MASTER

COMPAC : Compaction of hierarchical chip layouts based on a constraint graph method with double-sided constraints

Timmermans, X.I.M.

Award date:
1983

547

AFDELING DER ELEKTROTECHNIEK
TECHNISCH HOGESCHOOL
EINDHOVEN

VAKGROEP AUTOMATISCH SYSTEEMONTWERPEN

COMPAC

Compaction of hierarchical chip layouts based on a constraint graph method with double-sided constraints

X.I.M. Timmermans

Rapport van het afstudeerwerk
uitgevoerd van juni 1982 tot mei 1983
in opdracht van prof.dr.ing. J. Jess
en ir. M. van der Woude

308814

CONTENTS

Summary 3

1. Introduction 4

2. Methods for compaction 6
2.1. Shear line or compression ridge compaction 6
2.2. Virtual grid compaction 7
2.3. Critical path compaction 8

3. Objectives 11
3.1. Hierarchy 11
3.2. Representation of layout elements 13
3.3. Interactivity 15
3.4. Design rules 16

4. The compaction process 19
4.1. Overview 19
4.2. Input the layout elements 20
4.3. Sort the elements and search for overlap 22
4.4. Partition the elements into features 28
4.5. Generate the constraint graph 30
4.6. Determine an ST ordering of the nodes of the graph 33
4.7. Compac 34
4.8. Construction of the compacted compound cell 37
4.9. Examples 37

5. Extensions 43

6. Conclusion 44

References 45

Appendix: Program Documentation (not included)

SUMMARY

In this report the compaction program Compac is described. Compaction is a useful tool in interactive computer-aided generation of layouts of integrated circuits. With a layout editor the layout designer can construct a layout in a fast way by approximate placement of the layout elements. With automatic compaction and design rule checking he can generate a compact layout with no design rule violations.

The compaction method applied in Compac is based on a constraint graph. Compaction is achieved in one ore more subsequent steps by grouping the layout elements into features and then shifting these features together in horizontal or vertical direction. The nodes and edges of the graph represent the features and the constraints between the features.

The compaction algorithm presented can handle both one-sided constraints and two-sided constraints. The two-sided constraints are resolved by backtracking. Constraints on the shift of layout elements which overlap each other are determined with a linear method. For determination of constraints of non-overlapping elements windowing and shadowing techniques are used to reduce computation time.

1. INTRODUCTION

The compactor described in this report is a part of the Interactive Design System (IDS) for integrated circuits [1]. The system will be used for the IC production unit at the THE. The participants in the design of IDS are the researchgroup ES of the department of EE and the computing centre of the THE. They co-operate under NELSIS [14] with the universities of technology of Delft and Twente. Further it is intended to co-operate with Philips Physical Laboratory.

Portability of circuit and layout designs is one of the requirements. Therefore standards such as standardized design description languages will be defined. This enables the spread-out of a large layout project over several centres, as well as the use of pieces of layouts which are develloped in other centres and stored in local libraries. The hierarchical approach in IDS facilitates the editing of layouts and all other phases in the design of layouts for VLSI.

The design system IDS will encompass in the near future the following modules:

1. PL-simulator, a mixed-mode mixed-level simulator based on Piecewise Linear Modelling, which can simulate integrated circuits at circuit level, logic level and behavioural level [2].
2. ISLE, an Interactive Symbolic Layout Editor for editing and drawing symbolic layouts with a graphics terminal or digitizer [3].
3. Colormask, a geometric layout editor, for editing pieces of geometric layout which can be used as leave cells in ISLE [4].
4. Compaction [11].
5. Design rule checking.
6. Circuit verification.
7. Circuit extraction.

The modules will partly use common databases. Besides interactive input via terminal or graphics tablet, textual input and output with description languages will be possible.

The program COMPAC has been develloped on the PDP-11 of the researchgroup ES. The language used is Fortran 4. The Burroughs version of ISLE has been implemented on the PDP-11, to assure good interaction between COMPAC and ISLE. This has been done in co-operation with student A.G.J. Slenter.
Main differences between the B7700-version of ISLE and

the PDP-version are:

- On the PDP an overlay table is needed;
- The drawing with GINO-routines is on the PDP done in
  a separate process.

2. METHODS FOR COMPACTION

All the various methods for compaction use a topological placement of layout elements as starting point. Due to the complexity of the layout compaction problem, all published algorithms perform compaction in seperate steps in horizontal and vertical direction. Compaction algorithms introduce only marginal changes in the topology of the layout. Presently attempts are made to use the concept of simulated annealing for two-dimensional compaction algorithms [5].

In this chapter brief descriptions are given of compaction methods which are or have been used in several layout design systems. Chapter 4.1. gives an overview of the compaction process used in the IDS system; the compaction algorithm is presented in chapter 4.7.

2.1. Shear line or compression ridge compaction

One of the first compaction algorithms published is based on a coarse grid (fixed grid) method (Akers [6]). The layout elements are represented as entries in a matrix. Compaction is achieved by searching for and eliminating a path of blank cells across the matrix (fig.2.1.a). Figure 2.1.b gives the example used by Akers.

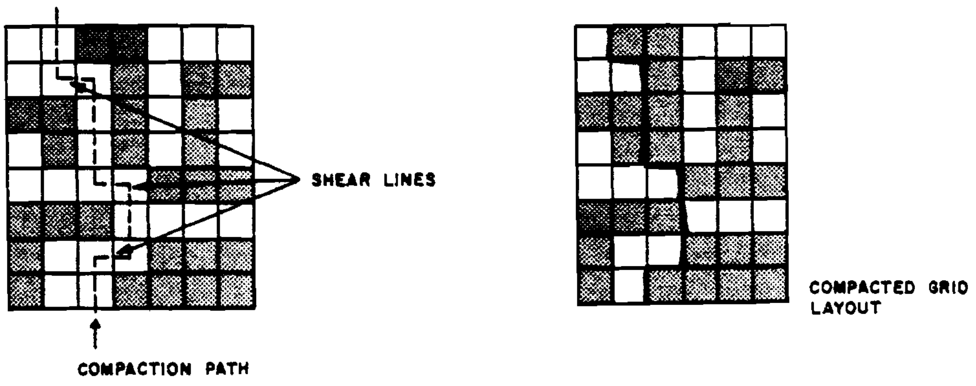


Fig.2.1.a Shear line compaction

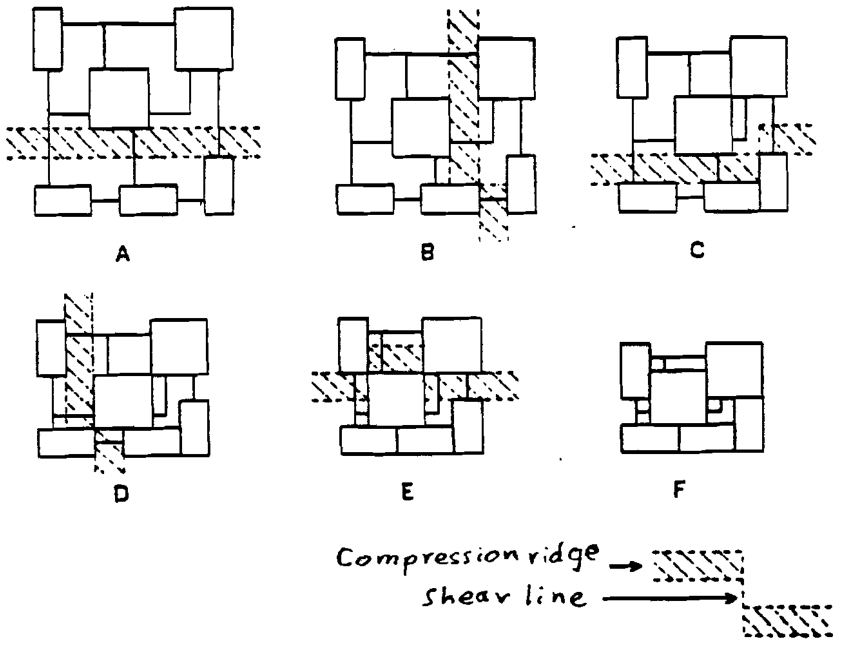


Fig.2.1.b Example of shear line compaction used by Akers

The algorithm for finding compression ridges often runs into dead ends, which causes time-consuming backtracking.

Dunlop [7] uses the shear line method in the SLIM-system during the global compaction phase. In this system a relative grid symbolic layout method is used. The global compaction is preceded by a local compaction phase in which a kind of clustering of layout elements takes place based on a critical path method such as in the CABBAGE-system [8].

2.2. Virtual grid compaction

This method is used in the MULGA-system (Weste [9]). A relative grid approach is used in order to avoid waste of mask space inherent to the fixed grid compactors such as mentioned above. However, to minimize design rule type calculations, adjacency information is treated on grid basis.

The symbolic layout is drawn on a virtual grid. The layout elements are all placed on virtual grid lines.

During compaction the spacing between two adjacent grid lines is determined based on spacing requirements of the elements which are resident on these grid lines. So all elements on the same vertical virtual grid line will shift over the same horizontal distance. After compaction all spacings has been determined, and the virtual grid is in fact transformed into a geometrical grid.

2.3. Critical path compaction

The critical path method (also known as the longest path method) originates from research in the field of operations research and network planning. In the field of layout compaction this method is (o.a.) applied by Hsueh in the CABBAGE-system [8].

The layout is represented by a constraint graph. There are various ways to map the layout into the graph.

One way is to map the edges of elements (building blocks and interconnection lines) into nodes, and the sizes and spacing requirements into branches (fig.2.2).

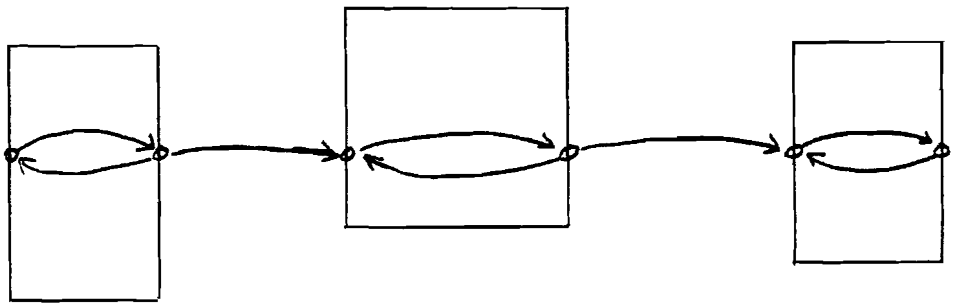


Fig.2.2 An example of a graph representation of a layout

In the Cabbage system another mapping has been applied. In this system, those layout elements that are fixed with respect to each other in the direction of compaction, are partitioned into groups (features). All topologically connected elements sharing the same vertical or horizontal center line will shift as a single block during compaction. As a consequence of this approach the groups are mapped into nodes and the separation requirements between groups are mapped into branches (fig.2.3). So the weight assigned to the branch represents the constraint between two groups. Note that the sizes of elements are not mapped into the graph.

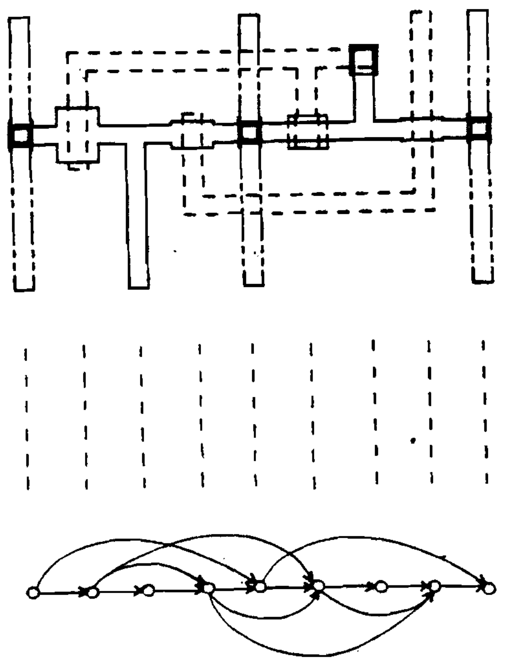


Fig.2.3 Graph representation for horizontal compaction in Cabbage.

The graph is a-cyclic and can be resolved with a single-pass algorithm. For horizontal and vertical compaction, different graphs are used. As depicted in fig.2.4 the result of compaction may be improved by allowing jogs in straight lines. Insertion of a jog means splitting of a straight line into two parts connected by a perpendicular line of initially zero length. In the constraint graph this implies the splitting of one node into two. This splitting is only useful for nodes on a critical path. A critical path is defined as a path from the graph source to the sink, where a change of the position of an arbitrary node on the path results in a change of the position of the sink. If a node on the critical path is split, the path no longer goes from source to sink. Another critical path that is shorter will turn up. Automatic jog insertion implies the recalculation of the critical path each time a jog is inserted.

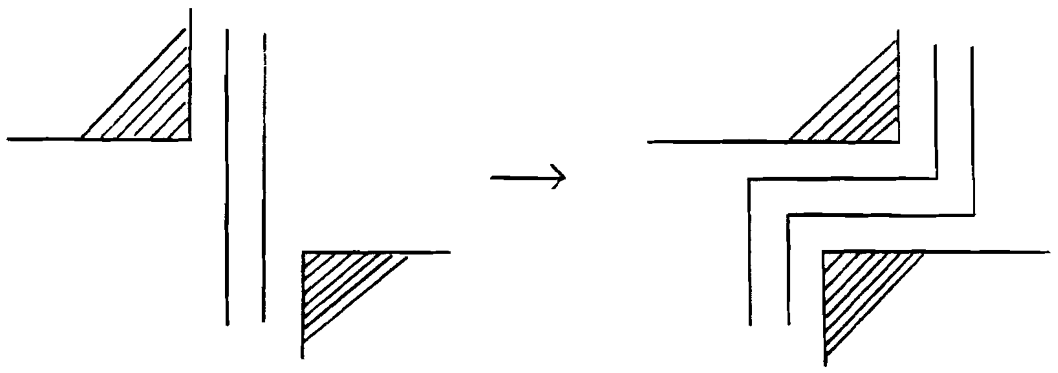


Fig.2.4 Jog insertion caused by the "torque" applied by neighbouring structures on the critical path.

3. OBJECTIVES

In this chapter the concepts of hierarchy and interactivity as implemented in the IDS-system and the influence of these concepts on the design of the compactor are discussed. An introduction to the symbolic "stick-based" representation of a layout which serves as input for the compactor is given in section 3.2.

3.1. Hierarchy

An hierarchical approach for design systems for very large integrated circuits is very useful [10].

The advantages of such an approach are clear:
- the designer will keep survey over what he is doing since the number of details at a particular hierarchical level can be kept within the limits of the human mind.
- standard cells (gates, flip-flop, register, multiplier) have to be designed only once and can be instantiated at several places in the same or another IC-layout.
- storage requirements are reduced by storing the definition of each cell only once.
- layout editing will be easier since instances of cells of a lower hierarchical level can be shifted, rotated and mirrored as a single block.
- computational tasks such as network verification compaction and design rule checking will run much faster since their scope can be limited to a single hierarchical level.

However, there are some difficulties and disadvantages which arise with an hierarchical approach:
- advantages disappear after module expansion
- some loss in efficiency of chip area due to necessary domain separation
- restricted freedom in design
- extra efforts must be made to generate a well structured hierarchical separation which can be used both for the network description and the layout description. In general the lower echelons in the hierarchy will differ.
- decisions must be made wether (domain-) separation between hierarchical levels or easy editing should have prevalence. This is illustrated in fig. 3.1. It is easier to draw long global interconnections which make contact at several places,

than to draw a great number of small line-pieces.

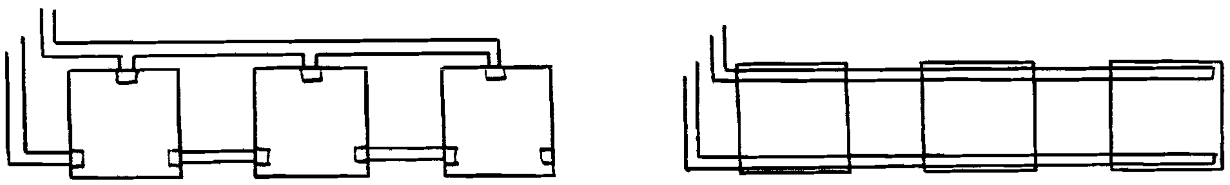


Fig. 3.1 Strong hierarchical separation versus simple editing

However, if wires are allowed to overlap instances of compound cells, the scope of design rule checking and compaction must be extended to more than one hierarchical level. Or, alternatively the designer must use a different layer for global interconnections (e.g. metal) than is used for internal connections in the cell (poly or diff.) to avoid possible design rule violations.

- extra symbols must be introduced to symbolize domain and connection areas (terminals) of the cells. Special terminal symbols may be introduced to implement concepts like "shared buses", see fig.3.2. A shared bus denotes a terminal area where overlap of compound instances is allowed.

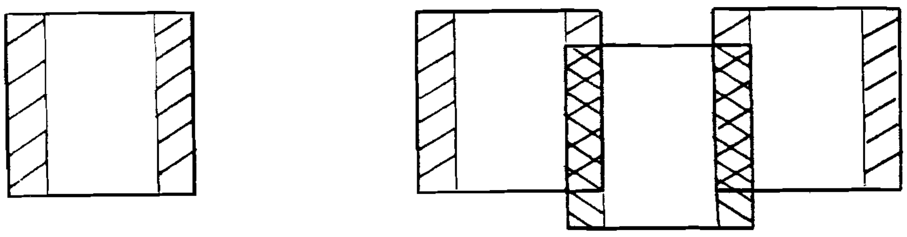


Fig. 3.2 A cell definition with a shared bus, and connection of compound instances via shared buses.

Since for VLSI design the advantages of a hierarchical approach are dominant, the IDS system is hierarchically oriented. In the present implementation a strong separation between hierarchical levels is used.

The consequences for the compactor are:

- Compaction can be performed at a single hierarchical level. During compaction of a particular compound the instances of subcompounds are not resolved, rather the subcompounds are treated as black boxes of which only the terminals are of interest.

- The number of elements in a hierarchical compound is rather small. This leads to a fast compaction and opens the way to iterative editing and compaction cycles.
- The graph representation of the layout can no longer be resolved with a single-pass algorithm, due to the introduction of double-sided constraints in the graph. As is shown in fig. 3.3 the shift of compound cell C2 with respect to the position of C1 is double-contrained. Compound C2 may shift 10 units to the left or 5 units to the right relative to the place of C1. The compactor uses a modified critical path algorithm to solve the graph with double-sided constraints.

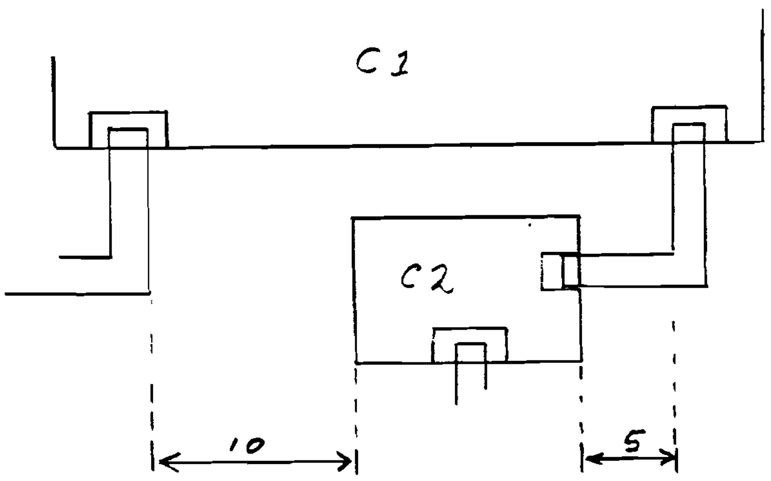


Fig. 3.3 The use of hierarchical cells introduces double-sided constraints.

3.2. Representation of layout elements

One of the major advantages of a symbolic layout method is the enormous reduction in design time (about a factor 10). The layout editor ISLE [3] uses a symbolic representation of layout elements. These symbols are called "sticks" . In fig.3.4 the symbols presently used in ISLE are given.

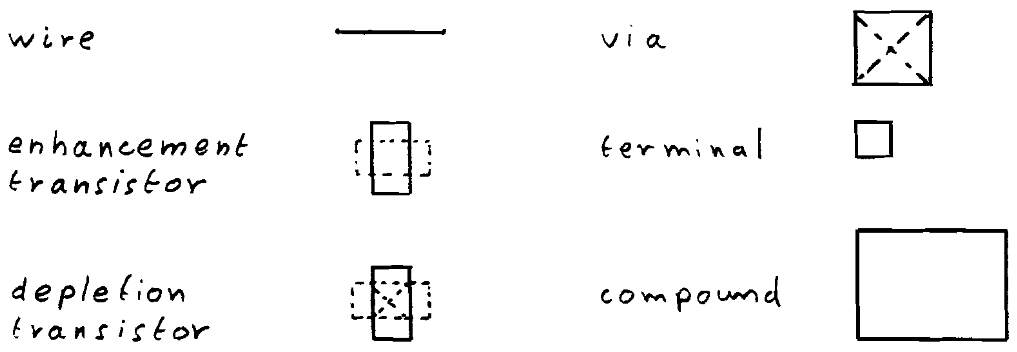


Fig.3.4 Basic symbols used in ISLE

The symbols are adapted to NMOS technology. The dimensions shown are in lambda units, and are default values. Lambda is the characteristic minimal distance, e.g. a half minimal line spacing. In a actual symbolic layout a great number of variations on these symbols can be found:

-- The user can specify greater dimensions in both horizontal and vertical direction of transistors and other sticks.
-- Different symbols are used according to the layers involved:
    - three different dashings for wires, symbolizing diffusion,polysilicon or metal
    - three types of vias: diffusion-metal contact, poly-metal contact, and burried contact
    - transistor: crossing of horizontal poly and vertical diffusion, or the other way round.

In the near future some new symbols may be introduced. For instance, a terminal symbol to symbolize shared buses. Further extensions are various symbols to instruct the compactor:

- jog symbol, to instruct the compactor that a particular wire may be bent,
- expansion symbol to instruct the compactor to add room for extra logic.

The layout designer may introduce self defined basic layout elements (basic cells) with the geometric layout editor Colormask [4]. These basic cells will be symbolized with a symbol similar to the compound-box symbol.

The wire symbol mentioned above deserves somewhat more attention. The simple form of the wire symbol leads to a clear symbolic drawing. However, the wire symbol is too simple since wires of various width are all symbolized with the same line. In the example of fig. 3.5.a two abutting vertical wires of width 2 and 4 lambda are drawn. After compaction the wires may be placed as in fig. 3.5.b.

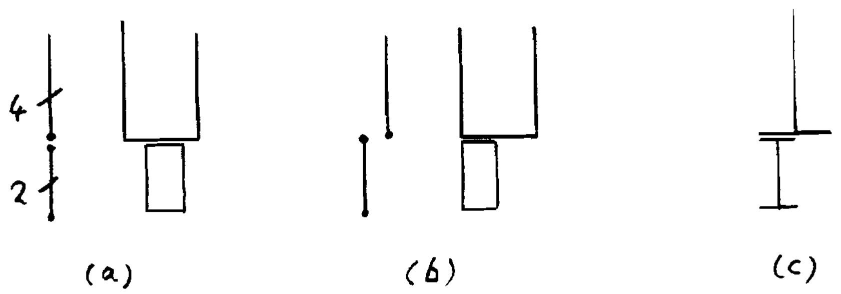


Fig.3.5 In the symbolic drawing geometrically connected wires sometimes seem to be split.

Geometrically the wires are still fully connected, but in the symbolic representation the wires seem to be not connected. Similar problems occur with wires abutting or overlapping other basic symbols.
A solution could be the introduction of another wire-symbol as in fig.3.5.c.

For the moment an extra symbolic design rule is used. The compactor prevents the symbolic splitting of wires by putting those wires into the same feature and thus shifting the wires over the same distance.

3.3. Interactivity

Interactivity is an important feature of the IDS-system. Tasks for designing a layout are distributed among man and machine in such way that each of them does the tasks that suit him best. As in other layout design systems, tasks such as drawing, compaction and design rule checking are performed by the system, and topology design is left to the designer.

With the IDS layout editors a designer can input the layout in a fast way. Changing the layout topology is facilitated by various commands such as shift, rotate, mirror and delete. So, since the IDS editors fully support topology changes, the changes in topology introduced by the compactor can be restricted to shifting.
The advantages of this approach are:
- the designer will not be confrontated with unappreciated changes in topology,
- the compaction process will be speeded up. So the designer can fully utilize the option of

re-editing a compacted compound.

3.4. Design rules

Constraints on the shift of elements are based on:
- geometric design rules
- symbolic design rules
- compaction rules

Geometric design rules follow from the technology used (NMOS). (See f.i. Mead and Conway). These rules give the minimum distances between and the minimum overlaps of wiring, transistors and vias. Overlap requirements are satisfied by the compactor by preserving the original overlap situation, or by placing restrictions on the relative shift of pairs of elements based on the geometrical dimensions of the elements.
At the moment the separation requirement for each pair of layout elements is set at 2 lambda. Compound instances are allowed to abut. A future improvement of the compactor is to introduce more variation in spacing requirements, and to make the compactor table-driven, so that changes in technology can easily be implemented in the program by reading in a new table of design rules.

Symbolic design rules and compaction rules are less universally as the geometric ones, and are therefore discussed below in more detail.

Symbolic design rules

Symbolic design rules follow from the hierarchical design of the layout editor and from the use of special symbols (e.g. transistor) [11], [3], [1].

The most important rules are:
a) Crossing of poly and diffusion only if there is a p-d via or a transistor at that place
b) Instances of compounds may not overlap, abutment is allowed.
c) Terminals are the only places where a compound is allowed to make contact with the "outside world". The connection can be a wire or a via. The (geometric) area of the wire or via may not exceed the terminal boundary.
d) The distance between transistors, vias, wiring and the surrounding compound box must be at least 1 lambda. Also the distance between a layout element and the box of an instantiated compound must

be greater than lambda. This is to ensure a separation of at least 2 lambda between a layout element and the elements of an instance of a compound.

e) Terminals must abut on the (inside of) the surrounding compound box.

f) Terminals may not be placed in the corner of a compound box, so may not abut on two sides of the box, in order to avoid conflict of rule d) and the minimum overlap requirements.

g) Geometric design rules may allow overlap of metal with f.i. diffusion lines. However, problems with the symbolic drawing occur if metal and diffusion totally overlap (fig. 3.6.a).

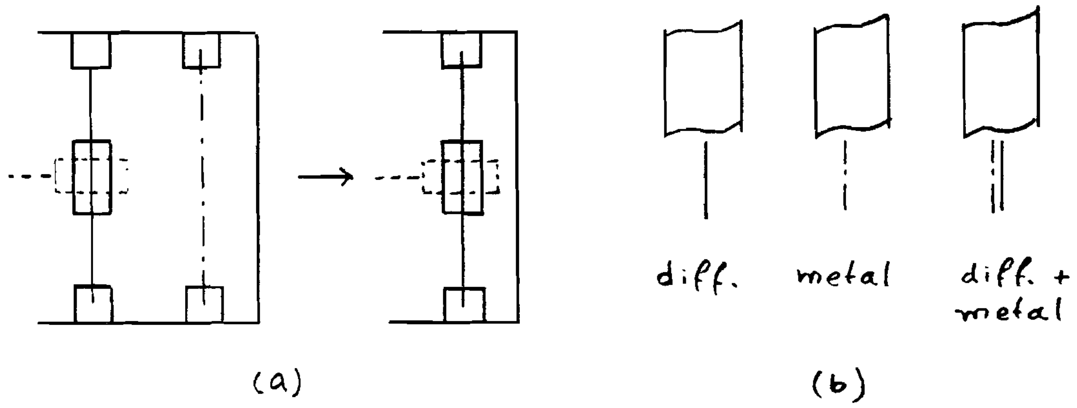


Fig.3.6 (a) Metal line apparently disappeared in the symbolic drawing after compaction. (b) Suggestion for an alternative wire symbol.

Therefore (at the moment) the compactor always generates constraints between elements, without looking to the layers the elements occupy. In fig.3.6.b a suggestion is presented to overcome disappearing of lines in the symbolic drawing: represent one line-type by a center-line, and the other two by a line which is placed a little left or right of the center-line.

Compaction rules

In this subsection the compaction rules are discussed. These rules give no restriction on the freedom of the layout designer, rather they show some internal processes of the compactor.
Compaction rules:

a) The position of subterminals relative to the corresponding compound box is fixed.

b) Terminals of the compound under compaction:
   - at the left side of the box won't move

- at the bottom and top side may shift left
- at the right side of the box may shift left if the following conditions are satisfied:
  - each of them shifts over the same distance
  - they stay the rightmost elements of the cell.

c) Two elements of width W1 and W2 (f.i. a via and a wire), which abut on each other have relative freedom of abs(W1-W2), see fig.3.7.a.

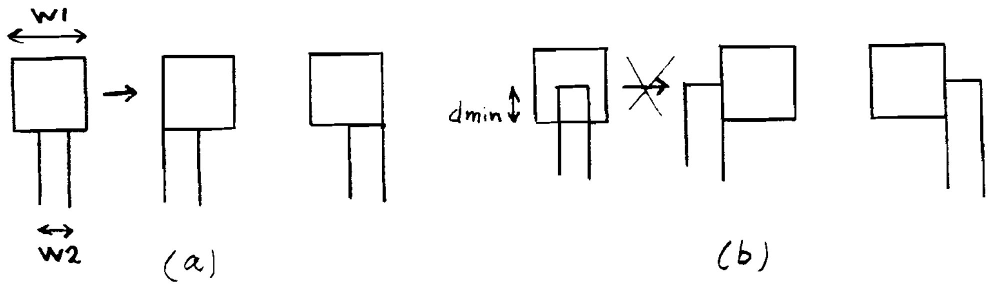


Fig.3.7 (a) The relative shift of abutting or overlapping elements is abs(W1-W2). (b) If the overlap is greater than dmin, the compactor does not allow a larger shift.

If there is overlap of more than dmin (=MIN(W1,W2)), the freedom is the same as above, altough the geometric design rules would tolerate a larger shift, see fig.3.7.b.

d) A line terminating on a perpendicular line of the same layer must (at the moment) cover it completely. This is necessary since telescopic elements won't generate constraints on shifts. This requirement is not fullfilled automatically by the layout editor ISLE. At the moment the user himself must take care of it. Further, the user must place a horizontal line-piece at the place where two vertical lines make contact.

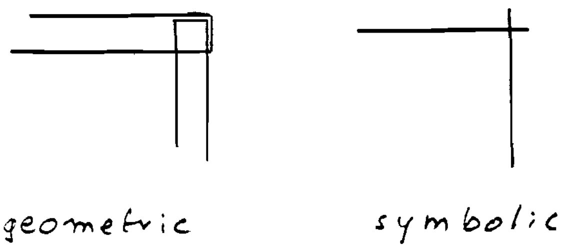


Fig.3.8 Complete covering of perpendicular lines

4. THE COMPACTION PROCESS

In this chapter the various actions will be discussed which are performed before and after execution of the compaction algorithm discussed in section 4.7. Section 4.1. gives an overview of the compaction program. Since the program treats compaction in horizontal or x-direction, and vertical or y-direction in the same way, only the case of horizontal compaction is discussed.

4.1. Overview

The compaction program will be initiated after an appropriate instruction to the interactive symbolic layout editor ISLE [3]. The program asks the user to enter the filename and the name of the compound to be compacted. By default, the names used in the previous editing or compaction cycle will be used. After names are entered or defaulted, the user may choose one of the following options: 1) horizontal compaction, 2) vertical compaction, 3) return to the menu of the editor.

The first action performed is input of the N layout elements of the compound. Of all elements the type of the element and geometric information (i.e. a surrounding rectangle) is needed from the database.

The second step is to sort the elements and search for overlap and abutment. For this search we use the algorithm for finding intersections of rectangles as described in [12], which is linear "in the average" with N.

In the third step the elements are partitioned into features. Overlapping elements which are not allowed to shift relatively to each other are put in the same feature. So elements belonging to the same feature will move as a single entity during compaction. Horizontal pieces of wiring (telescopic elements) are not put into features, because these wires can be stretched and shrunk according to the shift of elements connected to it.

In the fourth step the constraint graph is generated. The features correspond with the nodes, and the constraints correspond with the edges of the graph. The constraint between two features F1 and F2 results from the constraints between the elements of F1 and the elements of F2.

An ST-ordering of the nodes of the constraint graph is determined in the fifth step. In an ST-ordering the nodes are numbered so that each edge points from a lower to a higher number. The double-sided constraints, which can be considered as bidirectional edges, are not taken into account when the ST-ordering is determined. An ST-ordering can be determined linear in time, f.i. with a depth first search. The algorithm used in Compac is given in section 4.6.

In the sixth step the compaction algorithm is executed. It is an extension of the critical path algorithm. Backtracking is used to satisfy the max-constraints. The algorithm is presented at ISCAS-83 [13]. In section 4.7. the algorithm and a complete description are given.

In the last step of the compaction process the compacted compound is constructed. The elements of the features get their new place according to the distance the features have been shifted. After this is done the new length of the horizontal wires is calculated and the compacted compound is stored in the database.

4.2. Input of the elements of a compound cell

The compactor requires as input a compound cell which is free of design rule errors. Further requirements are that each line terminating at a perpendicular line must contact it by complete overlap (fig.3.8), and that a horizontal line is placed at the location where two vertical lines make contact.
If the input contains a design rule error, the error will be reflected into the compacted compound. If the other requirements have not been obeyed, then the compactor sometimes causes design rule errors, depending on the topology of the layout.

A compound cell of the hierarchical ISLE database consists of various types of elements:
- point-type sticks: transistors (horizontal or vertical, enhancement or depletion) and vias (p-m, d-m, or burried)
- line-type sticks: wiring in horizontal and vertical direction (poly, diffusion or metal). Each line-piece is a separate element.
- pins or terminals
- instances of sub-compound cells

Further, a domain is used to indicate the total chip area occupied by the elements of the compound. In the present version of ISLE the domain is a rectangle surrounding all elements. This rectangle is called the box of the compound. In future, a more complex domain will be used.

The place and the dimensions of each element, including compound instances, are given by a rectangle.
The type of an element (e.g. horizontal enhancement transistor) is denoted by a number. The type-number of a compound instance is used as a pointer to the compound definition. More information about the various types can be found in [3].

A similarity between point-type sticks and wiring is that for both the geometric representation consists of a rectangle. However, the compactor treats both types of elements in a different way: In the case of horizontal compaction wiring in horizontal direction will be treated as telescopic (or elastic) elements: the length of the horizontal connections is variable. On the other hand, the shape of the point-type sticks and the shape of the vertical connections will be preserved.
We don't use automatic jog insertion for two reasons:
- it takes more computer time, and
- the designer may not be pleased with 'unexpected' compaction results.
With ISLE, jogs can always be inserted manually.

The elements of a compound are stored in a linked list. Input of the elements can be described as follows.

```
DOMAIN(1...4):= domain of the compound
PT:=PT1            % pointer to the first element
N:=0
WHILE PT <> NILL
DO
  N:=N+1
  GET(PT,REC)      % REC contains element description
                   % REC(1...4) contains a rectangle
  PT:=REC(6)       % pointer to next element
  TYPE:=REC(5)     % type of element
  IF TYPE <> CITYPE  % compound instance type
  THEN BEGIN
           enlarge rectangle REC(1...4)
           ELEM(N,1...6) := REC(1...6)
       END
  ELSE BEGIN
       % store domain of compound without enlargement
       ELEM(N,1...6) := REC(1...6)
       FOR each terminal of this compound
```

```
        DO
          get terminal description
          enlarge rectangle
          N:=N+1
          store description in ELEM
        OD
       END
OD
```

In order to detect abutment of elements at the same time when overlap is detected, the rectangle of each element is enlarged by an amount MARGE in all directions (fig.4.1). MARGE will also be used for the separation requirements, it is currently set to 1 lambda. Of a compound instance, the domain and the terminals of the compound are retrieved from the database (no other elements). The rectangle corresponding with the domain of a compound instance is not enlarged.

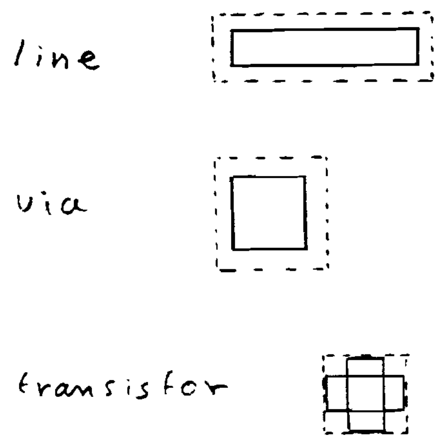


Fig.4.1 In the compactor, elements are represented by an enlarged rectangle.

4.3. Sorting the elements and detecting overlaps

In order to compose the features efficiently, the program first generates a list of overlaps for each rectangle. Finding all pairwise intersections among rectangles is called the 'rectangle intersection problem'. Checking each element against each other element would give an algorithm of order n**2. In the program COMPAC the solution for this problem proposed by Bentley, Haken and Hon [12] is used. (lineair expected time).

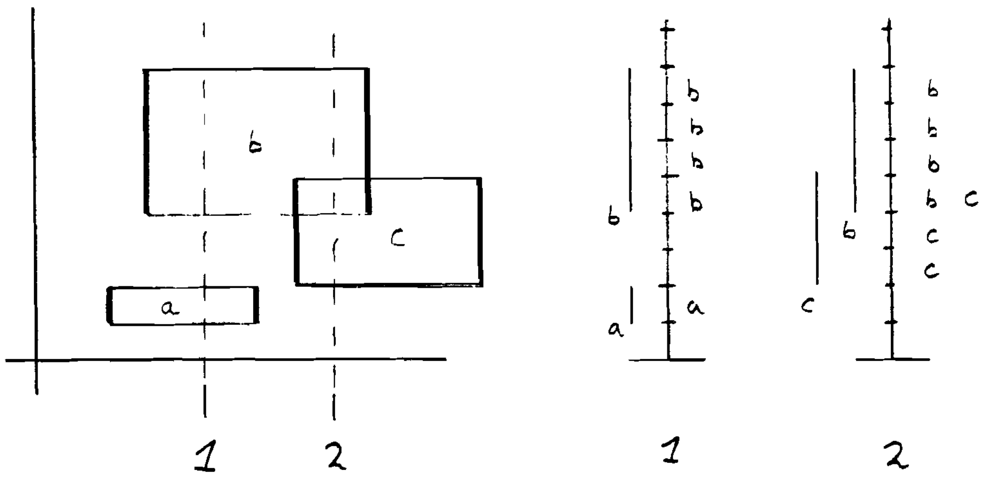


Fig.4.2 Illustration of the scan-line algorithm.

In this method a vertical scan line moves from left to right over the layout. At each point in the scan, all rectangles currently intersecting the scan line are represented as one-dimensional line segments. As shown in fig.4.2 these "active" line segments are stored in bins along the Y-axis. The Y-axis is divided in NBINY bins of width WY. In the paper of Bentley is argued that a good choice for WY is the average width of a rectangle, so that each segment is placed in two bins on the average. For each bin a chain (linked list) is used to store the line segments which totally or partly overlap this bin. All bin chains together give the total set of line segments intersecting the scan line. This set is called the segment set and is stored in array SS.
When the scan line encounters a left segment, first the bottom and top bin (IBYB and IBYT) corresponding with this segment are determined (projection onto the Y-axis). Second each line segment already present in bin IBYB through IBYT is reported, and at last a pointer to the rectangle corresponding to the left segment is stored in the chains of bin IBYB trough IBYT. Note that if two line segments are in the same bin, this does not necessarily mean that they actually overlap.

When a right segment is encountered the line segment is deleted from the corresponding bin chains.

In the algorithm below, the next three arrays are used:
EL: event list, contains a sorted list of left and

right segments. For each segment a record with three fields is used:
- ELPT: pointer to next segment,
- RL: indicating left or right segment,
- ILM: number of element the segment belongs to.

SS: segment set, contains a chain for each bin IBY. SS[IBY] points to the first link of the chain for IBY. Each link (record) has two fields:
- SSPT (linkpointer), and ILM.

CHECK: this array and the variable LAST are used to report overlapping segments only once, although they may be resident in more than one bin chain. The reported elements in CHECK are organized in a 'backward' chain: if segments 3 and 8 have been reported, then LAST=8, CHECK[8] contains 3, and CHECK[3] contains ZERO.

The scan-line algorithm:

```
FOR I:=1(1)N DO CHECK[I]:=NIL    % initialize
FOR IBY:=1(1)NBINY DO SS[IBY]:=NILL  % initialize bin chains
FOR J:=NBINY+1 (2) FLMAX DO SS[J]:=J+2 % init free chain
LAST:=ZERO
ELPT:=ELPT1       % pointer to first segment in EL
WHILE ELPT <> NILL
DO
  % take segment S from EL
  ILM:=EL[ELPT+2]; RL:=EL[ELPT+1]; ELPT:=EL[ELPT]
  IBYB:=(YB(ILM)-YBC)/WY % determine bottom and top bin of S
  IBYT:=(YT(ILM)-YBC)/WY % YBC = bottom of compound
  IF RL = left segment
  THEN
    BEGIN
      FOR IBY:=IBYB(1)IBYT
      DO
        SSPT:=SS[IBY]
        WHILE SSPT<>NILL    % report possible overlap by
        DO      % setting flags for resident segments
          JLM:=SS[SSPT+1]
          SSPT:=SS[SSPT]
          IF CHECK[JLM]=NIL
          THEN BEGIN CHECK[JLM]:=LAST; LAST:=JLM END
        OD
        % insert S in IBY, use link from free chain
        FLN:=SS[FL]
        SS[FL+1]:=ILM   % FL points to free link
        SS[FL]:=SS[IBY] % insert  in front of bin-chain
        SS[IBY]:=FL
        FL:=FLN          % update free link pointer
      OD
      % report the flagged segments in CHECK
      WHILE LAST<>ZERO
      DO
```

```
        CHREPORT(LAST,ILM)    % check if rectangles  overlap
        JLM:=LAST             % and report overlap in OLLIST
        LAST:=CHECK[JLM]
        CHECK[JLM]:=NIL       % reset flag
      OD
    END
  ELSE                        % right segment
    FOR IBY:=IBYB (1) IBYT
    DO
      SSPT:=SS[IBY]
      WHILE SS[SSPT+1]<>ILM  % search for segment S
      DO SSPTL:=SSPT; SSPT:=SS[SSPT] OD
      SS[SSPTL]:=SS[SSPT]  % remove link from bin-chain
      SS[SSPT]:=FL; FL:=SSPT   % add link to free chain
    OD
OD
```

The overlaps are stored in a overlap list (OLLIST). If two elements ILM and JLM overlap, then the element with the higher number (JLM) will be stored in the overlap list of the element with the lower number (ILM). An exception is made for telescopic elements: if one of the elements is a horizontal line, then overlap is reported in the list of the horizontal line. Overlap of two vertical lines will not be reported, in order to allow two vertical lines overlapping a horizontal line to shift apart (fig.4.3.a).

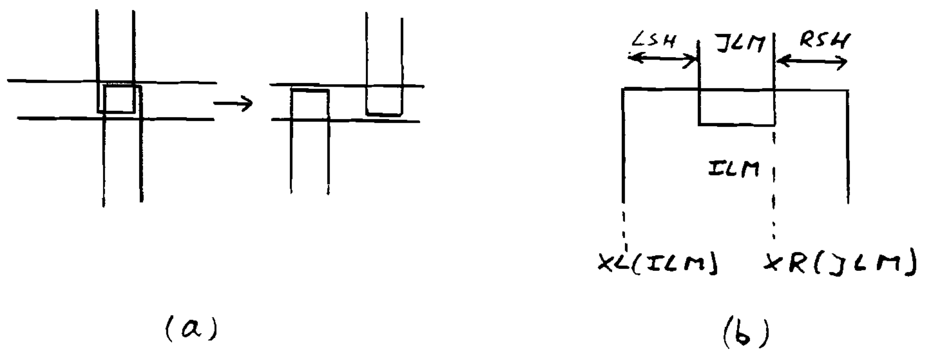


Fig.4.3 (a) No constraint is generated between two vertical lines overlapping the same horizontal line. (b) Allowed shift of overlapping elements.

Besides overlap, the maximum shift of JLM relative to the current position of ILM will be determined and stored in OLLIST. For example, the left shift and right shift of JLM in fig.4.3.b is calculated as follows:
LSH = XL(JLM) - XL(ILM)
RSH = XR(ILM) - XR(JLM)

So LSH and RSH are both positive in this case. In case of a fixed constraint, LSH and RSH are both zero. If there is no constraint, LSH and RSH will be assigned infinity.

A sorted "event list" has first to be generated before the algorithm above can be applied. In the event list (or segment list) all segments (left and right sides of the rectangles) are stored. The segments are sorted lexicographically (fig.4.4). The sortparameters are in lexicographical order:

- x-value of the segment
- type of segment (right before left segment)
- y-value of the bottom end of the segment

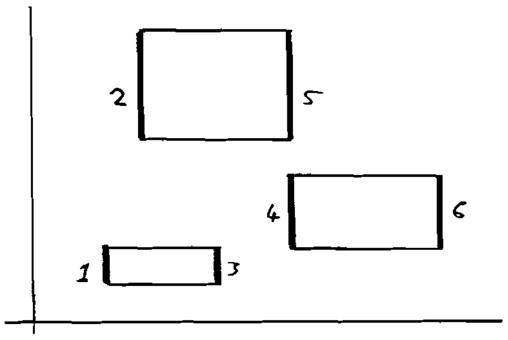


Fig.4.4 Lexicographical numbering of left and right segments of the rectangles.

For this sorting and storing of segments, the bin-method of Weide [15] is used. This method works as follows.

```
- divide the X-axis in NBINX bins of width WX
- initialize for each bin a chain in which the segments
    will be stored
- for each element do:
    - determine bin numbers IBX1 and IBX2 of both left
        and right segment (IBX=entier[X/WX])
    - insert the segments in the chain of bin IBX1 and
        IBX2 lexicographically
- couple all the bin chains in order to construct the
    event list.
```

At the end of the algorithm all line segments are partitioned into bins. Each bin contains a list of segments which have an x-value falling within the left and right boundary of the bin. The bin width WX is small, to reduce the lenght of the list.

For the following steps in the compaction process it is necessary that also the elements (rectangles) them-

selves are sorted. This sorting is very simple, since the sequence in wich we like the elements to be sorted is the sequence of the left segments in the event list. A possible way to gain access to the elements in this sequence is to use an extra array of pointers to the description of each element. However, in order to facilitate the (frequently) access to the element descriptions in further steps, we choose for actually re-organizing the list ELEM in which the elements have been stored.
This re-organizing is performed in the following way:

```
DO
     -find a cluster of two or more elements
          (a,b,c,...,k,l) which have to exchange
          places,
     - store the first element of this cluster tempo-
          rarily at place S (a->S),
     - copy the other elements of the cluster to their
          new places (b->a, c->b, ...,l->k),
     - copy the first element of the cluster to its new
          place (S->l)
UNTIL totally sorted.
```

Note that if there is only one element in a cluster, this element was already at the place according to the new sequence from the beginning.

The amount of work to be done is worst case as much as copying the entire array ELEM into a temporary array, and then copying back in the new sequence into ELEM. The advantage of this "cluster-method" is that no extra array is necessary.

The algorithm for the cluster method is as follows.

```
% NC[J] gives number of element that has to move to
% ELEM[J,1...6] (NC = new contents of place J)
CLPT:=0 %pointer in ELEM to first element of cluster
WHILE CLPT < N
DO
  CLPT:=CLPT+1
  IF NC[CLPT]=CLPT
  THEN element with number CLPT already at right place
  ELSE
    BEGIN
     STORE[1...6]:=ELEM[CLPT,1...6]
     FP:=CLPT               % free place pointer in ELEM
     FPN:=NC(FP)
     WHILE FPN <> CLPT
     DO
       ELEM[FP,1...6]:=ELEM[FPN,1...6]
       NC[FP]:=FP           % element now at right place
       FP:=FPN   % update free place pointer
```

```
            FPN:=NC[FPN]
          OD
          ELEM[FP,1...6]:=STORE[1...6]
        END
      OD
```

4.4. Partition the elements into features

Elements which overlap each other and which are not allowed to shift relative to each other (as reported in the previous step) are put in the same feature. Horizontal lines are not represented in the graph.

A simple extension to the present version of the compactor is that the user can specify a left bound and a right bound (vertical lines) between which compaction will take place. Default values for LBOUND and RBOUND are the left and right side of the compound box. The elements left of LBOUND or intersecting this line, will not move during compaction, and are therefore put in the same feature.

The partitioning can be performed linear in time with N, bye applying a breadth first search (BFS).

```
Algorithm:
- ILM=1
- while XL(ILM) <= LBOUND do
     - if element is a telescope then put in TLIST
     - else put element ILM into feature 1
     - ILM=ILM+1
- put all other elements into feature 1 which are not
     allowed to shift relative to elements already be-
     longing to feature 1 (use information from the
     overlap list).
- while ILM <= N do % N = total number of elements
     - if ILM not already belongs to a feature then
          - if element is telescopic then put in TLIST
            else
            - create new feature FTR
            - put ILM in FTR
            BFS(FTR) % see below
     - ILM = ILM + 1
where,
XL(ILM) is the left side of element ILM (The elements
     has been sorted lexicographically on x- and
     y-value of left-bottom point, cf. section 4.3.)
TLIST is the list of telescopic (horizontal) lines.
```

A consequence of this algorithm is that the features are ordered in the same way as the elements. After the execution of the algorithm above, the features on or right from the right bound are merged.

With bread first search all elements belonging to FTR are also put in FTR. This is done by searching the overlap list of each element ILM belonging to FTR. If an element JLM is found with a fixed constraint with ILM, then JLM is added to FTR.
Elements belonging to a feature are stored in the feature list FL. For each feature a separate chain is used with records <FLPT,ILM> (FLPT = linkpointer, ILM denotes the element). FL[FTR] points to the first link of the chain of feature FTR. FNUM[JLM] gives feature number FTR where JLM belongs to. (FNUM[JLM] is initially zero).

```
BFS(FTR)
FLPT:=FTR
WHILE FLPT <> NILL
DO
  FLPT:=FL[FLPT]
  ILM:=FL[FLPT+1]
  OLPT:=OLLIST[ILM]  % get pointer to first element in
                     % the overlap list of ILM
  WHILE OLPT <> NILL
  DO
    JLM:=OLLIST[OLPT+1] %get JLM overlapping with ILM
    OLPT:=OLLIST[OLPT]
    IF FIXED(ILM,JLM)
    THEN IF FNUM[JLM] = 0
         THEN ADD(FTR,JLM)  % add JLM to chain of FTR
         ELSE IF FNUM[JLM] = FTR
              THEN JLM has been added earlier
              ELSE MERGE(FTR,FNUM[JLM])
  OD
OD
```

In the overlap list of ILM only the overlapping elements JLM with a higher number have been stored. So, not all elements belonging to feature FTR1 can be found with a single call to BFS. This implies that in a latter phase, when feature FTR2 is created, FTR1 and FTR2 must be combined into one feature, if a fixed constraint is detected between elements of FTR1 and FTR2. This is done with the procedure MERGE. Since MERGE is not yet implemented, FTR1 and FTR2 stay apart, which leads to a fixed constraint in the graph.

4.5. Generate the constraint graph

In this step the separation requirements between features are determined. These are stored as constraints between the origins of the features. The origin of a feature is the left-bottom point of the surrounding rectangle. As a consequence, a constraint may get a negative value. Single constraints are stored in a min-constraint list (MINCLS), double-sided constraints are stored in a min-max-constraint list (MIMALS).

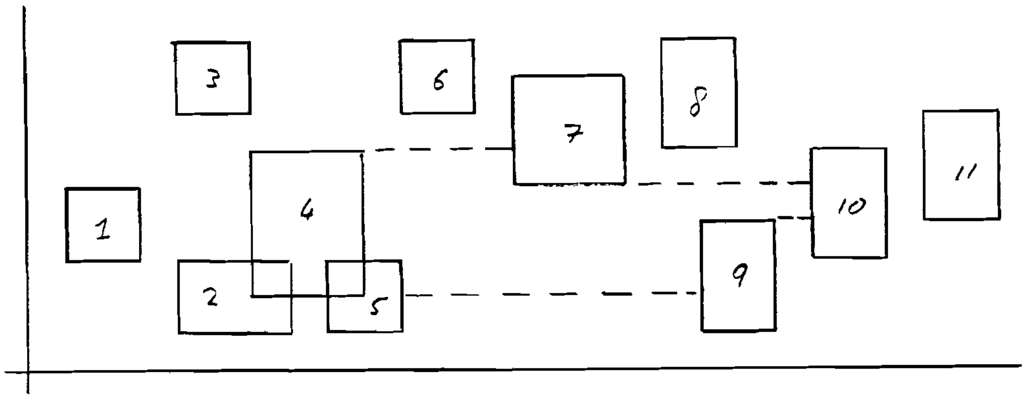


Fig.4.5 Constraints of feature 4. A window is used to reduce the number of comparisons. Each rectangle denotes the surrounding box of a feature.

Because windowing in combination with shadowing is used, the algorithm for determination of constraints is expected to be linear. The example of fig.4.5 illustrates how the constraints of feature 4 are determined. Min-constraints are generated from feature 4 to feature 7,9, and 10. Constraints to features 6,8, and 11 need not to be calculated, since these features fall outside the window set up for feature 4.
At the moment only one window is used for all the layers together. An improvement would be, the application of a separate window for each layer.
Checking feature 4 against 5 may, depending on the contents of the features, generate a double constraint or a min-constraint.

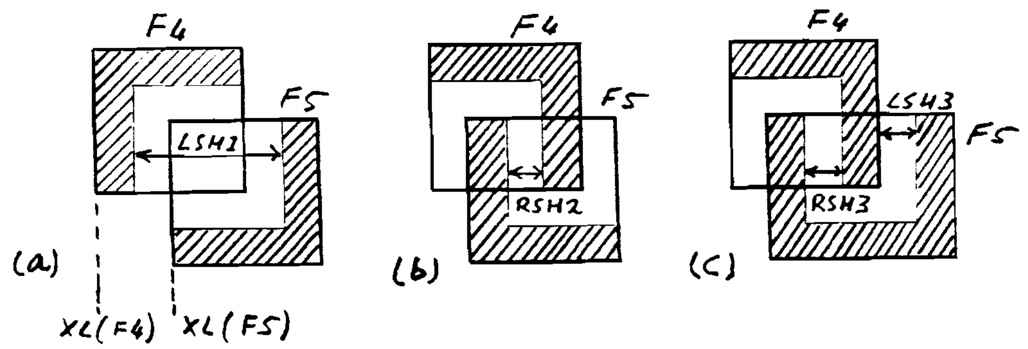


Fig.4.6 Three possible situations when feature boxes overlap.

As is shown in fig.4.6 overlapping features can generate a min-constraint from feature F4 to F5 (a), a min-constraint from F5 to F4 (b), or a min-max-constraint (c). The constraints for these three cases are calculated as follows:

a) LSH1 is the maximum left shift of feature F5 relative to the current position of feature F4 (LSH1>=0). The constraint between F4 and F5 is:
   MINCON(F4,F5) = XL(F5) - XL(F4) - LSH1

b) RSH2 is the maximum right shift of feature F5 relative to F4 (RSH2>=0).
   MINCON(F5,f4) = - (XL(F5) - XL(F4) +RSH2)

c) In case of a double constraint, the two constraints will temporarily be named D1 and D2, since it is not clear yet which one must be considered as a min-constraint and which one as a max-constraint.
   D1 = XL(F5) - XL(F4) - LSH3
   D2 = XL(F5) - XL(F4) + RSH3
   After generation of the ST numbering this double constraint is split into a min- and a max-constraint.
   If F4 becomes a lower ST-number than F5, then:
   MINCON(F4,F5) = D1; MAXCON(F4,F5) = D2.
   Otherwise, if F5 gets the lowest ST-number,then:
   MINCON(F5,F4) = - D2; MAXCON(F5,F4) = - D1.

The algorithm for determination of the constraints of feature 1 through NF is as follows.

```
FOR FTR:=1(1)NF       % NF = number of features
DO
  YWINB:=YB(FTR)        % bottom and top of feature box
  YWINT:=YT(FTR)        % give bottom and top of window
  FTRN:=FTR
  WHILE YWINT-YWINB > RWMIN .AND. FTRN < FTR
```

```
                          % RWMIN is min. width of rectangle
  DO                   % determine constraints of FTR
    FTRN:=FTRN+1
    IF YB(FTRN)>YWINT .OR. YT(FTRN)<YWINB
    THEN no interaction
    ELSE
    BEGIN      % determine constraints of FTR with FTRN
      CONSTR(FTR,FTRN,YWINB,YWINT,D1,D2,ICASE)
      CASE ICASE
      OF
        1: % no constraint
        2: ADMINC(FTR,FTRN,D1)          % (a)
        3: ADMINC(FTRN,FTR,-D2)         % (b)
        4: ADMINMAX(FTR,FTRN,D1,D2)     % (c)
      END
      % make window smaller if possible (shadowing)
      IF XL(FTRN)<XR(FTR) .OR. OVERLAP(FTRN)
      THEN do not adjust window
      ELSE BEGIN
             IF (YB(FTRN)<=YWINB) THEN YWINB:=YT(FTRN)
             IF (YT(FTRN)>=YWINT) THEN YWINT:=YB(FTRN)
           END
    END
  OD
OD
```

The features have been ordered lexicographically on x- and y-value of the left-bottom point of the feature box. Constraints of FTR and FTRN are only determined for FTRN>FTR.
The application of a window enables that a feature FTRN is skipped if it will not cause interaction. Unfortunally, it is necessary to test each feature FTRN if it falls outside or within the window.
The window is reduced by shadowing, if FTRN overlaps one of the window boundaries. All constraints have been determined when the window becomes smaller than the minimum width of a rectangle (RWMIN, i.e. 4 lambda). This shadowing can only be applied if the box of FTRN does not intersect the box of FTR. In addition, a feature FTRN can not reduce the window, if it overlaps with a feature with a higher number. This can be determined by the function OVERLAP(FTRN), which is not yet implemented.
A consequence is that the window stays longer 'open' than necessary. Shadowing can possibly be improved if constraints are determined in the sequence in which the elements have been ordered, rather than in the sequence of the features.

4.6. Determine an ST ordering of the nodes of the graph

The ST ordering of the nodes of the graph is based on the graph without the double-sided edges. The algorithm used for this numbering of the nodes is as follows:

```
- determine for each node N the number NP(N) of prede-
     cessors (incoming arrows)
- place all nodes with no predecessors in a list of
     candidates named CAND
- I:=0
- while CAND not empty do
     begin
     - take candidate C from CAND
     - I:=I+1
     - assign ST-number:  ST(I):=C
     - % remove node C from the graph:
          for each successor node S of C do
          begin NP(S) := NP(S) -1
               if NP(S) .EQ.  0 then add S to CAND
          end
     end
```

In general there will be more than one alternative for an ST numbering.
After each node has been assigned an ST-number by the algorithm above, the double-sided constraints are added to the graph. Each double-sided constraint is split in a min- and a max-constraint in such way that the min-constraint points from a lower to a higher ST-number.
User-defined constraints may also be added at this point, e.g. constraints for expansion, or fixed constraints between pairs of elements. These options are not yet implemented, but can easily be added to the porgram.

4.7. Compac

The critical path algorithm can be applied to resolve the graph if the graph has only min-constraints.

```
FOR FTR:=1(1)NF DO X[FTR]:=0;
FTR:=1
WHILE FTR <= NF
DO
  FOR each successor FTRS of FTR
  DO  % push successor to the right
      X[FTRS]:=MIN( X[FTRS], X[FTR]+MINCON(FTR,FTRS) )
  FTR:=FTR+1
OD
```

This algorithm is linear in the number of constraints. In the graph representation of the layout used by the compaction algorithm, the nodes are ST-numbered. A min-constraint MINCON(FTR1,FTR2) is directed from a node FTR1 with a lower ST-number to a node FTR2 with a higher number. A max-constraint MAXCON(FTR2,FTR1) is directed from a higher to a lower numbered node. The lower numbered node is called predecessor, the other successor. It will be clear that the min-max constraint graph contains cylcles of constraints. The cycles will never be overconstrained if the input for the compactor is a correct layout. An overconstrained cycle or a cyclic constraint may occur if constraints are added to the graph by the user. Therefore a test has been built in to detect such a cyclic constraint.

The compaction algorithm (see next page) can be looked upon as an extension of the critical path algorithm. Before min-constrained successors of a node FTRC are pushed to the right, first the max-constraints with its predecessors are satisfied. This is done by pulling the max-predecessors of FTRC to the right, if this is required by the max-constraint. If one of the max-predecessors has been actually moved to the right, backtracking is initiated. Until backtracking has been completed, the place of FTRC remains fixed, as is indicated with the variable XMAX[FTRC]. A node which has initiated a backtracking step, is not allowed to move further to the right. If it does has to move in order to satisfy the min-constraints, a cyclic constraint will be reported.
Further, each pulled predecessor FTRP, of which all constraints were satisfied (indicated by PLACED[FTRP]=TRUE), has to be reconsidered. This is indicated by resetting PLACED[FTRP]. At the time FTRP is reconsidered, only the nodes which are pushed or pulled further to the right by this node have to be reconsidered later. So, if backtracking occurs from node

10 to node 3, and node 3 has min-constraints to nodes 8, 10, 12, 17, only node 8 has to be considered for the second time. Other nodes between 3 and 10 are skipped by the algorithm, because the variable PLACED for these nodes remains set.

After execution of the compaction algorithm, X[FTR] contains the new place of each feature. The time complexity of the algorithm is exponential with the number of constraints. Each time a new node is placed, backtracking can occur. The subsequent replacement of predecessors may cause further backtracking cycles, initiated by those predecessors. Each cycle possibly taking as much work as the initial placement of that predecessor. However, since the graph is not fully connected, and if the number of max-constraints is low, the expected time of the algorithm is linear or slightly more than linear in the number of constraints.

Compaction algorithm

```
FOR FTR:=1(1)NF             % initialize each feature FTR
DO
  PLACED[FTR]:=FALSE        % true if all cons satisfied
  X[FTR]:=0                 % new place after compaction
  XMAX[FTR]:=infinity       % < infinity at backtracking
OD
PT=1                        % pointer in array ST
WHILE PT < NF
DO
  FTR:=ST[PT]      % ST[I] = feature with ST-number I
  IF PLACED[FTR]
  THEN PT:=PT+1
  ELSE
  BEGIN
    PTC:=PT; FTRC:=FTR  % FTRC = current feature
    IF XMAX[FTRC] = infinity
    THEN
      FOR each max predecessor FTRP of FTRC
      DO                    % check max constraints
        IF X[FTRC]-X[FTRP] > MAXCON(FTRC,FTRP)
        THEN                % pull max pred.
        BEGIN
          X[FTRP]:=X[FTRC] - MAXCON(FTRC,FTRP)
          PLACED[FTRP]:=FALSE  % cons of FTRP have to
                               % be re-checked
          PT:=MIN(PTC,PTP)     % PTP=ST-number of FTRP
        END
      OD
    ELSE IF X[FTRC]<=XMAX[FTRC]
         THEN max cons have been already satisfied
         ELSE exit and report cyclic constraint

    IF PT < PTC
    THEN XMAX[FTRC]:=X[FTRC]   % initiate backtracking
    ELSE
    BEGIN
      FOR            % each min successor FTRS of FTRC
      DO             % check min constraints
        IF X[FTRS] > X[FTRC] + MINCON(FTRC,FTRS)
        THEN BEGIN     % push min succ. of FTRC
                 X[FTRS]:=X[FTRC] + MINCON(FTRC,FTRS)
                 PLACED[FTRS]:=FALSE
             END
      OD
      PLACED[FTRC]:=TRUE; XMAX[FTRC]:= infinity
      PT:=PT+1
    END
  END
OD
```

4.8. Construction of the compacted compound cell

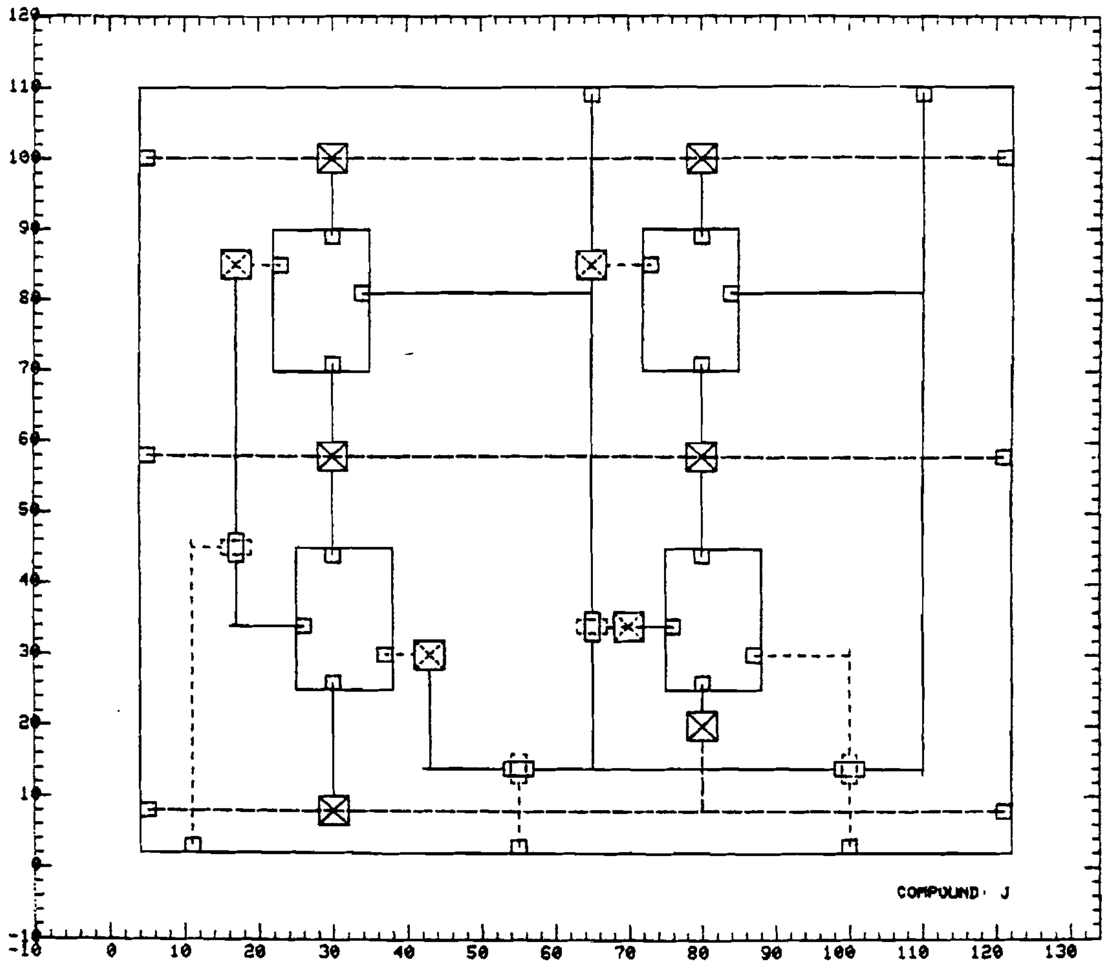
The new place of the origins of the features is stored in array X by the compaction algorithm. Since the original x-position XL(FTR) of the origin of each feature FTR is still known, the actual shift of the elements of each feature can easily be computed:
XSHIFT = X(FTR) - XL(FTR).
After each element is shifted to its new position, the length of the telescopic elements can be calculated. Information from the overlap list OLLIST is used to determine the left-most and the right-most element overlapping with the horizontal line.
If one of these elements is a vertical line, then the horizontal line is laid down so that full overlap occurs. Otherwise the original overlap situation with the leftmost or rightmost element is maintained. Horizontal lines which are not connected to something else are deleted.

4.9. Examples

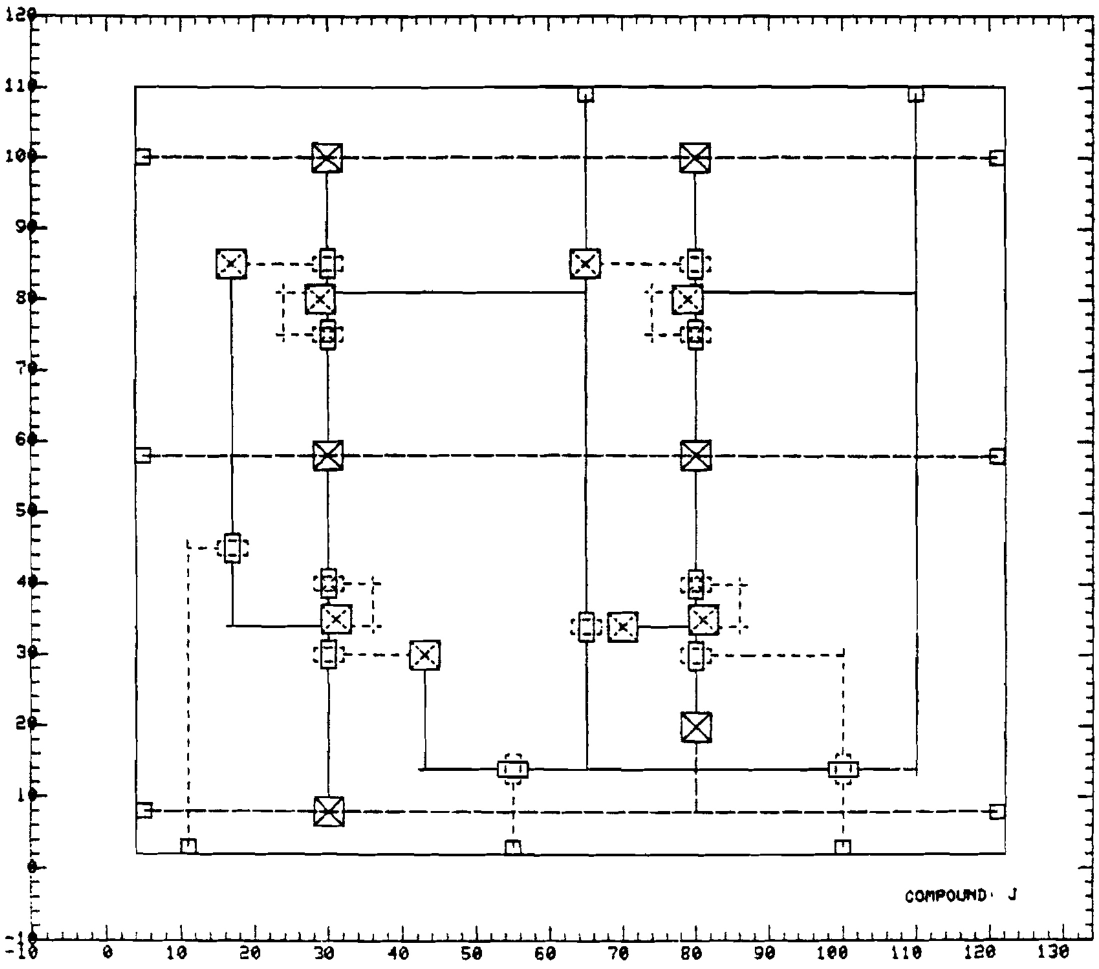
In this section some results of compaction are presented.
The first example is a JK-flip-flop which consists of four inverters, some pass-transistors and wiring. Double-sided constraints occur f.i. at the points where vias have been placed. The inverters are compounds of a lower hierarchical level (fig.4.7.a). Figure 4.7.b shows the flip-flop with the inverters unpacked. In fig.4.8.a the flip-flop is shown after a single x-compaction. A subsequent y-compaction gives the result of fig.4.8.b.

We will use this result to illustrate a drawback of the hierarchical symbolical approach.
The p-d via at x=44,y=26 in fig.4.8.b causes some waste of area because this layout element is not necessary. This becomes clear when we look at the expansion of the inverter in fig.4.7.b. Another thing which causes some waste of area is the symbolic design rule mentioned in section 3.4.g., which (for the time being) forbids overlap of f.i. metal and diffusion. This is illustrated with the horizontal diffusion line at y-value 11 and the horizontal metal line at y=7 in fig.4.8.b.

The second example (fig.4.9.a) shows that the compactor also can be used to compact a piece of interconnection. The result after a vertical and a subsequent horizontal compaction is shown in fig.4.10.b. The separation between the vertical lines at x=51,x=55 (at the place of the jog at y=72) is greater than necessary. This is because the compactor does not use electrical connectivity information at the moment.



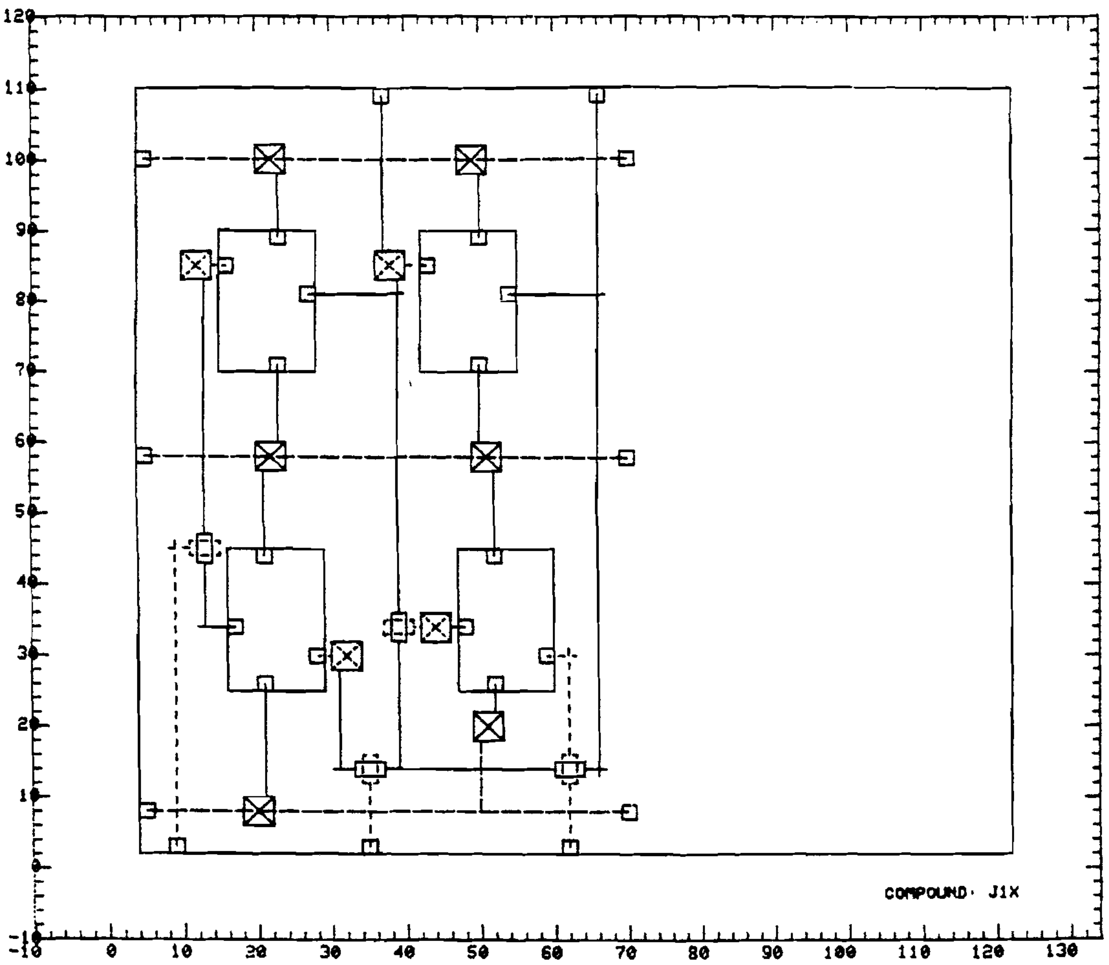
(a)



(b)

Fig. 4.7 JK flipflop

(a)



(b)

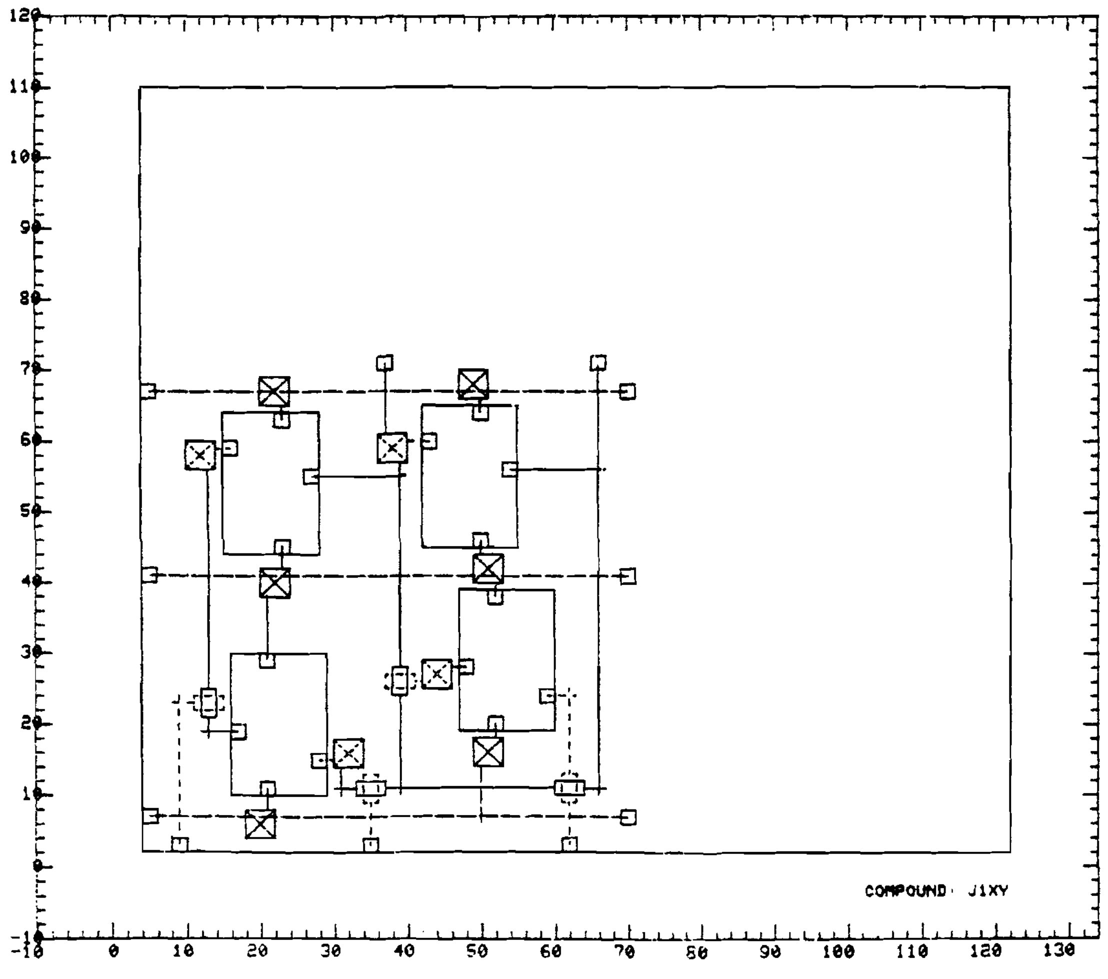
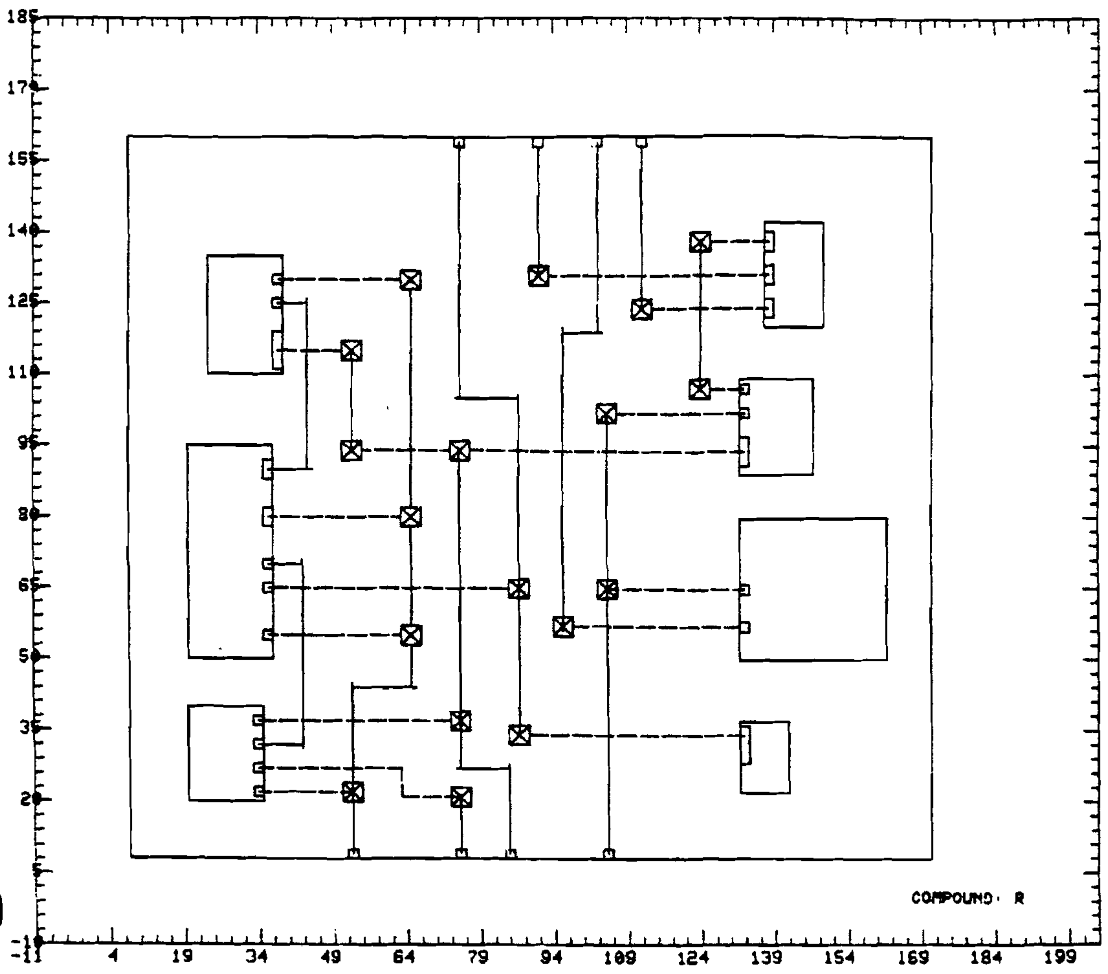


Fig. 4.8

(a)



(b)

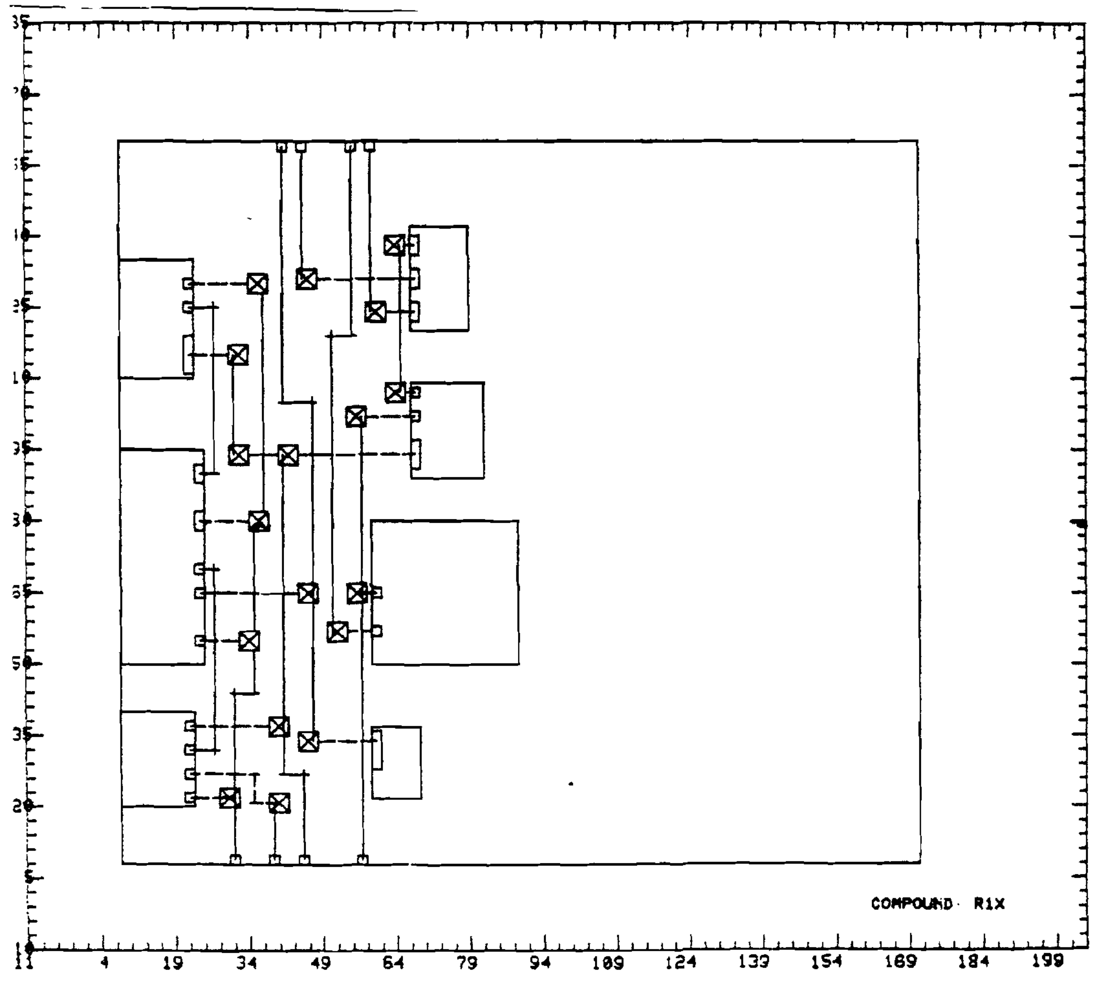
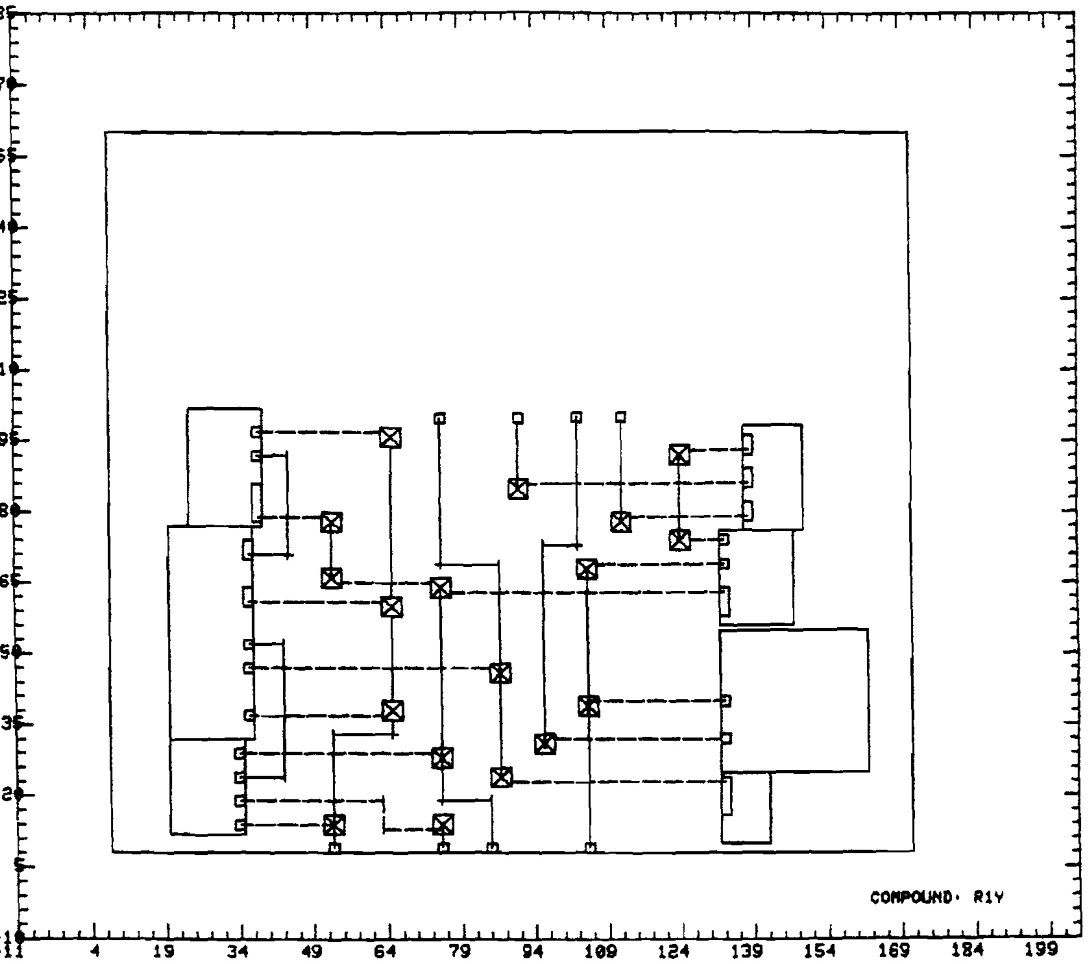
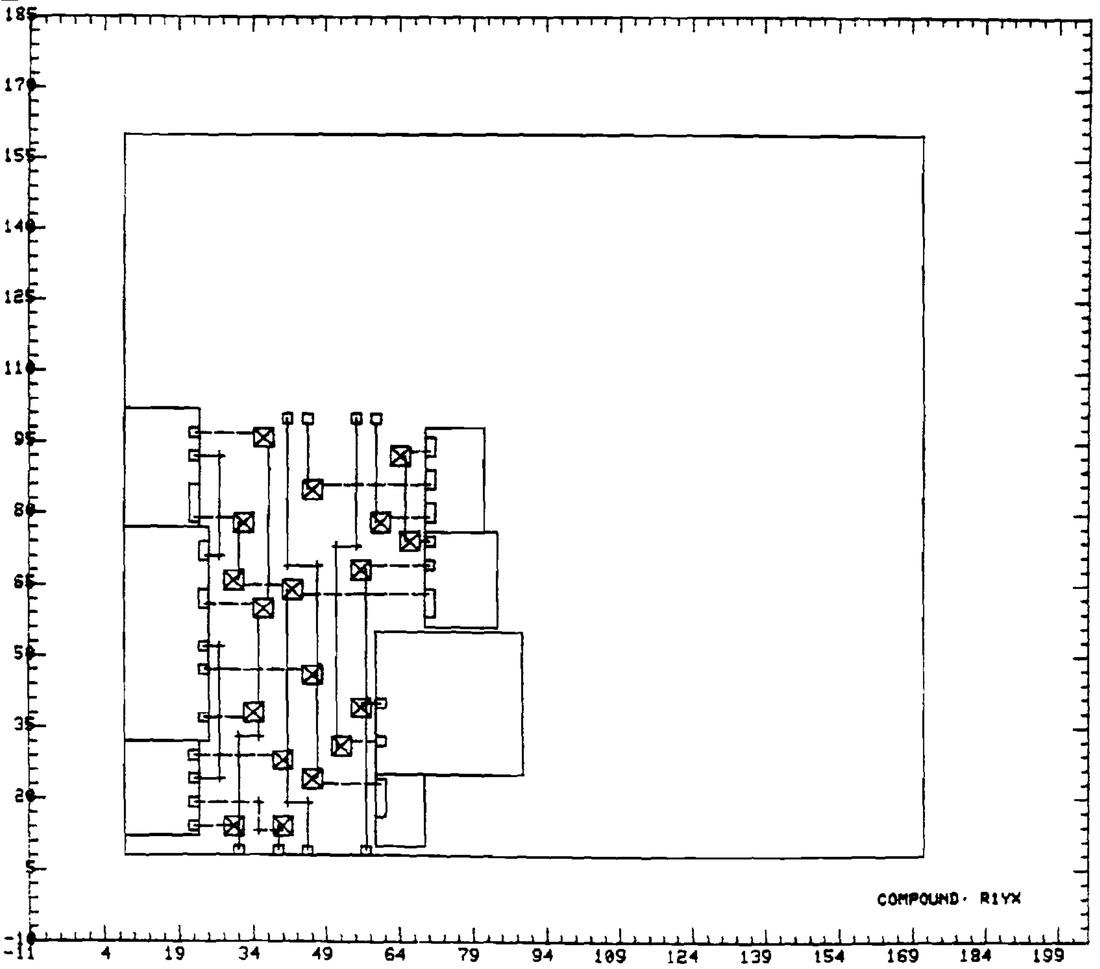


Fig. 4.9 Compaction of interconnection



(a)



(b)

Fig. 4.10

5. EXTENSIONS

The compaction program described in this report can be extended with several options.

Electrical connectivity information should be extracted from the layout in order to prevent that constraints are generated between vertical line-pieces of the same layer which initially do not overlap, but which are connected with each other via another path.
Some problems which will arise are:
- the generation of constraints will become super-linear, because shadowing will become complicated or impossible.
- terminals of compounds should provide net-information.
- diffussion lines must be split if a transistor is placed on the line.

A second option is that the compactor tells the user at which points useful jogs may be inserted. A useful jog point is always a point on a vertical line which is part of the critical path. In general there will be another critical path after the user has inserted one jog. So in most cases the user will compact again after each jog insertion.

A third possible extension is the use of 'affinity' in order to minimize the weighted length of wiring (a metal wire has a lower weight than poly and diffusion). Features which are on the critical path have a certain 'free float' and may shift a little to the left or to the right depending on the type and the number of wires connected at the left and the right side of the feature.

Besides jog insertion, other types of influence of the layout designer on the compaction process may be useful. For instance adding constraints between layout elements in order to save or generate room for extra logic (expansion). This can be done by adding extra edges in the graph. These user- defined constraints may lead to constraint cycles. The compaction algorithm can detect these cyclic constraints, and, at the moment, reports one of the nodes involved in the cycle. A possible extension is that the algorithm can 'recover' from a cyclic constraint evoked by the user, by not fully satisfying the user- defined constraint. This implies that the user- defined constraints should be assigned a lower priority than the constraints extracted from the layout.

6. CONCLUSIONS

The compaction program can compact hierarchical cells with double- sided constraints as is shown in the examples of section 4.9. The compounds of these examples consist of approximately 100 layout elements. The calculation of the compacted compound is performed in approximately one second. The compacted compound can be re-edited.

The compactor can be improved on the following points:
- improvement of shadowing and the application of a separate window for each layer.
- implementation of more design rules (the rules currently used are too conservative), and the application of a table of design rules, to make the program independent of changes in technology.
- elimination of the restriction on the input that each line terminating at a perpendicular line must contact by complete covering. This can be done by inserting for each horizontal line two extra nodes in the graph, representing left and right side of the line.
- elimination of the restrictions currently used to prevent the apparently disappearing of, and disconnection between symbolic layout elements, and elimination of the restriction that a horizontal line-piece must be present at the place where two vertical line-pieces abut.

References

[1] M.van der Woude, "IDS: an Interactive Design System for Integrated Circuits", THE Computing Centre Note 11, Eindhoven University of Technology, October 1982.

[2] W.M.G. van Bokhoven, "Piecewise-Linear Modelling and Analysis", Thesis Eindhoven University of Technology, 1981.

[3] C.A. Delhij, "ISLE, an Interactive Symbolic Layout Editor", Eindhoven University of Technology, 1982

[4] A. Borgt, "Een geometrische IC-Layout Editor in Fortran", Eindhoven University of Technology, 1982.

[5] R.C. Mosteller, private documentation.

[6] S.B. Akers, J.M. Geyer and D.L. Roberts, "IC Mask Layout with a Single Conductor Layer", Proceedings of the 7th. Design Automation Workshop, San Francisco, pp.7-11, 1970.

[7] A. Dunlop, "SLIM- The Translation of Symbolic Layout into Mask Data", Proceedings of the 17th. Design Automation Conference, pp.595-602, 1980.

[8] M.Y. Hsueh, "Symbolic Layout and Compaction of Integrated Circuits", Ph.D. Thesis University of California, Berkeley, UCB/ERL M79/80 Memo 1979.

[9] N. Weste, "Virtual Grid Symbolic Layout", Proceedings of the 18th. Design Automation Conference, Nashville, pp.225-233, 1981.

[10] C. Niessen, "The Role of CAD Tools in VLSI Design Methodology", ESSCIRC 1981 Digest of Technical Papers, Freiburg, 22-24 sept. 1981, pp.75-86.

[11] T.G.M. van Ooyen, "An Interactive Layout Editor/Compactor", Eindhoven University of Technology, 1982.

[12] J.L. Bentley, D. Haken and R.W. Hon, "Fast Geometric Algorithms for VLSI Tasks", VLSI New Architectural Horizons, Compcon Spring IEEE, 1980, pp.88-92.

[13] M. van der Woude and X. Timmermans, "Compaction of Hierarchical Cells with Minimum and Maximum Constraints", Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Newport Beach, 2-4 may, 1983.

[14] P. Dewilde, J.A.G. Jess, "NELSIS, a Co-operative System for Large Integrated Circuit Design".

[15] B.W. Weide, "Statistical Methods in Algorithm Design and Analysis", Ph.D. Thesis, Carnegie-Mellon University, August 1978.