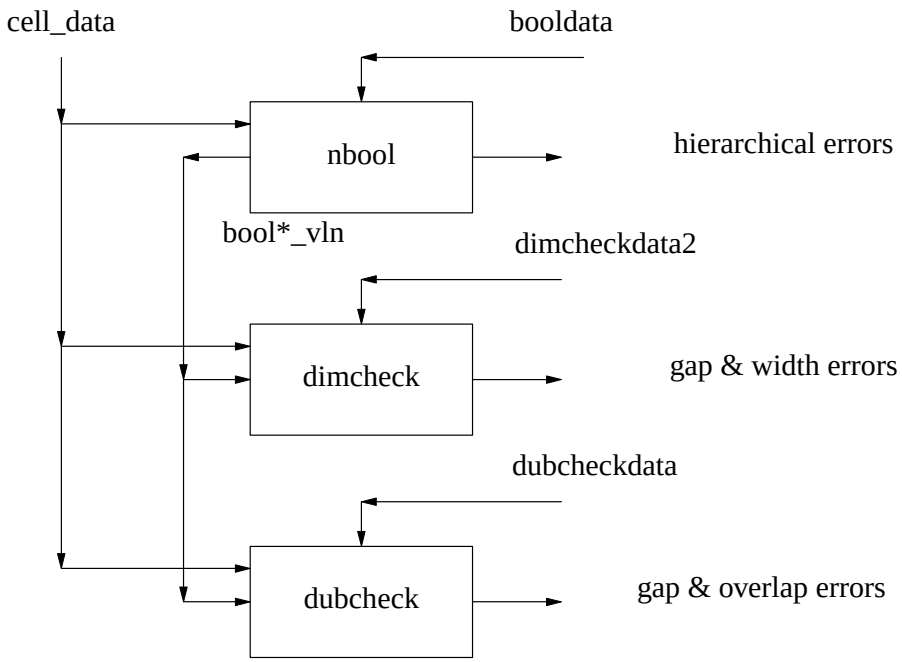

Figure 4.1. The checker dimcheck

How *dimcheck* is called and what options are possible is given in appendix E.

A short description of programs mentioned will be given in the next sub_sections.

4.1 The program nbool

Nbool is a program to generate logical combination masks, which are a combination of input masks in vertical line segment format (vln format).
As input it uses a file with the description of the logical formulas of the masks to be made. For a description of this file see appendix A. Also the vln files mentioned in these formulas and a file with the name of the cell(s) of which the files have to be made is needed. It uses the stateruler algorithm in a similar way as described in section 3. Globally the program works as follows:

```
read 'logical' file and set up an internal logical structure.
for each cell do {
    while not all segments inserted {
        read segments and select segment to insert.
        if ( x_segment != stateruler position) {
            check stateruler for hierarchical errors of
                  the cell.
            determine using the internal logical structure
                  and the layers present in the stateruler
                  fields what new segments have to be made
                  to what output_mask.
            update stateruler
        }
        insert new segment
    }
    check stateruler for hier. errors
    output segments from the last stateruler position

    add group_numbers to all segments made.
}
```

First the file which contains the logical combinations to make is read and a structure is made to indicate what layers must be present for a segment to belong to a certain logical formula, and what layers must not be present. This structure is used by the analysis of the stateruler.

The stateruler fields in this case must contain as its state two vectors, one indicating the masks present in the past(i.e. left of the stateruler) and one indicating the masks that will be present in the future(i.e. right of the stateruler). Also the check types of the edges must be stored in it.

In the stateruler process of making and analyzing stateruler profiles events are inserted which are read from the input vln files. The selection criteria are the x_value of the segment and its bottom y_value. Whenever during the analysis of a stateruler a change of layer combinations occurs, which is indicated by the fact that that the past_vector is different from the future_vector, the past_ and future_vectors are compared with the logical structure built to see if the field gives rise to the generation of line segments in one or more of the output masks.

If in a field different checktypes occur this indicates a possible violation of the hierarchical rules. These errors are reported by *nbool* in the following cases:

- Overlap of interconnection layers without the presence of a terminal in that layer.
- Overlap of layers which do not interconnect.
- Overlap of different layers belonging to different cells indicating that one has possibly created an unwanted element.

For a more detailed description of the program *nbool*, see appendix B.

4.2 The program dimcheck

The program *dimcheck* performs gap and width checks on one mask.
As its input it uses a file containing the masks that have to be checked and the widths and gaps they have to obey. Also a reduced gap may be defined for gaplengths smaller then a certain value. For a description of this file see appendix A. Also the vln files mentioned in the file above and a file containing the cell(s) to check must be present.
It uses the stateruler algorithm in a similar way as described in section 3. The global way the program works is:

```
for each cell do {
    for each vln_file do {
        while(not all segments read) {
            read segment from vln_file.
                if(x_segment != stateruler position) {
                    analyze the stateruler for presence of
                          width  and gap errors.
                    update stateruler
                }
                insert segment in the stateruler.
            }
        }
        analyze the last stateruler for width  or gap errors
    }
}
```

The events of the algorithm here are the segments read from the vln file.
In the stateruler the following variables are recorded:

- The x_position of the edge in the field previous to the one where the stateruler is analyzed.
- The status of the layer (PRESENT, NOT_PRESENT, CHG_TO_PRESENT or CHG_TO_NOTPRESENT).
- The group_number of the edge.
- The group_number of the previous edge in the field.

- The check_type of the edge, indicating from which cell the edge is originating.
- The check_type of the previous edge in the field.
- The status of an help_layer

Depending on the status of the layer in a stateruler field during analysis, width or gapchecks are performed in the x_ and y_direction. The group_numbers of the edges thereby can be used to suppress gap errors that occur between edges of the same polygon. The check_types are used to suppress errors that occur between edges belonging to the same sub_cell, as these already will be reported when this sub_cell is checked.
For a more detailed description of the program *dimcheck*, see appendix C.

4.3 The program dubcheck

The program *dubcheck* performs gapchecks between to different masks and determines if a mask is overlapped by another mask with a certain value. As its input it uses a file containing the files that have to be checked with respect to each other and the distance or overlap they have to obey. Also an integer is given to indicate what kind of gap_check or overlap_check has to be performed. For a description of this file see appendix A. Furthermore the vln files stated in the file mentioned above must be present and a file containing the cells to be checked.
It also uses the stateruler algorithm in a similar way as described in section 3. The global way the program works is:

```
for each cell do {
    for each line of check_file do {
        while (segment_files not empty) {
            read segment and select segment to insert
            if( x_segment != stateruler position)
                analyze the stateruler for the presence of
                      gap or overlap errors.
                update stateruler
            }
            insert segment in the stateruler
       }
       analyze last stateruler.
   }
}
```

The events of the algorithm here are the segments read from the two vln files. The selection criteria again are the x_value and the y_bottom value of the segment.
In the stateruler the status (PRESENT, NOT_PRESENT, CHG_TO_PRESENT or CHG_TO_NOTPRESENT) of both masks is recorded, together with the groups and checktypes of the edges and the presence of an helpmask.
The analysis of the stateruler is done in different procedures: one for gap errors and one for each kind of overlap.
At present the following gap and overlap checks are implemented in *dubcheck*:

- gap checks which only report errors for non_overlapping items.
- gap checks which report errors for overlapping and non_overlapping items
- overlap checks for a full overlap
- overlap checks for overlap over two opposite sides
- overlap checks only in places where the helpmask is not present
- overlap checks on sides set earlier by *dubcheck*

For a more detailed description of the program *dubcheck*, see appendix D.

5. Results

The programs described in the previous segments have been written in the program language C and are running under the UNIX operating system on a HP9000 series 500 machine. All programs in the design rule check system are based upon the stateruler concept. As shown in [1] this algorithm is linear with respect to the number of edges in the design of an integrated circuit. So one might expect the checker system also to be linear in this aspect. To investigate this the checker has been applied to a cell containing a random counter with about 1500 edges. Thereafter the checker has been applied to arrays of this cell of increasing size. The time needed for the checking is recorded. This gives the following results:

array size	number of edges	convert (hh:mm:ss)	check (hh:mm:ss)
1 x 1	1476	38	1:50
2 x 2	5904	2:01	6:57
3 x 3	13284	4:39	16:33
4 x 4	23616	9:59	33:34
5 x 5	36900	15:17	57:15
6 x 6	53136	24:39	1:14:10
7 x 7	72324	28:29	1:49:56
8 x 8	94464	43:40	2:27:32

TABLE 5.1. Checker cpu times

In this table under convert the cpu time to make the line segment files is recorded and under checker the cpu time needed to perform the boolean operations and do the actual checks. From this one may conclude that the check time indeed is about linear with the number of edges. The results have been obtained making no use of the hierarchy of the cell. The time saved by making use of the hierarchy of course is very much dependent of the number of repetitions of subcells in the cell. The cell rand_cnt mentioned above, is hierarchically build up in the following way.

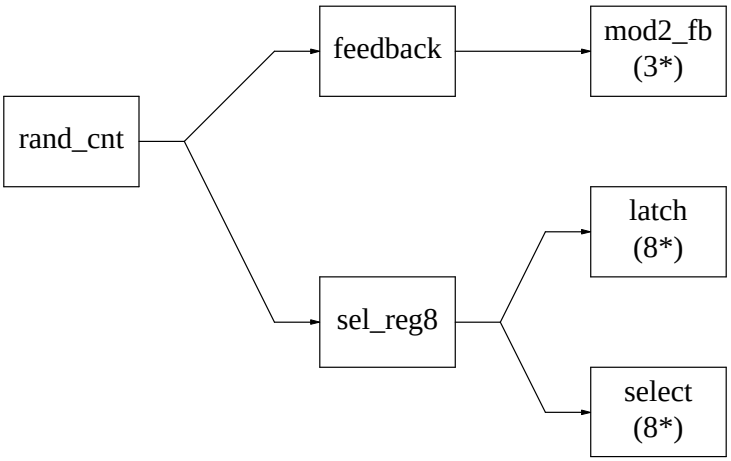

Figure 5.1. Cell Hierarchy

In the next table the results of the checking of this cell in an hierarchical way and linear are compared.

cell	linear		hierarchical	
	convert (seconds)	checker (seconds)	convert (seconds)	checker (seconds)
mod2_fb	9	26	9	26
latch	10	24	10	24
select	8	20	8	20
feedback	13	28	13	23
sel_reg8	32	95	29	56
rand_cnt	38	110	20	34

TABLE 5.2. Comparison between hierarchical and linear expanded cells

We see that the time needed to check the cells in a hierarchical way is much less for top level cells in this case, even though some overlaps are present. Even more dramatic changes in time may be expected if the repetitions of cells is greater.

6. Conclusions

Very important issues in achieving efficient operations in design rule checking are:

1. The exploitation of the hierarchy.
2. The generation of combination masks necessary for the checking of intermask rules.
3. The complexity of the scan itself.

We believe that we have achieved near optimal results on the three counts. Hierarchy is handled by making cells independent (for checking purposes) from their sub_cells through the notions of ’augmented instance’ and ’checktype’. The stateruler concept allows for a single pass to determine all combination masks needed. The scan itself is linear in the number of edges. Although at present only implemented for paraxial geometries the principles are generally applicable. At the moment we are extending the method to general geometries.

7. APPENDIX A: Implementation of Technology

The checks that have to be performed on the artwork of an integrated circuit vary from technology to technology. This appendix deals with the way the design rules must be implemented in the design rule checker.
In general we may distinguish between two items with respect to the technology:

1. Data about the masks used in a certain technology,such as mask_name, mask_type etc.
2. Data that are specific for the design rule checker, such as minimum widths and gaps etc.

The data mentioned under (1 are stored in a technology file for use by all programs needing it. In this paper we will not discuss this, but assume this file to be present. We will restrict ourselves to the data mentioned under (2.
This data is stored in four files for each technology present.

1. A file *booldata*, used by the program *nbool*, in which the logical combinations of the combination masks needed are described.
2. A file *dimcheckdata1*, used by the program *dimcheck*, which specify the width_ and gap_checks that have to be carried out if the single_layer checker *autocheck* is used.
3. A file *dimcheckdata2*, used by the program *dimcheck*, which specify the width_ and gap_checks in one (combination)mask that have to be carried out if the multy_layer checker *dimcheck* is used.
4. A file *dubcheckdata*, used by the program *dubcheck*, which specify the gap and overlap checks that have to be carried out between two different (combination)layers.

For the format of these files see the next section.

Our design rule checker can handle design rules of one of the following types:

1. Width checks.
   To implement a width check on a (combination)mask, the following steps must be taken:
   a. In the case of a combination mask, the logical formula of that mask, if not yet present, must be included in the file *booldata*. For the format of this file see later on under the sub_section on file formats.
   b. The (combination)mask to be checked must be given in the file *dimcheckdata1(2)*. In this file the minimum width of the mask must be given too. For the format of this input file see the sub_section on file formats later on in this appendix.

Examples of rules that can be tested this way are e.g. in the nmos process:
the width of the items of the diffusion mask, the width of the active areas etc.

2. Gap checks between items in one (combination)mask.
   To implement a gap check on a (combination)mask, the following steps must be taken:

   a. In the case of a combination mask, the logical formula of that mask, if not yet present, must be included in the file *booldata*. For the format of this file see later on under the sub_section on file formats.

   b. The (combination)mask to be checked must be given in the file *dimcheckdata1(2)*. In this file the minimum gap between areas of the mask must be given too. The possibility also exists to specify in this input a reduced gap in case the gaplength is smaller than a certain given value. With an helpmask specified in *dimcheckdata2* one can change the test so, that the gap is only tested at places where the helpmask is not present. One also can determine if one wants error_messages from gap_errors within the same polygon or from polygons with touching corners by specifying an integer kind in the file *dimcheckdata1(2)*. For the format of this file see the sub_section on file formats later on in this appendix.

   Examples of rules that may be tested this way are in the nmos process e.g.: the gap between unrelated diffusion areas, the gap between unrelated metal areas etc.

3. Gap checks between two (combination)masks.
   To implement a rule for the gapcheck between two masks, the following steps must be taken:

   a. In the case that one or both masks are combination masks, the logical formulas of these masks, if not yet present, must be added to the file booldata (for format see sub_section on file formats).

   b. The masks between which the check has to be carried out must be added to the file *dubcheckdata*. The minimum gap between unrelated areas of the masks must be specified here too. The possibility here also exist to specify a reduced gap if the gaplength is smaller than a certain given value. Furthermore with the integer kind one can specify if gap errors between overlapping items must be reported or not .

   Examples of rules that can be tested this way are e.g. in the nmos process:
   the gap between an undercrossing and unrelated diffusion and the gap between unrelated poly and diffusion.

4. Overlap checks of (combination)masks.
   To implement a rule to test an overlap of one combination(mask) over another one, the following steps must be taken:

a. In the case that one or both masks are combination masks the logical formulas for these masks, if not yet present, must be added to the file *booldata* (for format see sub_section on file formats).

b. The masks concerned must be given in the file *dubcheckdata*. Here also the value of the overlap must be given. One also must specify what kind of overlap one wants to test, by specifying an integer kind. At present the possibilities are:

- full overlap
- overlap over two opposite sides
- overlap only where a specified helpmask is not present
- left_right and/or bottom_top overlap only if an internally set array tells to do so. This array is set by stating tests in *dubcheckdata* with kind is 4 and kind is 5. These tests also must be defined before this last kind of overlap can be tested.

Examples of rules that can be tested this way are e.g. in the nmos technology: overlap of metal over a connect_cut, overlap of poly over an active area etc.

7.1 file formats

This section describes the file_formats of the files used by the design_rule checker.

1. The file *booldata*, read by the program *nbool*, contains the logical combination of masks to be made.

   Example:

```
od_vln nw_vln sp_vln ps_vln con_vln cop_vln
cps_vln cb_vln in_vln sn_vln               : filenames
od_vln&!nw_vln                             :  0 OD.3.1
od_vln&nw_vln                              :  1 OD.4.1.1
od_vln&sp_vln&!nw_vln                      :  2 OD.3.2+SP/SN.3.3+4.3
od_vln&ps_vln                              :  3 PS.3.1+PS.5.1
sp_vln|sn_vln                              :  4 SP.3.1+SN.3.1
od_vln&ps_vln&nw_vln                       :  5 SP.3.2+SP.4.2
od_vln&!ps_vln                             :  6 OD.2.1
od_vln&con_vln&!nw_vln|
od_vln&cop_vln&nw_vln|od_vln&ps_vln        :  7 SP/SN.3.3+4.3
od_vln&sn_vln&nw_vln                       :  8 SP/SN.3.3+4.3
od_vln&ps_vln&!nw_vln                      :  9 SN.3.2+SN.4.2
od_vln&con_vln                             : 12 CON.3.1+CON.3.2
od_vln&con_vln&sn_vln&nw_vln               : 13 CON.3.3+CON.3.4
od_vln&cop_vln                             : 14 COP.3.1+COP.3.2
```

```
od_vln&sp_vln&cop_vln&!nw_vln              : 15 COP.3.3+COP.3.4
od_vln&ps_vln&cps_vln                      : 16 CPS.4.1
cps_vln&ps_vln                             : 17 CPS.4.2+CPS.4.3
con_vln&!in_vln|cop_vln&!in_vln|cps_vln&!in_vln : 18 IN.3.1
con_vln&in_vln|cop_vln&in_vln|cps_vln&in_vln : 19 IN.3.2
cb_vln&in_vln                               : 20 CB.1.1
```

The first lines of the file until the first ':' contain the names of the input files that are involved in the formulas to come. After that each line of the file contains the logical formula to make. In this formula the logical AND operation is indicated by the character '&', the logical OR operation by '|' and the negation operation by the character '!'. The precedence of the operators is !, &, |. The end of the formula is indicated by the ':' on the line. After this ':' a number is given indicating the name of the file where the result has to be stored. The name of the file becomes bool_nn, where nn is the number just mentioned. After this number a string is given indicating what design rule is involved with the operation.

2. The files *dimcheckdata1* and *dimcheckdata2*, read by the program *dimcheck*, contain the names of the files to be checked and their width and gap dimensions.

   Example:

```
nw_vln   NOFILE  12 15  0 0 0  NW.1.1+NW.2.1
od_vln   NOFILE   6  6  0 0 2  OD.1.1+OD.1.2
ps_vln   NOFILE   6  6  0 0 2  PS.1+PS.2.1
sp_vln   NOFILE  12 12  0 0 0  SP.1.1+SP.2.1
sn_vln   NOFILE  12 12  0 0 0  SN.1.1+SN.2.1
con_vln  NOFILE   6  6 -1 6 2  CON.1.1+CON.2.1
cop_vln  NOFILE   6  6 -1 6 2  COP.1.1+COP.2.1
cps_vln  NOFILE   6  6 -1 6 2  CPS.1.1+CPS.2.1
in_vln   NOFILE   7  7  0 0 2  IN.1.1+IN.2.1
cb_vln   NOFILE 150 80  0 0 2  CB.3.1+CB.4.1
```

   Each line of one of these files must contain the following items in the order given:

   1. The name of the file to be checked.
   2. Eventually the name of an help_layer; if not needed 'NOFILE' is coded here. If a layer is specified errors will only be reported in places where this layer is not present.
   3. The minimum width of elements on the file. If it is zero no check will be carried out.
   4. The minimum gap between two elements on the file. If it is zero no check will be carried out.
   5. The minimum gap between elements on the file for short lengths of the gap. If a negative value is given here the program *dimcheck* will interpret it as an maximum width check, with the maximum value for the width given in the next item.

6. The maximum length of the gap for which the reduced gap may be applied, or if the previous item is negative the maximum value of the width permitted.
7. The value for kind. This variable may have one of the following values:
   - 0: gap_errors between edges of the same polygon and errors stemming from touching corners will not be reported.
   - 1: errors stemming from touching corners will not be reported, but gap_errors between edges of the same polygon will be reported.
   - 2: gap_errors between edges of the same polygon will not be reported, but errors stemming from touching corners will be.
   - 3: gap_errors between edges of the same polygon will be reported as well as errors stemming from touching corners.
8. A string indicating the design rule(s) involved.

In this example only primary vln files are used. However one may also use vln files made by *nbool*, so files bool_nn.

3. The file dubcheckdata, read by the program *dubcheck*, contains the names of the files to be checked and the gap and overlap dimensions.

   Example:

```
bool_0  nw_vln NOFILE 0 20 0 0 0 OD.3.1 (OD - NW)
bool_2  nw_vln NOFILE 0 20 0 0 0 OD.3.2 (p+OD - NW)
bool_1  nw_vln NOFILE 10 0 0 0 0 OD.4.1.1 (ovlp NW - OD)
bool_3  ps_vln od_vln  5 0 0 0 2 PS.3.1 (ovlp PS - gate)
od_vln  ps_vln NOFILE  0 3 0 0 0 PS.4.1 (PS - OD)
bool_3  od_vln ps_vln  6 0 0 0 2 PS.5.1 (ovlp OD - gate)
bool_1  bool_4 NOFILE  6 0 0 0 0 SP.3.1 (ovlp SP - OD)
bool_5  sp_vln NOFILE  6 0 0 0 1 SP.3.2 (ovlp SP - p_chan_gate)
bool_8  bool_7 od_vln  0 0 0 0 4 SP.3.3+SN.4.3 (det_hor_connection)
bool_8  bool_7 od_vln  0 0 0 0 5 SP.3.3+SN.4.3 (det_ver_connection)
bool_8  bool_6 NOFILE 12 0 0 0 3 SP.3.3+SN.4.3 (ovlp OD - nwell_cont)
od_vln  sp_vln NOFILE  0 6 0 0 1 SP.4.1 (SP - OD)
sp_vln  bool_5 NOFILE  0 6 0 0 0 SP.4.2 (SP - n_chan_gate)
bool_0  bool_4 NOFILE  6 0 0 0 0 SN.3.1 (ovlp SN - OD)
bool_9  sn_vln NOFILE  6 0 0 0 0 SN.3.2 (ovlp SN - n_chan_gate)
bool_2  bool_7 od_vln  0 0 0 0 4 SN.3.3+SP.4.3 (det_hor_connection)
bool_2  bool_7 od_vln  0 0 0 0 5 SN.3.3+SP.4.3 (det_ver_connection)
bool_2  bool_6 NOFILE 12 0 0 0 3 SN.3.3+SP.4.3 (ovlp OD - substr_cont)
od_vln  sn_vln NOFILE  0 6 0 0 1 SN.4.1 (SN - OD)
od_vln  bool_9 NOFILE  0 6 0 0 0 SN.4.2 (SN - p_chan_gate)
bool_12 od_vln NOFILE  5 0 0 0 0 CON.3.1 (ovlp OD - CON)
bool_12 ps_vln NOFILE  0 5 0 0 0 CON.3.2 (CON - PS)
bool_13 sn_vln NOFILE  3 0 0 0 0 CON.3.3 (ovlp CON - SN)
sp_vln bool_13 NOFILE  0 3 0 0 0 CON.3.4 (CON - SP)
bool_14 od_vln NOFILE  5 0 0 0 0 COP.3.1 (ovlp OD - COP)
bool_14 ps_vln NOFILE  0 5 0 0 0 COP.3.2 (COP - PS)
```

```
bool_15 sp_vln NOFILE  3 0 0 0 0 COP.3.3 (ovlp COP - SP)
sn_vln bool_15 NOFILE  0 3 0 0 0 COP.3.4 (COP - SN)
bool_17 ps_vln NOFILE  4 0 0 0 0 CPS.4.2 (ovlp CPS - PS)
bool_17 od_vln NOFILE  0 5 0 0 0 CPS.4.3 (CPS - OD)
bool_19 in_vln NOFILE  3 0 0 0 0 IN.3.2 (ovlp CO - IN)
bool_20 in_vln NOFILE 10 0 0 0 0 CB.1.1 (ovlp CB - IN)
```

Each line of this file must contain the following items in the order given:

1. The first file involved with the operation. In case of overlap check this is the file of whose elements have to be overlapped.
2. The second file involved with the operation. In case of overlap check this is the file whose elements have to overlap the elements of the first file.
3. A helpfile involved in the operation. This file is used for checks with a certain kind. If not needed, 'NOFILE' is coded.
4. The overlap the second file must have over the first file. If it is zero, no overlap check will be carried out.
5. The minimal gap between non overlapping elements of the first and second file. If it is zero no check will be carried out.
6. The minimal gap that must be maintained if the length of the gap is only small.
7. The maximum gaplength for which the reduced gap value may be applied.
8. The value of the variable kind. For gap checks the value of kind means:

   0: do not suppress gap errors of overlapping items.
   1: suppress gap errors of overlapping items.

   For overlap checks the value of kind means:

   0: check for a total overlap.
   1: check for overlap over two opposite sides.
   2: only check the overlap for places where the helplay is not present.
   3: check only at the sides indicated by the conn_dir array. This array will be filled using checks with kind = 4 and kind = 5.
   4: sets the conn_dir array to 'check bottom and top overlap' if in the same polygon of the helplayer there is one area of the second layer present to the left and one to the right of an area of the first layer.
   5: sets the conn_dir array to 'check left and right overlap' if in the same polygon of the helplayer there is one area of the second layer present to the bottom and one to the top of an area of the first layer.
9. A string indicating which design rules are involved.

8. APPENDIX B: The program nbool

8.1 Introduction

The program *nbool* is the program that performs the logical operations between the masks of a cell.
The program must be called as:

```
nbool [-c|-n] [-f] [cell_name]
```

If *nbool* is called with the option **-c** the program will check the input for hierarchical errors; if option **-n** is given it will not. If *nbool* has to operate on input files that themselves are boolean combination files, the last option has to be chosen, otherwise false error messages will occur, because *nbool* then does not know what terminal masks are involved with the boolean masks. Default hierarchical checks are carried out.
If *nbool* is called with the option **-f** the 'current working directory' will be searched for the presence of a file *booldata*. If found this file will be taken as the technology_file for *nbool* instead of the standard one for the technology one is working in.
If a cell_name is given this given cell will be tested. If no cell_name is specified *nbool* looks for the file *exp_dat* in which the cell(s) to test then must be given.
So as its input the program needs:

- A file exp_dat containing the cell(s) the program has to be applied to, or a cell_name specified as argument.
- A file booldata containing the logical formulas the program has to perform upon its input files. This file is either taken from the library or from the 'current working directory'.
- The vln files (edge files) of the cell(s) involved in the logical combinations.

As its output the program generates the vln files of the combination masks of the formulas. Furthermore the program generates error messages when the rules about hierarchy are violated.

The main parts of the program are:

- The part that decodes the logical formulas given in the file booldata and makes a structure to check if a certain mask combination belongs to the formula given. This part will be described in the part about decoding of the design rules.
- The part that builds up and updates the stateruler. This will be described in the part about the stateruler.
- The part that analyses the stateruler for hierarchical errors. This will be described in the part about hierarchy check.

- The part that analyses the stateruler and determines from that what edges have to be output. This will be described in the part about extract_profile.
- The part that adds the group_numbers to the vln files made. This will be described in the part about add_groupnumbers.

8.2 Decoding the design rules

Most of the design rules involve more than one mask. To check these rules masks must be made containing logical combinations of the masks needed for that particular check. The formulas of the masks that must be made for all the checks of a certain technology must be given on the file booldata, which is described in appendix A. This section will describe the way these formulas are stored in memory to allow for an efficient way to produce all output masks wanted in one pass of the algorithm. In the program this is done in the routine mk_formstruct.
This routine starts by reading the first line of the file booldata which names all masks involved in the formulas to be made.
Then the routine ini_heap is entered which makes an input structure. For each file listed in the input line, and if the hierarchy must be checked also for each terminal file, a structure is set up containing:

- The name of the mask involved.
- The binary number of the mask. If the name of the mask is known to the process, the corresponding mask number is taken from it. If not it gets a number twice as high as the previous unknown mask, starting at the mask_number twice as high as the highest mask_number known in the process.
- The mask_type of the mask. If the mask is known to the process the mask_type is copied from it. So terminal masks become 1, connection masks become 2 and the others become 0. For masks unknown to the process the mask_type is set to BOLEAN (=3).
- The pointer to the vln file
- The data of the first edge in the vln file. So x_position,y_bottom, y_top, edge_type and check_type.

The latter data will be updated with a new line segment when the program has inserted the line segment in the stateruler.

After ini_heap has made this structure the procedure mk_formstruct starts reading the formulas, line by line. For each line it sets up a c_structure ’form’ which contains:

- The name of the file,where the edges of the mask to form must be written. This name is bool_xx, where xx is the form_number.

- A number of buffers to temporarily store the edge data before it is written to the output file.
- A number curr_place indicating which buffer has been filled last. Initially this variable is set to -1 to indicate that all buffers are free.
- A pointer to a list of min_term structures, which will be explained later on
- A vulnerability mask containing all masks present in the formula. This variable is not strictly needed, but added for efficiency reasons.
- A pointer to the next form_struct, or if there is not any a NULL_pointer.

The formula read now is decoded to fill this structure with its information. File names are detected from the file and also the special characters ! (negotiation) | (logical or) and & (logical and). Two masks are kept for each term of the formula:

- The masks that must be present for a mask combination to be part of the term of that formula.
- The masks that must NOT be present for a mask combination to be part of the term of that formula.

These variables are updated in the procedure 'update_masks' each time a '&' or '|' character is discovered. If a term of the formula is finished (a | character discovered or end of formula) these variables mask and not_mask are placed in a structure min_term and this structure is added to the list of min_term structures of the formula. This is done in the procedure 'add_minterm'. The variable vuln_mask of the form_structure there is updated too. Upon leaving the procedure 'mk_formstruct' we thus have created a structure like the one shown in figure 8.1

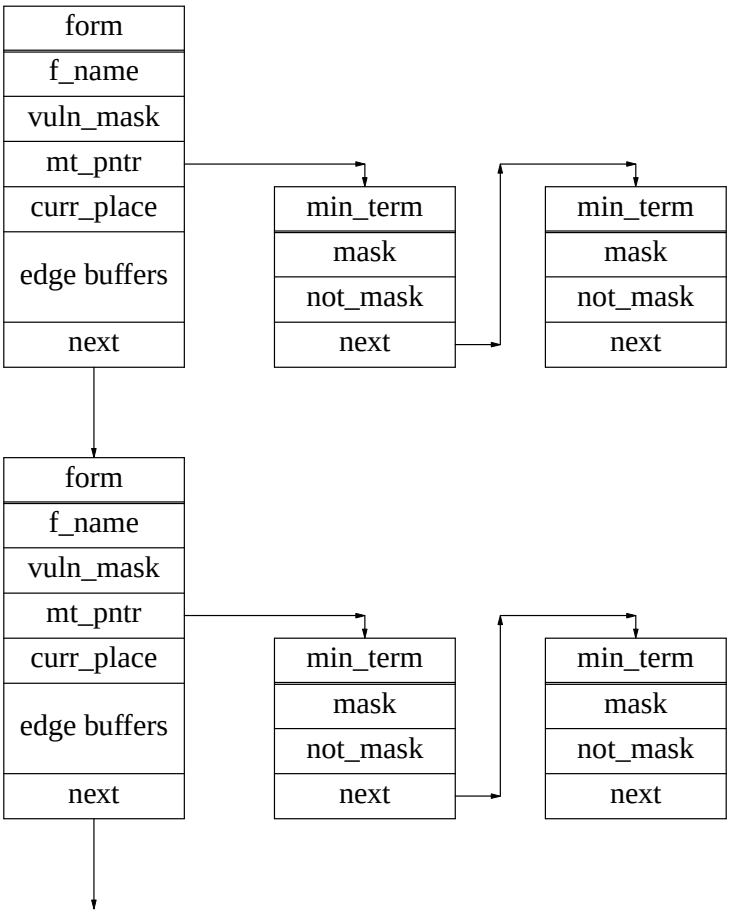

Figure 8.1. the formula structure

8.3 The stateruler

In this chapter the contents of the stateruler and the way it is formed and updated will be described. In this program the stateruler consists of fields with the following variables:

- yb: the bottom of the field.
- yt: the top of the field.
- chk_type: the check_type of the layers in the field. If the layers have different check_types this value is set to DIFF_CT (= -2).
- p_check: A pointer to a structure in which the check_types are stored per layer. If all layers have the same check_type there is no need for such a structure and p_check is a NULL_pointer.

- p_chg_ct: A pointer to a list of structures which contain the old check_type and the mask a change of check_type occurred in. If no change of check_type occurred this is a NULL_pointer.
- mask_past: This variable bitwise contains the layers present in the field before the edges at the present x_value are installed.
- mask_fut:This variable bitwise contains the layers that are present after the insertion of the edges at the present x_value.
- ov_mask: This variable contains bitwise the layers in which an overflow of layers of different check_type has occurred.
- next: A pointer to the next stateruler field.
- prev: A pointer to the previous stateruler field.

The stateruler is initialized to contain one field, from yb = -MAXINT to yt = MAXINT, with no masks present,so mask_past = mask_fut = ov_mask = 0 and chk_type set to INITIAL (=-1). The pointer to the check_type structure is set to NULL. Through the procedure 'select_edge' the edges that have to be inserted then are selected from the edge_heap in such a way, that the edges with the lowest x_coordinate come first and for edges with the same x_coordinate the one with the lowest bottom value comes first.
The procedure 'insert_edge' then inserts the edge in the stateruler. In this procedure the stateruler is scanned from the current field until the new edge and a field in the stateruler have an overlap. If this occurs, and the values of the bottom of the stateruler field and the bottom of the new edge do not coincide, the procedure 'split_field' is called, which splits the field in two parts, the split point being the bottom value of the new edge. The bottom and top values of the two created fields are updated and the other values of the old field are copied into the new field. The current stateruler pointer is set to the top field of the two fields being created/updated. As long as the top value of the next fields in the stateruler is not greater then the top value of the new edge, the fields in the stateruler are updated with information from the new edge. This is done in the procedure 'update_fld'. In this procedure the values of mask_fut, ov_mask and chk_type are updated, according to the values of the edge. If the top value of the stateruler field becomes smaller then the top value of the new edge a split is carried out with the procedure 'split_field' and the bottom field of the two newly created/updated fields is updated with the procedure 'update_fld'.
This process of selecting and inserting fields is continued until a new x_value is found. Then the stateruler (if this option is chosen) is checked for hierarchy errors and after that analyzed to extract the edges for the boolean files to be made. Then the stateruler is updated. This is done in the routine 'update_sr'. In this routine first the value of the mask_past in the fields are set to mask_fut, and the check_types of the stateruler fields are updated. After that fields containing the same values for the masks and check_types are joined.

After being updated a new stateruler is built for the next x_value until all edges have been read. A schema of the operations is given in figure 8.2

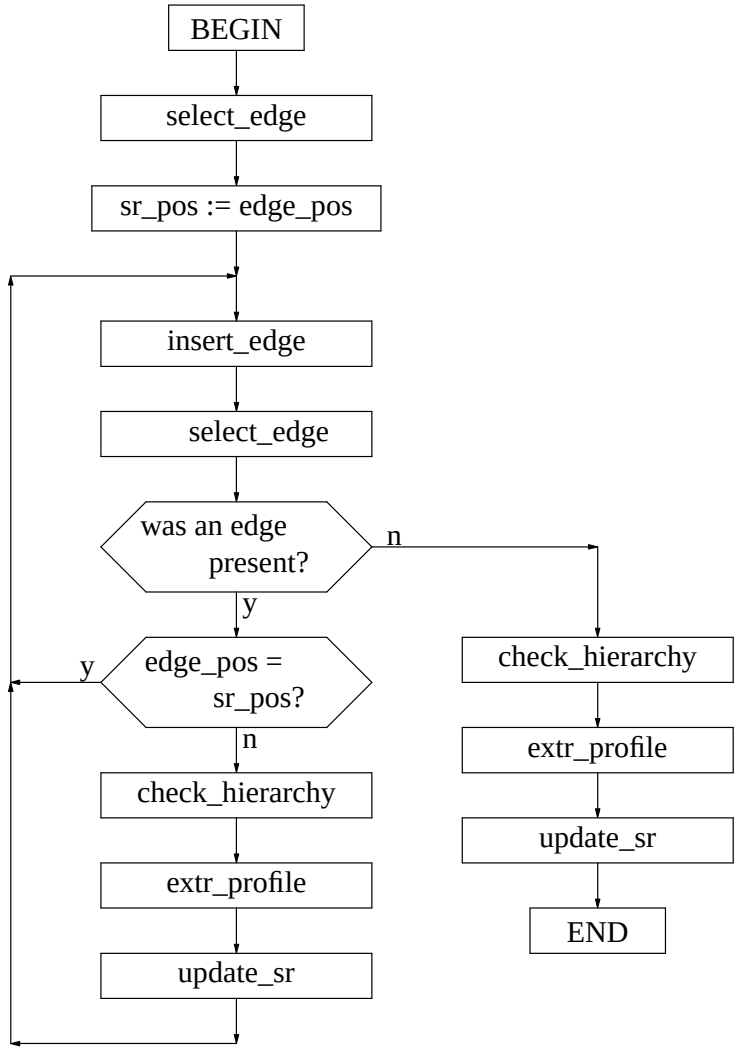

Figure 8.2. Stateruler main flow

8.4 The hierarchy check

The checking of the hierarchy rules is carried out in the procedure 'check_hierarchy'. This procedure contains a loop for checking all of the fields in the stateruler for:

- The ov_mask.
  If a bit in this mask is set, indicating that an overlap in the corresponding mask has occurred, the following is done:

— If the mask in which the overlap occurred is a connection mask, a check is carried out to see if the corresponding terminal mask is present. If not an error massage is generated, telling where the error occurred and in which mask.

— If the layer is not a connection mask a warning massage is generated, telling where the overlap took place and in which mask.

- The check_types.
  If the variable chk_type in the stateruler field is DIFF_CT, indicating that in the field layers with different check_types are present, the following actions are taken:
  A check is made to see if the difference is caused by a check_type 0 in a connection mask with the presence of a terminal in the same layer, indicating an overlap permitted. In this case no messages are generated. If the difference is not caused by the situation described above, a warning massage is generated telling the place where different checktypes occur, and the checktypes of the layers present.
- Change of checktype.
  If a change of checktype in the y_direction occurs the following steps are taken:
  - If the change takes place in a connection layer a check is carried out if a terminal is present there. If not, an ERROR message is generated, stating where the error occurred and in which layer.
  - If the change takes place in another layer a WARNING is generated, stating the place of the change of checktype and the layer it occurred in.

8.5 Analysis of the stateruler

The analysis of the stateruler, is carried out in the procedure 'extr_profile'. It finds the edges that have to be output in the vln file of the corresponding formulas. In this procedure a loop is set up, which examines each stateruler field. If the values of mask_past and mask_fut differ, indicating that one or more layers have changed state, the procedure 'buff_edge', which does the actual work is called. The main flow of this procedure is given in figure 8.3
The way the program checks if a certain mask combination belongs to a formula is done using the structure made with mk_formstruct. The mask combination is compared to the masks and not_masks of the min_terms of the formula. If all the masks present in the variable mask of the min_term structure are also present in the mask combination to check, and the masks set in the variable not_mask do not appear in the mask combination to check, the mask combination belongs to that min_term, and hence to the formula.
According to the presence of mask_past and mask_fut in the formula the pres_flag is set. If mask_past belongs to the formula and mask_fut does not, pres_flag is set to 1, indicating a stop edge. If mask_past does not belong to the formula and mask_fut does, pres_flag is set to -1, indicating a start edge. If mask_past and mask_fut both belong to the formula, or if they both do not belong to the formula, pres_flag is set to 0, indicating that no edge has to be output.

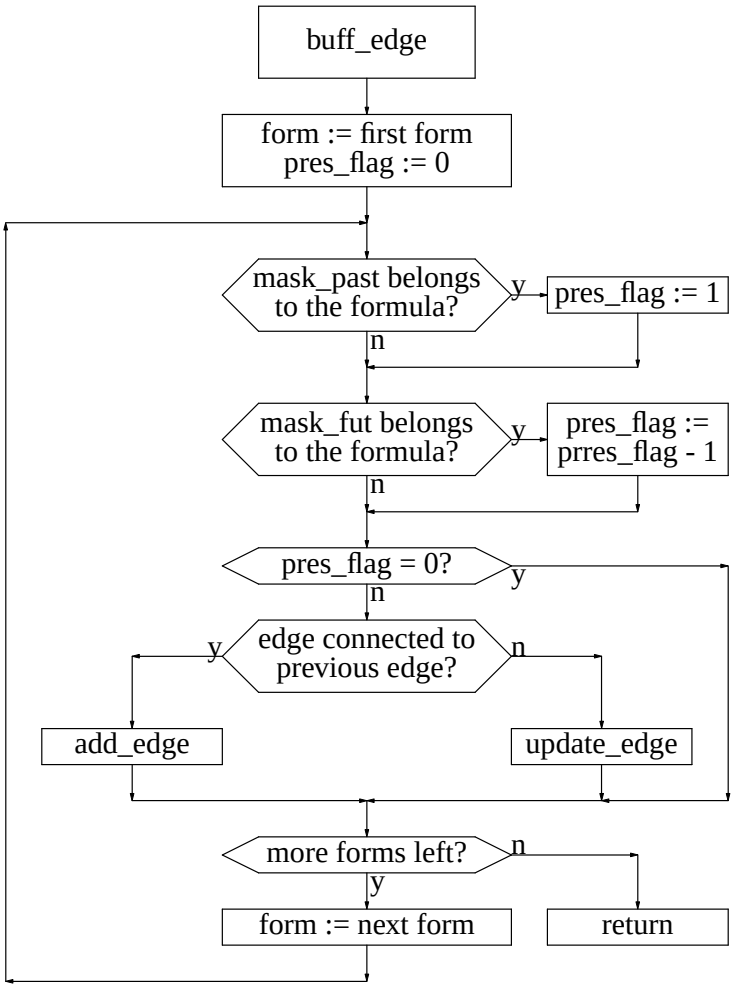

Figure 8.3. buff_edge main flow

After the value of the pres_flag is established, and the pres_flag = 0, no further actions are taken. If this value not equals zero, two cases may occur:

- The bottom value of the newly found edge and the top value of the last buffered edge of the formula are the same and so are their x_positions.
  In this case the last buffered edge is updated, i.e. its top value and its connection type are updated. This is done in the procedure 'update_edge'.
- If the values mentioned above do not coincide, the edge is added to the next place in the buffer. If all buffers of the formula have been filled, the buffer is appended to the file, whose name is given in the f_name variable in the form_structure. These actions are carried out in the procedure 'add_edge'.

8.6 The generation of the group_numbers(connectivity)

The group_numbers of the edges indicate to which connected region they belong. They are generated after *nbool* has generated the edges. This is done on temporary files, which for efficiency reasons are in binary format. They have a boolean name, with bt1 added to it. For example bool_2bt1.
Now the files generated are read one by one and a pointer structure is set up in the same way as it is done in i.e. the program *makevln*. The pointers are added to the file and written on a file with the addition of bt2 (e.g.bool_2bt2), and the bt1_file is removed from the system. The bt2 file then is read and the pointers are replaced by their corresponding group_numbers. Then the edges are written to the boolean file that remains in existence and is used by the programs *dimcheck* and *dubcheck*. These files (e.g bool_2) are in the known vln_format. The bt2_files are also removed from the system.

9. APPENDIX C: The program dimcheck

9.1 Introduction

The program *dimcheck* checks a cell for the presence of width or gap errors in a single (combination) layer.
The program is called as:

```
dimcheck [-a|d][-d][-f][-t][-g][cell_name]
```

The meaning of the options is:

-a The program is used as a part of the single_layer checker *autocheck*(see appendix E), and the file *dimcheckdata1* is taken as design_rule input_file.

-d The program is used as a part of the multi_layer checker *dimcheck*(see appendix E), and the file *dimcheckdata2* is taken as design_rule input_file.

-f The program looks for the file *dimcheckdata1* (or *dimcheckdata2*) in the current working directory instead of taking the standard one for the technology used.

-t This option has been added for debugging purposes. It generates a lot of test_data.

-g With this option gap_errors within the same polygon, which otherwise may be suppressed, are always reported.

cell_name The name of the cell to be tested. If not specified the program looks for a file *exp_dat* in which the cell(s) to be tested must be given.

So as its input the program needs:

- A file *exp_dat* containing the cell(s) *dimcheck* has to be applied to, or a cell_name as argument in the call of the program.
- A file *dimcheckdata1* (or *dimcheckdata2*) containing the layers to check and the gaps and widths permitted.
- The vln files of the cell(s) to be tested.

As its output *dimcheck* generates error messages on the terminal, stating the rule that was violated and the place where the error occurred.

The program may be divided into two mayor parts:

- One part consisting of the building and updating of the stateruler.

- A second part consisting of the analysis of the stateruler and the generation of the error messages from it.

These two parts will be discussed in the sections ’Making and updating the stateruler’ and ’The analysis of the stateruler’ respectively.

9.2 Making and updating the stateruler

In this sections the contents of the stateruler fields and the way they are formed and updated will be described.
In *dimcheck* the fields of the stateruler hold the following variables:

- xstart: The x_position in which the the field was started.
- yb: The bottom of the field.
- yt: The top of the field.
- lay_status: The status of the layer. This may be:
  - NOT_PRESENT: This means that the layer is not present at the stateruler position.
  - CHG_TO_PRESENT: This means that the layer starts at the stateruler position.
  - CHG_TO_NOTPRESENT: This means that the layer stops at the stateruler position.
  - PRESENT: This means that the layer is present at the stateruler position.
- helplay_status: The status of the helplayer if used.
- group: The group in the layer the field belongs to.
- group_old: The group of the edge before the last one.
- chk_type: The checktype of the layer in the field.
- chk_type_old: The checktype of the edge before the last one.
- next: A pointer to the next stateruler field.
- prev: A pointer to the previous stateruler field.

Upon initiation the stateruler consists of one field, reaching from -MAXINT to MAXINT, with lay_status NOT_PRESENT , xstart = -MAXINT, next and prev pointing to the field itself and the other variables set to zero.
After the initiation a loop is started in which edges are read from the vln file(s) and inserted into the stateruler (procedure insert_edge). The loop is continued until all edges with the same x_value have been inserted. The stateruler for that x_value then is completed, and an analysis of the stateruler then will take place (procedure extr_profile).

The stateruler is updated (procedure update_sr) and new vln files are read and inserted to form the stateruler for the next x_position. This process is repeated until all edges have been read from the vln file.
The process described above is carried out in the procedure main_check.
As stated above the insertion of new edges in the stateruler is done in the procedure insert_edge. In this procedure the fields of the stateruler are scanned from the current position to the topmost position to see if an overlap with the edge to insert is present. If this is the case and the bottom values of the field and the new edge do not coincide, the stateruler field is split into two and the variables are copied from the old field. As long as the top value of the edge is greater then the top value of the stateruler fields, the latter are updated. If the top value of the new edge becomes smaller as the top value of the stateruler field, again a split is carried out and the bottom field of the two newly created fields is updated. The splitting of the fields is carried out in the procedure 'split_fld', the updation of the fields is done in the procedure 'update_fld'.
An example is shown in figure 9.1.

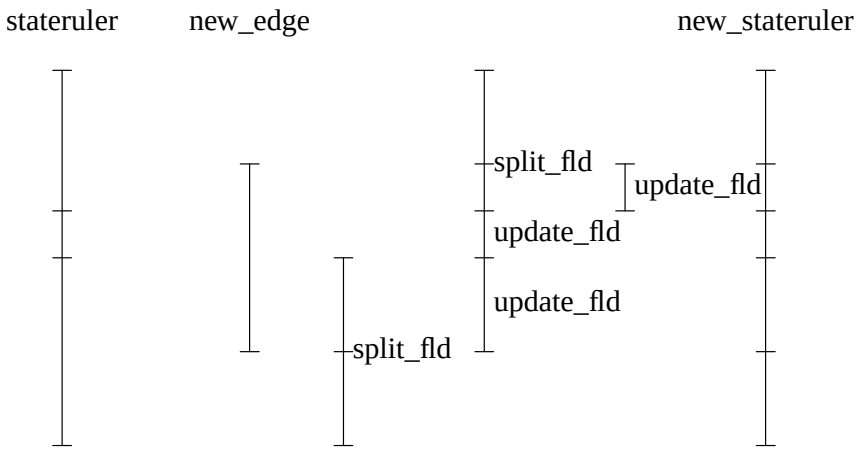

Figure 9.1. Building the stateruler

In the procedure 'update_sr' the stateruler is updated after being analyzed. This means :

— The lay_status is updated: CHG_TO_PRESENT becomes PRESENT and CHG_TO_NOTPRESENT becomes NOT_PRESENT.

— The group_nbr, check_type and xstart are updated

— If possible stateruler fields are merged.i.e:
If two adjacent fields have:

  — the same checktype

— the same group_nbr

— the same lay_status

— the same xstart or for both fields holds stateruler position - xstart >= MAXINFLUENCE

the two fields are merged.

9.3 The analysis of the stateruler

The analysis of the stateruler to detect possible design rule errors is done in the procedure 'extr_profile'. In this procedure all fields of the stateruler are checked for possible design rule errors. According to the lay_status the following checks are carried out:

- lay_status = PRESENT.
  This means that no change of lay_status has taken place, so nothing needs to be checked.
- lay_status = CHG_TO_PRESENT.
  This means that a new area has started. In this case the following checks are carried out:
  — A check to see if the distance between the previous edge and the new edge is great enough (procedure 'check_xgap').
  — If in the previous stateruler field the layer is not present a check to see if previous edges are not too close to the bottom of the new edge (procedure 'check_g_circle').
  — If in the next stateruler field the layer is not present a check to see if previous edges are not too close to the top of the new edge (procedure 'check_g_circle').
  — If in the previous stateruler field the lay_status is not CHG_TO_PRESENT and in the next stateruler field the lay_status is not PRESENT (in which cases an error, if any, already has been reported), a check of the y_width of the edge starting in the stateruler field is carried out (procedure 'check_ywidth').
- lay_status = CHG_TO_NOTPRESENT
  This means that an area has stopped. In this case the following checks are done:
  — A check to see if the area that stopped was not too small in the x_direction (procedure 'check_xwidth').
  — If the previous stateruler field the lay_status is not CHG_TO_NOTPRESENT (in which case an error, if any, already has been reported) and in the previous stateruler field the lay_status is PRESENT a check is carried out to see if the area that is left under the stop is not too small in the y_direction (procedure 'check_ywidth').

— Under these conditions also a check is done to see if the layer is present in a circular area around the bottom of the stateruler field (procedure ’check_w_circle’).

— If the layer is not present in the previous stateruler field a gap check is done to see if the gap between the stopped area and the first area below it is not too small (procedure ’check_ygap’).

— If the next stateruler field the lay_status is not CHG_TO_NOTPRESENT (in which case an error, if any, already has been reported) and in the next stateruler field the lay_status is PRESENT a check is carried out to see if the area that is left above the stop is not too small in the y_direction (procedure ’check_ywidth’).

— Under these conditions also a check is done to see if the layer is present in a circular area around the top of the stateruler field (procedure ’check_w_circle’).

— If the layer is not present in the next stateruler field a gap check is done to see if the gap between the stopped area and the first area over it is not too small (procedure ’check_ygap’).

- lay_status = NOT_PRESENT.
  This means that no change of lay_status has taken place, so nothing needs to be checked.

The width_checks mentioned will only be carried out if according to the file *dimcheckdata1(2)* the width_flag is set. The gap_checks are only carried out if the gap_flag is set and if the helplay_status is NOT_PRESENT, if a helplayer is specified.
In the check routines check_xwidth etc. gap and width errors will not be generated if the edges have the same checktype (except if it is zero). This situation means that the error originates from a subcell and has already been reported there.

10. APPENDIX D: The program dubcheck

10.1 Introduction

The program *dubcheck* is the program that does the checking for overlap and gap errors between two (combination)masks.
It is called as:

```
dubcheck [-f][-t][cell_name]
```

The meaning of the options is:

-f The program looks for the file *dubcheckdata* in the current working directory instead of taking the standard one from the technology used.

-t This option has been added for debugging purposes. In generates a lot of test_data.

cell_name The name of the cell to be tested. If not specified *dubcheck* looks for a file *exp_dat* in which the cell(s) to be tested must be given.

So as its input the program needs:

- A file *exp_dat* containing the cell(s) *dubcheck* has to be applied to, or a cell_name must be given in the call of the program.
- A file dubcheckdata containing the layers to check and the overlaps and gaps permitted.
- The vln files of the cell(s) to be tested.

As its output *dubcheck* generates error messages on the terminal, stating the rule that was violated and the place where the error occurred.

The program may be divided into two mayor parts:

- One part consisting of the building and updating of the stateruler.
- A second part consisting of the analysis of the stateruler and the generation of the error messages from it.

These two parts will be discussed in the chapters ’Making and updating the stateruler’ and ’The analysis of the stateruler’ respectively.

10.2 Making and updating the stateruler

In this chapter the contents of the stateruler fields and the way they are formed and updated will be described.

In *dubcheck* the fields of the stateruler hold the following variables:

- xstart[0]: The x_position of the previous edge of mask1 in the field.
- xstart[1]: The x_position of the previous edge of mask2 in the field.
- yb: The bottom of the field.
- yt: The top of the field.
- lay_status[0]: The status of mask1. This may be:
  - NOT_PRESENT: This means that the layer is not present at the stateruler position.
  - CHG_TO_PRESENT: This means that the layer starts at the stateruler position.
  - CHG_TO_NOTPRESENT: This means that the layer stops at the stateruler position.
  - PRESENT: This means that the layer is present at the stateruler position.
- lay_status[1]: The status of mask2.
- helplay_status: The status of the helplay.
- group[0]: The group of mask1 in the stateruler field.
- group[1]: The group of mask2 in the stateruler field.
- chk_type[0]: The checktype of mask1 in the stateruler field.
- chk_type[1]: The checktype of mask2 in the stateruler field.
- next: A pointer to the next stateruler field.
- prev: A pointer to the previous stateruler field.

Upon initiation the stateruler consists of one field, reaching from -MAXINT to MAXINT, with lay_status NOT_PRESENT , xstart = -MAXINT, next and prev pointing to the field itself and the other variables set to zero.
After the initiation a loop is started reading edges from the vln files (procedure get_vln), selecting the one with the smallest x_value and the smallest value of y_bottom and inserting them into the stateruler (procedure insert_edge). The loop is continued until all edges with the same x_value have been inserted. The stateruler for that x_value then is completed, and an analysis of the stateruler then will take place (extr_*** procedures). The stateruler is updated (procedure update_sr) and new vln files are read and inserted to form the stateruler for the next x_position. This process is repeated until all edges have been read from the vln file.
The process described above is carried out in the procedure main_check.
As stated above the insertion of new edges in the stateruler is done in the procedure

insert_edge. This procedure is similar to the one used in the program *dimcheck*, with only a difference in the variables that are present in the stateruler fields.

In the procedure 'update_sr' the stateruler is updated after being analyzed. This means :

- The lay_status is updated: CHG_TO_PRESENT becomes PRESENT and CHG_TO_NOTPRESENT becomes NOT_PRESENT in lay_status[0] , lay_status[1] and helplay_status.
- The group_nbr, check_type and xstart are updated for both masks.
- If possible stateruler fields are merged.i.e:
  If two adjacent fields have:
  - the same checktype for both masks
  - the same group_nbr for both masks
  - the same lay_status for both masks
  - the same xstart or for both fields holds stateruler position - xstart >= MAXINFLUENCE for both masks

  the two fields are merged.

10.3 The analysis of the stateruler

The analysis of the stateruler to detect possible design rule errors is done in the procedures 'extr_profile' , 'extr_overlap' , 'extr_overlap1' , 'extr_overlap2' and 'extr_overlap3'. The first procedure is used to detect gap errors, the last ones to detect overlap errors.

10.3.1 detection of gap errors

In the procedure extr_profile all fields of the stateruler are checked for possible gap errors. According to the lay_status of the masks the following checks are carried out:

- lay_status[0] = CHG_TO_PRESENT.
  This means that an area in mask1 is starting. In this case the following checks are done:
  - A check is carried out to see if mask1 and mask2 do have an overlap here. In this case the status of mask2 is PRESENT or CHG_TO_PRESENT. No error exists then and a structure is set up, to indicate that the group of the item in mask1 and the item in mask2 have an overlap. If other errors occur between these equivalent groups of mask1 and mask2 they will be suppressed is this is wanted.
  - If the masks have no overlap the distance to the last recorded edge of mask2 in the field is checked.
  - If lay_status[0] of the previous or next field in the stateruler is NOT_PRESENT checks are carried out to see if no error exists in the areas left under respectively

left above the edge.

- lay_status[1] = CHG_TO_PRESENT.
  This means that an area in mask2 is starting. In this case the same checks are carried out with respect to mask1, as in the previous case with respect to mask2.
- lay_status[0] = CHG_TO_NOTPRESENT
  This means that an area in mask1 has stopped. In this case the following checks are done:
  - A check is carried out to see if mask1 and mask2 do have an overlap. An equivalence of groups then is set up again.
  - If no overlap occurs a check is carried out to see if no error occurs at the bottom of the field and a check is carried out to see if no error occurs at the top of the field.
- lay_status[1] = CHG_TO_NOTPRESENT.
  This means that an area in mask2 has stopped. In this case the same checks are carried out with respect to mask1, as in the previous case with respect to mask2.

In the check routines no errors will be reported between two edges if they have the same check_type,and this checktype does not equal zero, indicating that the edges stem from the same instance of a subcell. If the errors exist,they will be reported when the subcell is checked. If the variable kind is made zero, also no errors between areas of mask1 and mask2 that have an overlap will be reported. Else these errors will be reported.
The errors found in this case are not immediately shown, but temporarily stored first. In this way one can suppress purely geometric errors, which turn out to be unimportant when connectivity is taken into account (this is an important topic in hierarchical design, because the design rule checker output often gets clothered with unimportant 'faults' obstructing the really important messages).

10.3.2 detection of overlap errors of kind 0

In the procedure extr_overlap checks are carried out to see if all areas of the first layer are fully overlapped by a distance overlap by the areas of layer 2. According to the lay_status of the masks in the stateruler fields the following checks are carried out:

- lay_status[0] = CHG_TO_PRESENT. In this case the next checks are carried out:
  - A check to see if lay_status[1] is PRESENT. If not an error is recorded.
  - If PRESENT a check to see if the stop of the last area in mask2 in the stateruler field is at least a distance of overlap smaller then the position of the stateruler.
  - Checks to see if the areas left under the bottom of the edge and left upper of the top of the edge are covered by mask2.

- lay_status[1] = CHG_TO_NOTPRESENT. In this case the next checks are carried out:
  - A check to see if lay_status[1] is NOT_PRESENT. If not an error is recorded.
  - A check to see if no area of mask1 is present over a distance of overlap before the stop of mask2.
  - If not a check to see if mask1 is not present over a distance of overlap under or above the edge.
  - A check to see if mask1 is not present left under the top of the edge or left above the bottom of the edge.

Errors are reported immediately in this case. The check procedures are such that no errors are generated if the area of mask1 in the stateruler field has a checktype not equal zero, indicating it originates from a subcell. In this case the error will already be detected when the subcell is checked.

10.3.3 detection of overlap errors of kind 1

In the procedure extr_overlap1 checks are carried out to see if all areas of the first layer are overlapped by a distance overlap by the areas of layer 2 in the x_ or y_direction. According to the lay_status of the masks in the stateruler fields the following checks are carried out:

- lay_status[0] = CHG_TO_PRESENT. In this case the next checks are carried out:
  - if lay_status[1] = PRESENT, a check is carried out to see if the overlapping area started at least a distance overlap earlier.
  - if lay_status[1] = CHG_TO_PRESENT, checks are carried out to see if the overlaps over the starting area of layer1 to the top and bottom are great enough.
  - if lay_status[1] = NOT_PRESENT or CHG_TO_NOTPRESENT an error is generated.
- lay_status[1] = CHG_TO_NOTPRESENT. In this case the next checks are carried out:
  - if lay_status[0] = NOT_PRESENT, a check is carried out if the stop edge of the area to be overlapped has occurred at least a distance overlap before.
  - if in the previous stateruler field lay_status[1] = PRESENT and lay_status[0] = NOT_PRESENT, a check is carried out to see if over a distance of at least overlap under the stateruler field no area in mask1 is present.
  - if in the next stateruler field lay_status[1] = PRESENT and lay_status[0] = NOT_PRESENT, a check is carried out to see if over a distance of at least overlap over the stateruler field no area in mask1 is present.

Errors are reported immediately in this case. The check procedures are such that no errors are generated if the area of mask1 in the stateruler field has a checktype not equal zero, indicating it originates from a subcell. In this case the error will already be detected when the subcell is checked.

10.3.4 detection of overlap errors of kind 2

In the procedure extr_overlap2 checks are carried out to see if all areas of the first layer are overlapped by a distance overlap by the areas of layer 2 at places where the helplayer is not present. According to the lay_status of the masks in the stateruler fields the following checks are carried out:

- lay_status[0] = CHG_TO_PRESENT. In this case the next checks are carried out:
  - if helplay_status != PRESENT a check is carried out to see if the overlapping started at least a distance overlap earlier.
  - if in the previous stateruler field lay_status[0] = NOT_PRESENT and the helplay_status is NOT_PRESENT of CHG_TO_NOTPRESENT here, a check is carried out to see if the overlap to the bottom is large enough.
  - if in the next stateruler field lay_status[0] = NOT_PRESENT and the helplay_status is NOT_PRESENT of CHG_TO_NOTPRESENT here, a check is carried out to see if the overlap to the top is large enough.
- lay_status[1] = CHG_TO_NOTPRESENT and helplay_status != PRESENT. In this case the next checks are carried out:
  - A check to see if the mask to overlap does not exist at least for a distance overlap before the x_position of the overlapping edge.
  - If in the previous stateruler field lay_status[1] = PRESENT a check is carried out to see if the overlap of layer[1] to the bottom of layer[0] is large enough.
  - If in the next stateruler field lay_status[1] = PRESENT a check is carried out to see if the overlap of layer[1] to the top of layer[0] is large enough.

Errors are reported immediately in this case. The check procedures are such that no errors are generated if the area of mask1 in the stateruler field has a checktype not equal zero, indicating it originates from a subcell. In this case the error will already be detected when the subcell is checked.

10.3.5 detection of overlap errors of kind 3

In the procedure extr_overlap3 overlap checks are carried out in accordance to the direction given in the con_dir array. The latter is initialized if an overlap check with kind = 4 or kind = 5 is given. According to the lay_status of the masks in the stateruler fields the following checks are carried out:

- lay_status[0] = CHG_TO_PRESENT. In this case the next checks are carried out:

  - If the direction is (BOTTOM + TOP) a test is carried out to see if the overlap to the left of the area to overlap is large enough.
  - If the direction is (LEFT + RIGHT) tests is carried out to see if the overlaps to the bottom and the top are large enough.

- lay_status[1] = CHG_TO_NOTPRESENT. In this case the next checks are carried out:
  - If the direction is (BOTTOM + TOP) a test is carried out to see if the overlap to the right of the area to overlap is large enough.
  - If the direction is (LEFT + RIGHT) tests is carried out to see if the overlaps to the bottom and the top are large enough.

Errors are reported immediately in this case. The check procedures are such that no errors are generated if the area of mask1 in the stateruler field has a checktype not equal zero, indicating it originates from a subcell. In this case the error will already be detected when the subcell is checked.

10.3.6 setting of the con_dir array

The setting of the con_dir array needed for overlap checks with kind = 3 is done in the procedures det_conn_hor and det_con_ver.

Det_con_hor adds to the appropriate entry in the array conn_arr the value LEFT if, under presence of the same polygon of the helplay, an item of the second given file is present to the left of the item of the first given file under consideration. It adds the value RIGHT if under the same conditions an item of the second file is to the right of the item of the first file.

Det_con_ver adds to the appropriate entry in the array conn_arr the value BOTTOM if, under presence of the same polygon of the helplay, an item of the second given file is present to the bottom of the item of the first given file under consideration. It adds the value TOP if under the same conditions an item of the second file is to the top of the item of the first file.

The procedures det_con_hor and det_con_ver are performed if a check with resp. kind = 4 and kind = 5 is present in the file *dubcheckdata*. To perform an overlap check of kind = 3, the checks with kind = 4 and kind = 5 must be performed first.

References

1. M. Newell and D.T. Fitzpatrick, ‘‘Exploitation of Hierarchy in Analysis of Integrated Circuit Artwork,’’ *IEEE Trans. on CAD* **CAD-1**(4) pp. 192-200 (Oct. 1982.).

2. J.T. Fokkema and T.G.R. van Leuken, ‘‘An efficient datastructure and algorithm for VLSI artwork verification,’’ *Proc. IEEE ICCD-83*, New York, pp. 350-353 (Oct. 1983).

CONTENTS

LIST OF FIGURES

LIST OF TABLES