EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Redesign of the GADL compiler

Boogers, W.A.P.M.P.

Award date:
1990

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section (ES)

Redesign of the GADL compiler

by W.A.P.M.P. Boogers

Master Thesis
Report on graduation work from April 1989 to May 1990
by order of professor dr. ing. J.A.G. Jess
and coached by ir. E.P. Huijbregts

The Eindhoven University of Technology is not responsible for the contents of training and thesis reports.

Abstract

The Gate Array placement and routing System (GAS) developed at the Eindhoven University of Technology is a CAD Tool for designing integrated circuits based on semi-custom technology. The system consists of software modules which perform separate steps in the design process. A chip design starts with a description of the gate array core and the macro library in a Gate Array Description Language (GADL), for which a new definition is proposed. GADL has facilities for both specifying the gate array image (the 'nude' chip area) and the appropriate macros in one grammar. Next, this description must be compiled into a database, which is done by the imagecompiler described in this report. The database is common to GAS , and has a human readable format. The size of the database is dependent on the size of the macro library, but it is largely independent of the image description.
The following steps in a design process are performed by the placer, global router, local router , and the mask-generator, respectively.

CONTENTS

1. Introduction 1
   1.1 Semi Custom design 1
   1.2 Automation of the gate array design process 2
   1.3 Gate Array Features 2

2. GAS overview 5

3. Language definition 6
   3.1 Basic notions 6
   3.2 Input file structure. 6
   3.3 Syntax of the MASTER section 7
   3.4 Syntax of the CORE_CELL section 14
   3.5 Syntax of the MACRO section 15

4. Implementation of the compiler. 18
   4.1 The compilation process. 18
   4.2 Compilation of Core Cells. 18
   4.3 Macro compilation. 21
   4.4 Technology compilation. 31

5. Conclusions 34
   Literature 35
   Appendix A : Syntax diagram of GADL 36
   Appendix B: algorithm description meta language. 42
   Appendix C : Description of SBNF 44

LIST OF FIGURES

Figure 1.1. Examples of gate array layouts 2

Figure 1.2. Core cells 3

Figure 2.1. GAS framework 5

Figure 3.1. Shadowing illustrated with stacked via example 10

Figure 4.1. Orthogonal orientations of a box in 2-space. 19

LIST OF TABLES

TABLE 4.1. The transforms reduction table. 20

1. Introduction

In the most recent decades of the electronics industry, a number of ways to implement designs are deployed. Basically we distinguish three methods :

- Discrete design
- Semi Custom design
- Full Custom design

The order shown indicates increasing cost, turn around time and density.

Discrete design uses standard off-the-shelf components and these are interconnected externally, for example on a printed circuit board. This technique has advantages for rapid prototyping, but designs quickly tend to become bulky and there is little protection against unauthorized copying. Also, reliability of the realized design is a matter of concern.

At the other end of the scale there is Full Custom design. Its main advantages are a high density, a high reliability, good protection against piracy. However, these virtues are compromized on by some other facts: Full Custom is not profitable if not used in high volume applications, and the turn around time is quite long (months). Furthermore, flaws in the design will carry a huge price tag.

Semi Custom design fills the gap between Discrete and Full Custom Design. The method is based on standard parts which can be customized within reasonable time (weeks) at a low cost per chip. This makes the integration of a circuit profitable at much lower volumes than Full Custom design.

1.1 Semi Custom design

There are a number of forms known; the basic three are:

- Field Programmable ICs
- Gate Arrays
- Poly Cell design

The order shown is increasing cost, turn around time and density.

Field programmable ICs are completely finished (from the manufacturer's point of view) and usually put into packages. Customization is done by the customer himself ("in the field") and has to be done very carefully. In general, no repair is possible in case of mistakes. An example of this kind of semi custom design is PAL (Programmable Array Logic), which can be customized by blowing fusable links.

The second kind of design is quit different : the chips initially are finished up to the metalization phase. To complete the metalization, the customer has to design the needed masks, which will be applied by the chip manufacturer. The metalization is considered here as the customization, and this completes the chip. A large library of often needed cells, called a macro library, is used to speed up the design. These macros can be seen as metal wire patterns, which, when mapped onto the gate array, perform a specified function. The performance of these macros is guaranteed by the manufacturer.

The Poly Cell approach needs much more layout design. A large library of standard cells is used to speed

up the design. The cells stored in the library are essentially the same as those for the gate array method; however, the complete layout is stored instead of just metalization patterns. This enables a higher density to be achieved, but this method is more time consuming.

1.2 Automation of the gate array design process

Usually, the chip manufacturer provides the software needed to aid the design process. The software is dedicated to a particular type of gate array, and even the hardware to run the programs might be prescribed. This creates a monopoly situation for the manufacturer.

To facilitate the development of circuits based on gate arrays in a technology and manufacturer independent way, a Gate Array placement and routing System (GAS) is developed at the Eindhoven University of Technology [Jess86]. The system consists of software-modules, each performing a specific task in the design process, and the modular approach provides for flexibility in two respects: first, when better algorithms become available a module can be rewritten and the old one exchanged, and secondly the design system is not limited to a specific technology. Furthermore, the design system has an open architecture and runs on a variety of hardware platforms.

The entry to the GAS is a chip description written in GADL, the Gate Arry Description Language. GADL facilitates an easy and hierarchical way of describing gate array features, as well as a concise description of the set of predefined function blocks called the macros. The description in GADL will be compiled into a database common to all the GAS modules, and this compilation is done by the GADL compiler. The main focus of this report is at the definition of GADL and at the development of the GADL compiler.

1.3 Gate Array Features

In order to appreciate the way gate arrays are described in GADL, we first need to look at some basic features of gate arrays.

Gate arrays can be seen from a very detailed point of view : one can study features of individual components such as transistors and capacitors. A more hierarchical approach groups together those components into basic cells which are the basis for the description in GADL. Common to most gate arrays is a rectangular layout of basic cells, the cells repeated in rows and columns on the chip area. Often, but not always (e.g. the Sea Of Gates Concept) the basic cells are separated by routing channels. The area thus formed is often called the gate array image (or, for short, the image), to which the customization is to be applied. Surrounding the image are the Input/Output buffers and bonding pads, which usually don't offer much space for routing. Figure 1.1 shows some typical examples of gate array layouts.

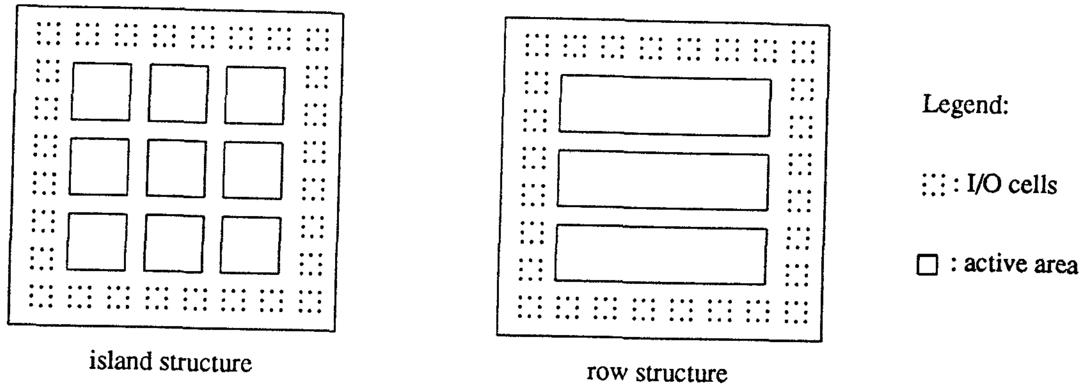


Figure 1.1. Examples of gate array layouts

The above mentioned routing channels may consist of various components: they range from a few orthogonal tracks to constructions with prefabricated underpasses with fixed or programmable contacts to the metal layers above. The gate array cells are more complex, since they usually contain a number of active components which enables one to build a set of functions in a particular technology. The basic cells are referred to in GADL as core cells. Figure 1.2 shows some types of cells. A combination of core cells and a suitable wiring pattern enables the designer (or the chip manufacturer) to build logic functions such as flipflops, nands, and the like. These wiring patterns are referred to in GADL as stamps.

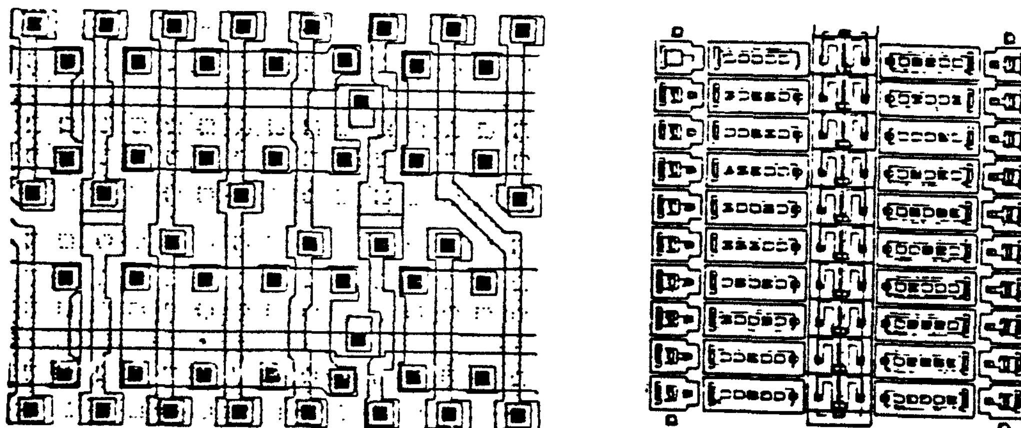


Figure 1.2. Core cells

Nearly all gate array designs limit the wiring to orthogonal directions, on predefined tracks. Non orthogonal wires are usually provided by the gate array foundry.

The above mentioned properties lead to the use of an orthogonal three dimensional grid for the modeling and description of core cells and routing features in GADL.

A description of the core cells and the macro library is insufficient for GAS to complete a design: apart from all the design info some extra information for the routing phase is needed. To guide the router, costs and designrules can be described in GADL. Cost statements are used to force certain routing decisions e.g. make a routing path attractive by specifying lower costs for particular edges. Designrules stem from physical limitations (technology dependent), and therefore may not be violated. Finally, GADL provides

statements in which distances between gridlines and widths of the various components (wires, vias and holes) can be expressed.

2. GAS overview

The layout of the Gate Array design System is given in figure 2.1. At the top level, there are two descriptions needed to inititate the realization of a design. First, a description concerning the gate array image and the macro library has to be supplied. This description is written in GADL, and it describes those parts of the chip substrate that are of importance to the placing and routing phases, and it also describes the collection of functions available for customizing the image.
Secondly, a description reflecting the design is to be supplied, which can be generated by hand or by a schematics entry tool, consisting of a net-list, a module-list, and some other information (see also [Huij90] and [Slen90]).

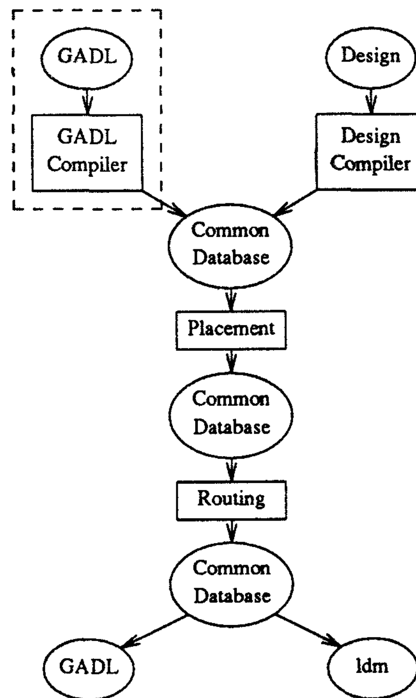


Figure 2.1. GAS framework

The GADL compiler takes the gate array description and compiles it into a library of macros and a core database, dubbed "Common Database" in figure 2.1. Other entries in this database are made by the design compiler, which translates the design description into the database format.
The next step in the design process is placement of function blocks. The placement procedure consists of two phases : global placement and detailed placement. First, the global placement subdivides the image into cells and attempts to assign a cell-location to each macro, whereas the detailed placer subsequently tries to find a legal placement and an appropriate stamp for each macro. Currently there are placers based on three different principles: simulated annealing ([Jon84] and [Otten84]), eigenvalue decomposition ([Frankle86]), and random placement.
Following the placer comes the routing phase, which is also split in two pieces: a global- and a detailed routing program. The global router determines the interconnect between the cells mentioned above, while the detailed router determines the layout of the interconnect inside cells. The principles underlying the routing phase are detailed in [Lee61],[Nuij85] and [Slen85].
To obtain masks for the concluding customization process, a mask generator (the arrow between "Common Database" and "ldm" in figure 2.1) takes the information generated by the router and generates the needed factory-ready information.
The items covered in this report are within the dashed box in figure 2.1.

3. Language definition

Input to the GADL compiler is a description of the image, together with a macro library, technology information, designrule and cost functions.
As an introduction to the GADL compiler, we will first introduce some basic notions and design considerations. Next, we will look at the language itself and the facilities it provides for supplying all the information needed.

3.1 Basic notions

The description of the gate array chip in GADL is hierarchical, and this is based on looking at the chip from two points of view.
In one view, this is the chip as it is supplied by the foundry, having no customization applied. At the other side, we look at the applicable customization patterns. Descriptions written in GADL however must be created bearing in mind that GADL only describes the features of importance to the placement and routing phases.

In GADL, the description of the "bare" chip is contained in two blocks, a block called MASTER and a block containing all the CORE_CELLs. The latter block details the basic patterns which are repeated in rows and columns over the chip area. The description consists of layer properties, wiring patterns, designrules and cost functions. This block is the lowest level in the hierarchy. The MASTER block deals with features which apply to the entire chip and therefore cannot (conveniently) be described in the CORE_CELLs. (Wires "between" core cells are examples of such a feature, and the placement of the core cells themselves is another example). Added to the MASTER level are statements to describe technology dependent characteristics, such as distances between gridlines and widths of wires and vias.

The customization patterns are specified in the third block in GADL, the block which contains the MACROs.
This part has descriptions of the macro library applicable to this particular image. A macro is considered here as a *functional item* , e.g. a *flipflop* or a counter. Depending on the exact position on the chip and the underlying CORE_CELL(s), such a functional item has a number of physical implementations. To describe these, each MACRO has one or more STAMPs, each stamp being an implementation and having its own allowed positions on the chip.
Stamps differ, generally speaking, in geometric shape, area and performance. To facilitate a hierarchical design, a stamp can be constructed with some basic statements (wiring, etc.) and calls to other stamps. This latter feature will cause the called stamps to be "incorporated" in the stamp under construction. Note that this process is not performed by the compiler. From a physical point of view it should be clear that recursion is not allowed. In appendix A a syntax diagram of GADL is presented.

3.2 Input file structure.

The GADL input file is logically divided into three parts :

i. First, we have a section describing features of the entire gate array image, this section is called MASTER.

ii. Following is the CORE_CELL section, which has the description of all the basic cells the image consists of. Certain defaults can be set for each core cell.

iii. The last section is the MACRO section, which describes the macro library applicable to this particular image.

The three parts mentioned above may be preceeded by an optional section containing DEFINEs, in which integer constants can be given convenient names.
GADL is easily described in SBNF, which will be used here to explicate the various parts of the language. Appendix C contains a syntax description of SBNF. A complete and valid input description written in GADL has the following syntax:

<sentence> ::= [<defines>] <master> <core_cells> <macros>.

Nested C or PASCAL style comments may be used inside a sentence.

Syntax of the optional DEFINE part :

<defines> ::= { " DEFINE " { <define_name> <integer> }+ }+.

semantics:
DEFINE allows integer values to be associated with identifiers, as to clarify the input description.

examples:

```
DEFINE   vdd 5
         max 100
DEFINE   drl_offset -2
         num_y 1
```

3.3 Syntax of the MASTER *section*

The image is described in the master statement,
syntax:

<master> ::= " MASTER " <master_name> "(" <align> <layers>
[<dimension>] <calls> [<nets>] [<npwirings>]
[<drl>] [<cost>] <distances> <widths> ")" .

<align> ::= " ALIGN " <neg_number> <neg_number>.

<layers> ::= { "LAYER" { <layer_name> ("FIX(ED)?"|
"PROG(RAMMABLE)?") ("FIX(ED)?" |
"PROG(RAMMABLE)?") }+ }+.

<dimension> ::= " DIM(ENSION)? " <number> <number>.

<number> ::= <*define_name*>|<integer>.

Note : w1(w2)? means : w1 is shorthand for w1w2, specifying just w1 is sufficient.

semantics:
An ALIGN statement specifies the alignment of the image with respect to the physical chip boundaries. The given integers will be used in the mask generation process, but are of no relevance to the compilation. The names of the layers and their default properties are given in a LAYER statement. Layers are taken in top down order, the first parameter behind the layer_name indicates the routing freedom for wires, the second indicates the routing freedom for vias. Next, in a DIMENSION statement the size in the x and y direction might be given. If ommitted, the size of the master will be computed from the sizes of the called CORE_CELLs.

example:

```
MASTER   MyDesign   (

         ALIGN      min_x    min_y

         LAYER      metal3   PROG     PROGRAMMABLE
                    metal2   PROG     FIX
         LAYER      metal1   PROG     FIXED
         LAYER      poly     FIXED    FIXED

         DIM        x_size   y_size   /* not mandatory here */
         .
         .
                    )
```

syntax:

```
<calls>          ::=  { " CALL " { <core_cell_name> [<trans_list>]
                      <position_list> <copy_list> }+ }+

<trans_list>     ::=  { " MX " | " MY " | " R1 " | " R90 " | " R2 " | " R180 " | " R3 " | " R270 " }+

<position_list>  ::=  { "(" <number> <number> ")" }+

<copy_list>      ::=  ["CX" "(" <delta> <times> ")"] ["CY" "(" <delta> <times> ")"]

<delta>          ::=  <number>
<times>          ::=  <number>
```

semantics:
The CALL statements actually define the floorplan of the image, by specifying the coordinates at which CORE_CELLs (possibly with some geometric transformation) are placed by the manufacturer of the gate array. For the representation in the database, the optional transformations in <trans_list> are applied to the called CORE_CELL in the order given. R1 means rotate 90 degrees counterclockwise (can also be specified as R90), R2 rotates 180 degrees (R180) and R3 rotates 270 degrees (R270). MX mirrors the core cell in the x-axis, and MY mirrors in the y-axis. The <delta> in a copy statement specifies the x or y increment for the copy operation, and <times> is the number of repetitions. When both CX and CY are specified, a "grid" will be created over the ranges in x and y direction, all crosspoints of the "grid" being addressed.

example:

```
CALL MyCore MX R1 (0 0) (0 10) CX (5 2) CY (5 1)
```

will cause the list of core cells to be searched for MyCore, the transformations will be applied to create a new instance, and this new instance will be placed at positions (5j 5k), $j \in [0..2]$, $k \in [0..3]$.

syntax:

```
<nets>          ::=  { " NET " <net_name> ( " POS " <layer_name>
                     <position> | <net_wirings> ) }+.

<net_wirings>   ::=  {(( " WIRE " | " EQ " ) { <layer_name> <position>
                     <position_list> <copy_list> }+
                     | " VIA "  { <layer_name> <position_list>
                     <copy_list> }+ )}+.
```

semantics:
NET statements describe (inside the MASTER section) nets which are global to the entire image, and can be used for example for power and clock lines. The names given to these nets are also global. Nets having a global name and defined inside a MACRO will be associated with (and are assumed to be connected to) the corresponding global net. WIREs and EQs always span two or more positions, while wire segments between adjacent coordinates in the position list must be orthogonal.

syntax:

```
<npwirings>   ::=  { (( " PWIRE " | " NWIRE " ) { <layer_name>
                   <position> <position_list> <copy_list> }+
                   | ( " PVIA " | " NVIA " ) { <layer_name>
                   <position_list> <copy_list> }+ ) }+.
```

semantics:
NVIA and NWIRE statements define via and wire blockades. They can be used for layers which are set to 'programmable' but do have certain fixed regions. At the given positions, the router can not create a via or a wire. The reverse holds for the PVIA and PWIRE statements: these allow for a programmable region in an otherwise fixed layer.

examples:

```
NET     vdd   POS metal3 (1 1)
              WIRE metal2 (1 1) (10 1) (10 10) CY (10 3)
              VIA  metal3 (2 2)
              EQ   metal3 (1 1) (2 2) CX (2 2) CY (2 num_y)

NWIRE   metal3 (5 1) (5 2)
PVIA    metal2 (3 1)
.
.
```

syntax:

```
<drl>          ::=  { "DRL" {( " HOR(IZONTAL)? " | " VER(TICAL)? " | " VIA " )
                    <layer_name> <position_list>
                    <copy_list> <shadow_list> }+ }+.

<shadow_list>  ::=  "SHADOW"
                    "(" { ( "HOR(IZONTAL)?" | "VER(TICAL)?" | "VIA" )
                    "(" <layer_name> <offset_x> <offset_y> ")" }+
                    ")" .

<offset_x>     ::=  <neg_number>
<offset_y>     ::=  <neg_number>
```

semantics:
Designrules are divided into two types : numerical and structural designrules. Structural rules tell something about preferred directions, not recommendable configurations, etc. They can be expressed in the COST function, and will not be discussed here. In general, numeric designrules take the form of a set of permissible geometries available to the designer to make devices and interconnections within the limitations of the process and without violating the physical constraints required for proper operation. Gate array designrules, in contrast, are quite different from the general designrules because only the interconnection processing steps have to be carried out. Since the devices are prediffused, the only concern for gate array designrules is for the minimum separations between interconnect elements (wires and vias). Two basic designrules are implicit in our way of modeling the image:

— All interconnections must be made on gridlines;

— All vias must be placed on crossing points of gridlines.

Also, since everything is expressed in terms of grid steps, we don't need to concern about the actual size of objects. This enables the use of the "shadowing" concept for the modeling of the designrules: each edge in the grid shadows other edges in its surrounding area. Figure 3.1 explicates this notion.

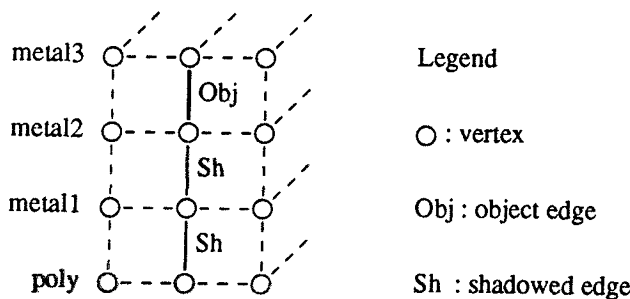


Figure 3.1. Shadowing illustrated with stacked via example

In practice this means that if the router wants to extend a path along an edge s, its shadowed edges must not be occupied. The current implementation in the compiler allows for some nuance in this statement: if an edge is shadowing multiple edges in one SHADOW clause, the router will be blocked if and only if all edges in the clause are occupied. On the other hand, when there's more than one designrule for an edge s, the router will be blocked as soon as one of the rules will get violated. Of course the reverse holds : when the router wants to occupy one of the edges in a SHADOW clause, not all the others can be occupied.

These complementary rules will be generated automatically. The following example will clarify matters (see also figure 3.1).

example:

```
DRL   VIA   metal3   (1 1)   SHADOW   (     /* VIA metal3 = "Obj" in figure 3.1 */
                                      ( VIA metal2  0  0)   /* "Sh" */
                                      ( VIA metal1  0  0)   /* "Sh" */
                                      )
```

This example solves the "stacked via" problem : no three vias can be placed above each other, but any combination of two vias (or one via) is allowed. The following extra complementary rules are generated internally.

example:
(cont'd)

```
DRL   VIA   metal2   (1 1)   SHADOW   (
                                      ( VIA metal3  0  0)
                                      ( VIA metal1  0  0)
                                      )

DRL   VIA   metal1   (1 1)   SHADOW   (
                                      ( VIA metal2  0  0)
                                      ( VIA metal3  0  0)
                                      )
```

Something looking similar to the example, but behaving quite different is :

(cont'd)

```
DRL   VIA   metal3   (1 1)   SHADOW   (   ( VIA metal2  0  0)   )

DRL   VIA   metal3   (1 1)   SHADOW   (   ( VIA metal1  0  0)   )
```

In this situation, the via in metal3 cannot be made if there's only one via below it. The complementary holds for vias in metal1 and metal2 : neither one can be made if the via in metal3 is present. Practice indicates that nearly all of the designrules can be expressed using the shadowing model.
The designrules given in this section have an 'additive' nature : they will be combined with the ones stated in the core cells to result in more restrictive rules.
It is important to note that default designrules preferably be specified in the core cells, because these cells are stored according to a grid model and will not take more storage space than without the designrule data. The other way, specifying default rules at the MASTER level, will yield enormous update-lists since for each point in the entire master area an entry will be created.

syntax:

```
<cost>          ::=   { "COST"
                            {    (   ( "HOR(IZONTAL)?" | "VER(TICAL)?" | " VIA " )
                                     <layer_name> <position_list> <copy_list>
                                     <inf_number>
                                 |   "BLOCKED"  <layer_name> <position_list>
                                     <copy_list>
                                 )
                            }+
                      }+.

<inf_number>    ::=   "INF(INITE)?" |<number>.
```

semantics:
The COST function guides the router by supplying costs for path extensions. The function defines (for each edge s) a cost F(s) associated with an extension of the path along s. The router tries to find a minimum cost path, and to be usable by the router function F has to abide to the consistency criterion :

Suppose point A is connected to B through a minimum cost path P, and B is connected to C through a minimum cost path Q, then if A is connected to C through B and the connecting path PQ has minimum costs, F is called consistent with respect to P and Q. If F has this property for every choice of A, B, C, P and Q, then F is called consistent.

For a more rigourous treatment of the consistency property, refer to [Rubin74] and [Lee61]
The consistency property implies that every path cost function F, where the costs to add an edge to the path P are independant of the edges P consists of, is consistent.
The cost function could have an arbitrary shape and could be dependent on anything, the task of the router demands a simple form to keep the computational overhead low. The syntax stated above can be used to model almost all practical cases, and the overhead for the router is limited to two table lookups, a test and an addition. This is explained easily: suppose the router has to connect adjacent points P and Q, and that it already incorporated point P into the path. The router then will take (being at point P) the cost specified at P in the direction of Q (a table lookup), and compare it to the cost it takes to go from Q to P (doing another table lookup for the costs specified at Q). The router will take the highest cost value, and adds this to the total path cost. The costs specified in the statements are related to the outgoing edges at the positions given; HORIZONTAL is an "east" or "west" edge , VERTICAL is a "north" or "south" edge, and VIA is a down- or upward edge. The word BLOCKED applies to the vertices at the given positions (not the outgoing edges), and is used to completely rule out the vertices for routing purposes.

example:

```
COST    BLOCKED    (0 0) CX (1 10) CY (1 10) /* blocks a 10x10 grid for routing */

COST    HOR        (0 0) (0 1) INFINITE    /* vertices still usable */
```

The costs specified at the MASTER level overrule those given in the core cells; but as in the case of designrules, it is important (with respect to storage space) to supply the default values in the core cells.

syntax:

```
<distances>   ::=   {     " DIST(ANCE)? "
                          {              ( " HOR(IZONTAL)? " | " VER(TICAL)? ")
                                         ( <def_number>
                                         | ( <number> "CP" "(" <number> <number> ")" )
                                         )
                                         <number>
                          }+
                    }+.
```

semantics:
DISTANCE statements specify (of course) the distance between gridlines perpendicular to the direction given. The distance itself lies in the direction given, and also the optional copy statement (CP or COPY) copies in that direction. DEFAULT applies to the entire region, a given linenumber sets the distance to this coordinate and the next one. Distance is independent of layers. The information is compiled into tables which will be used during mask generation.

examples:

```
DIST   HOR   DEFAULT   5                 /* all x coordinates are 5 units apart
                                         meaning that adjacent VERTICAL wires
                                         are 5 units apart */
DIST   VER   2         CP   (5 2)   3    /* distance between y=2 and y=3 is 3
                                         units, also for y=7 and y=8, and also
                                         for y=12 and y=13 */
```

syntax:

```
<widths>   ::=   {     " WIDTH "
                       {     ( " WIRE " | " VIA " | " HOLE " )
                             ( " HOR(IZONTAL)? " | " VER(TICAL)? " )
                             <layer_name>
                             ( <def_number>
                              | ( <number> "CP" "(" <number> <number> ")" )
                             )
                             <number>
                       }+
                 }+.
```

semantics:
This statement is much the same as a DIST statement, however measures of objects are addressed instead of distances in between. Also the orientation of WIDTHs given is different.

example :

```
WIDTH   WIRE   VER   metal3   DEFAULT   2
```

sets the width of vertical wires in metal3 to 2 units. But :

```
WIDTH    VIA    VER    metal3    DEFAULT    3
```

sets the dimension of the x - axis parallel border of the vias to 3 units. The same holds for holes. Of course, when in the last part of the example "VER" is exchanged for "HOR", the dimension of the y - axis parellel border will be addressed.

3.4 Syntax of the CORE_CELL section

In this section we will look at the syntax used to express CORE_CELLs in GADL.

syntax:

```
<core_cells>  ::=  {    "CORE_CELL" <core_cell_name>
                        "("   [<layers>] <dimension> [<wirings>]
                              [<drl>] [<cost>]
                        ")"
                   }+.
```

semantics:
Most of the constituents of a core cell have been mentioned in the previous section, so a few remarks here will suffice. First of all, the optional layer statements *overrule the statements in the MASTER section.* This might seem strange, but since the layer specification in the MASTER section cannot be postion dependent, it doesn't require much storage space. To bring in some position dependence the layers can be redefined inside core cells. Thus, if (in the region of this core cell) metal3 is no longer capable of accepting programmable vias, a statement might be :

```
LAYER metal3 PROG FIX
```

The statement does not always have to be more restrictive; equally well it can relax on some constraints given earlier. The dimension statement is mandatory at this place, for reasons mentioned in the previous section. Designrule and cost statements specified in core cells are defaults; they can be updated respectively overruled by constraints given in the MASTER section.

syntax:

```
<wirings>  ::=  {   ((   " WIRE " | " EQ " | " PWIRE " | " NWIRE " )
                     {   <layer_name> <position> <position_list>
                         <copy_list>
                     }+
                     |   ( " VIA " | " PVIA " | " NVIA " )
                          { <layer_name> <position_list> <copy_list>
                          }+
                     )
                }+.
```

semantics:
Core cells themselves cannot have NETs because they can be customized by different macro stamps. The overlaying wire pattern of a stamp will determine which wire inside a core cell belongs to which net.

3.5 Syntax of the MACRO section

This section details the syntax used to describe the macro library used to customize the image. Macros are considered being functional building blocks, each macro performs a logical function (e.g. a flipflop or a counter). Each stamp of a macro is an implementation (essentially a wire pattern) for the function of that macro. Stamps can be supplied to realize a function with different types of underlying core cells, or to pack a function in different geometric shapes.

syntax:

```
<macros>    ::=   { "MACRO" <macro_name> "(" <pins> <stamps> ")"  }+.

<pins>      ::=   { "PIN"  { <pin_name> }+ }+.
```

semantics:
Specific for a macro and global to its stamps are the PIN definitions, which specify connection terminals. A macro must have at least one pin. For each pin, each stamp must have a net associated with it. Pins are not allowed to have globally (at the MASTER level) defined names.

syntax:

```
<stamps>    ::=   { "STAMP" <stamp_name>  "(" (<stcopy>|<stdesc>) <legals> ")"  }+.

<stcopy>    ::=   ( "CP" | "COPY"  ) <stamp_name> <trans_list>.

<stdesc>    ::=   [<layers>] <dimension> <nets> [<npwirings>]
                  [<mcalls>] [<drl>] [<cost>].

<mcalls>    ::=   { " CALL "  { <macro_name> <stamp_name>
                  <position_list> <copy_list> }+ }+.

<legals>    ::=   { " LEGAL "  <position_list>  <copy_list> }+.
```

semantics:
Stamps come in two flavors, one kind defines a stamp by means of its layers, dimension, nets, etc. (see <stdesc>), while the other kind (<stcopy>) is used to create transformed instances of other stamps in the same macro. New is <mcalls> : this construction allows for stamps in other macros to be "called" at the coordinates given. Calls can be nested to an arbitrary depth (the called stamps calling other stamps). For each called stamp the compiler must be able to locate it, but the nesting will not get expanded. The main implication of this will be shown in the next example :

```
MACRO   jk_ff
   (
        PIN      j k q q_inv

        STAMP    narrow_jkff
           (
                 LAYER    ...
                 DIM      little_width  big_heigth
                 .
                 NET      j ....
                 NET      k ....
                 NET      q
                 NET      q_inv
                 NET      clock   /* global defined net */
                 NET      vsup    /* power supply also global */
                 .
                 .
                 CALL     nand2  narrow_nand ...  /* call a similar shaped nand */
                 LEGAL    (0 0) CX (20 5) CY (20 2)  /* valid pos. on the chip */
           )

        STAMP    wide_jkff
           (
                 COPY     narrow_jkff  R90
                 LEGAL    ....
           )
   )

MACRO   nand2
   (    PIN      ...
        STAMP    narrow_nand
           (     ....
           )
        STAMP    flattened_nand
           (
                 COPY  narrow_nand MX MY R270
                 LEGAL  ...
           )
   )
```

Now, when the compiler encounters macro *jk_ff*, it first substitutes every stamp that is constructed of a COPY statement by its original instance with appropriate transformations applied. All macros are treated this way, and the CALL statements (including those which are getting copied) are left alone. Next, in the second pass the compiler tries to resolve the CALL statements. If it doesn't succeed, the culprit here is that for stamps which were instantiated during the first pass, the copied calls also need to undergo the particular transformations. That means : the called macro needs to have a stamp which has undergone the same transformation. To clarify this, let us trace the compiler going through the example given:

i. Stamp *wide_jkff* will get the body (everything between ( )) of stamp *narrow_jkff* filled in, except for the LEGALs. Every component to which this applies will be transformed R90 during this process. The CALL statements are just copied. The same process will happen to stamp *flattened_nand*, it will get the body of stamp *narrow_nand* and the transformations

sequence reduces to R90.

ii. Next, the CALLs will be resolved. For stamp *narrow_jkff* this is straightforward. Stamp *wide_jkff* however has a call to macro *nand2* stamp *narrow_nand*, and this call remains to be transformed. The compiler now will look for an R90 transformed instance of stamp *narrow_nand*, and it will find that stamp *flattened_nand* can be substituted. Stamp *wide_jkff* will now look like :

```
STAMP   wide_jkff
   (
        LAYER ...
        DIM  big_heigth little_width  /* x and y swapped */
        .
        NET j ....
        NET k .... /* all is transformed R90 */
        NET q
        NET q_inv
        NET clock   /* global defined net */
        NET vsup    /* power supply also global */
        .
        .
        CALL nand2  flattened_nand .. /* call the new found nand */
        LEGAL ....               /* original LEGAL */
   )
```

So, when using CP statements in stamp definitions special attention must be paid to the resulting CALLs. It is impossible to create a new stamp when a needed instance cannot be found. The positions on the image where the stamp can be placed (that is, the particular wiring pattern can be used there for customization) are specified in the <legals> statement. With this statement the links between the wiring patterns and the underlying core cells are established for the other components of GAS.

4. Implementation of the compiler.

4.1 The compilation process.

Before compilation can start, the GADL source file must be parsed and translated into a parse tree. To accomplish this, the source file is split into tokens by a lexical scanner constructed with the Unix tool Lex, and a parser built with Yacc uses these tokens to construct a parse tree. Because the previous chapter outlined the language definition and the parser construction is automated, we won't go into any detail of the scanning and parsing processes, but instead our main focus in this chapter is the subsequent compilation process. Following the parsing phase should come a semantic analyses phase. However, because of the limited amount of time available, it was felt more important to build a working compiler and add the semantic analyses afterwards. So, the current compiler has no substantial semantic analyses, and therefore is not quite responsive to errors.
Next comes the actual compilation process, in which we can distinguish three phases:

- core cell compilation
- macro compilation
- technology compilation

Of course, due to the language definition, these phases will have some routines in common.

The compilation process translates the statements given in GADL into structures and tables in the *target* database. This database is an integral part of the Gate Array placement and rotuing System, and it must be seen in a broader scope than just a place to keep data stored. As part of the database must be considered a comprehensive library of functions and procedures to perform a wide range of tasks with and within the database. An obvious advantage of this library is the fact that it hides the internals of the structures in the database; future (minor) modifications can be carried out in the library routines instead of in every module in GAS. Furthermore, the presence of such a (tested) library greatly reduced the efforts needed to develop the GADL compiler.
To avoid long-windedness, we will not discuss the routines to the smallest detail, but instead mention and discuss the most important subroutines.
To describe the algorithms employed, a meta language presented in appendix B will be used.

4.2 Compilation of Core Cells.

Before the compilation can take place some initializing is done by a routine called *install_layers*; it puts the number of layers and their characteristics in the database.

The compilation of core cells starts by calling procedure *compile_core_cells()*.

synopsis :

    void compile_core_cells()

description :
This routine compiles the core cells which are called in the MASTER section. If necessary, new instances of core cells will be made.

algorithm:

```
BEGIN
    FOR each call in the MASTER block DO
        IF transforms specified THEN reduce_transforms
        FI;
        IF a transform left THEN append transform to core_cell name
        FI;
        IF not compiled already  { search based on the adapted core_cell name }
            THEN
                search for the core cell in parse tree;
                expand its image statements; { remove CX ,CY of wiring statements }
                IF a transform left THEN copy and transform the core cell;
                FI;
                compile the core cell;
                compile the positions in the call statement;
                add the compiled cell and the compiled positions to the target database;
            ELSE
                locate compiled core cell;
                compile the positions in the call statement;
                add pointer to compiled cell and the compiled positions to the target database;
        FI;
    ROF;
END;
```

Only the core cells which are called in the MASTER block are compiled. First, the call is checked for specified transformations, and if found they will be reduced to a primitive. Next, the character representation of this primitive (if not the identity transformation) will be concatenated with the original core cell name. This new name will be searched for in the target database. If not found, this implies that the needed instance must be created.
The image statements must be expanded because the wire-like objects are stored in the target database as sequences of coordinates.

subroutines:

```
void redtfr();
CORE_DESC  *transform_core ();
CORE_CELL  *compile_core ();
```

— routine *redtfr()* :
This routine is based on the fact that an orthogonal placed rectangle in 2-space can have eight orientations, shown in figure 4.1

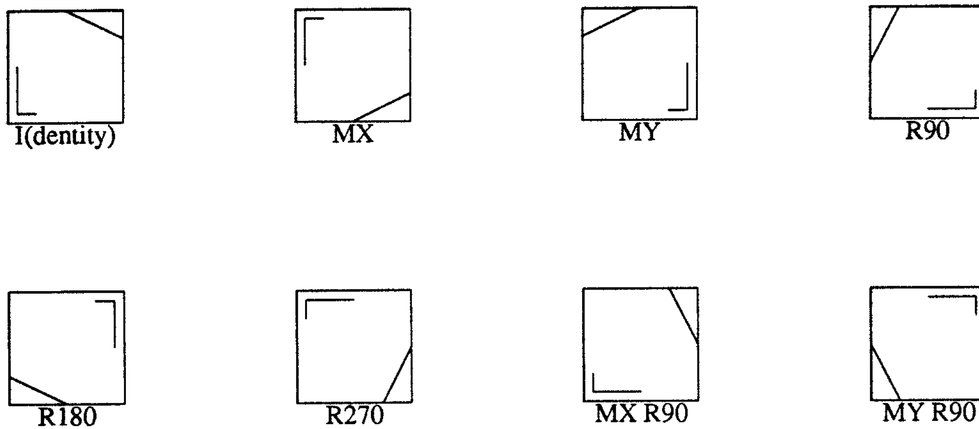


Figure 4.1. Orthogonal orientations of a box in 2-space.

Internally "MX R90" is dubbed "TAU1", and "MY R90" is called "TAU2". Since each basic transformation supplied in GADL leaves the sides of a rectangle in parallel with the x and y axis, clearly each arbitrary length sequence of transforms does. Moreover, the result of such sequences always boils down to one of the configurations shown above. Hence it was decided to implement the transformation process in two phases. First, the transformations list will be reduced (by routine *redtfr()*) to one of the transformations shown above using a table lookup. Next, when the transformation needs to be applied (by other routines), a simple representing transformation matrix will be used to manipulate objects. The table used in the reduction process is shown in table 4.1.

TABLE 4.1. The transforms reduction table.

	T2						
T1	MX	MY	R90	R180	R270	(MX R90)	(MY R90)
MX	I	R180	(MX R90)	MY	(MY R90)	R90	R270
MY	R180	I	(MY R90)	MX	(MX R90)	R270	R90
R90	(MY R90)	(MX R90)	R180	R270	I	MX	MY
R180	MY	MX	R270	I	R90	(MY R90)	(MX R90)
R270	(MX R90)	(MY R90)	I	R90	R180	MY	MX
(MX R90)	R270	R90	MY	(MY R90)	MX	I	R180
(MY R90)	R90	R270	MX	(MX R90)	MY	R180	I
							T1 T2

Since all objects in the GAS are taken to have their origin (0,0) in the lower left corner, each coordinate transformation matrix is supplemented by a second one which adds the offsets needed to put a transformed object back into the first quadrant.

— routine *transform_core()*:
This routine transforms the body of the core cell according to the transformation resulting from the reduction. That is, wire-like statements are transformed position by position, designrule- and cost statements will have their position, copy and offset parts adapted. The routine creates a new instance of the core cell and returns a pointer to it.

— routine *compile_core():*
At this place, all components of a core cell subsequently will get compiled. To this end, first a grid structure is allocated and the layer characteristics are set to default or to local updated parameters. Next, the image statements are translated, followed by the designrule and cost statements.

subroutines:

```
void compile_core_image();
void compile_core_aux();
```

— routine *compile_core_image:*
The compilation here mainly boils down to setting up the right frames for the calls to the library routines. For wire-like statements the library routine *write_edge_status* is called to update the grid. Equivalence statements are translated into an appropriate intermediate format, and are handed to the library routine *compile_core_eq_set*, which will check them against the current equivalence table in the target database, and will add them if necessary.

— routine *compile_core_aux():*
This routine translates the cost- and the designrule statements. The cost statements are given position by position to the library routine *compile_core_cost_set*, which updates the grid of the current core cell and also updates the cost table in the database.
The designrule statements are given to *compile_drl* which will later be discussed in greater detail.

4.3 Macro compilation.

The compilation of macros and stamps is somewhat more involved than the compilation of core cells. This stems from the hierarchy in the language: stamps can be nested to an arbitrary depth. This fact gives reason for the compilation to be carried out recursively rather than "flat" as is done with the core cells. Preliminary to the compilation, some parts of the MASTER block are translated into a macro and an accompanying stamp.
Hereto routine *create_master_macro()* takes the following actions:

```
BEGIN
    allocate MACRO and STAMP structures; { the MASTER macro and the MASTER stamp }
    assign them appropriate names;
    insert the structures in the parse tree;
    FOR each NET statement in the MASTER block DO
        allocate a similar named PIN;
        expand the image statements; { remove CX ,CY }
    ROF
    expand image statements of NWIRE,PWIRE,NVIA,PVIA  statements;
    IF dimensions not specified in MASTER
        THEN compute dimensions from CALL statements;
    FI;
    make LEGAL position (0,0);
END
```

Next, routine *compile_macros()* separates the compilation of the macros into three parts:

- substitution for stamps being copies of other stamps
- resolution of the call statements used
- translation to the target database and computation of net identifications

In the material following, each of these subjects will be discussed in detail.

4.3.1 Substitutions for stamps consisting of a copy statement

Routine *compile_macros()* immediately after entry calls a subroutine to do the substitutions.

synopsis:

```
void reduce_stamp_calls();
```

description:
As already stipulated in the previous chapter, the stamps which are constructed with a CP or a COPY statement will have the indicated stamp copied. If the indicated stamp itself is also constructed with a copy statement, the search will continue until a stamp is found which is defined using the <stdesc> construction in GADL.

algorithm:

```
BEGIN
    FOR the 2nd macro in the list to the last one DO
        { the first is the macro created from the MASTER }
        FOR each stamp in the current macro DO
            IF the current stamp is constructed with a copy statement
                THEN
                    make search_stamp equal to the current stamp;
                    WHILE search_stamp is constructed with a copy statement DO
                            make search_stamp equal to the stamp to be copied;
                            accumulate and reduce transformations;
                    ELIHW;
                    expand the copylists in the search_stamp; { expands CX,CY in wire-like
                    statements }
                    copy the body of search_stamp to the current stamp while applying the
                    resultant transformation;
                    register that the current stamp is a copy; { CP is removed }
                    { subsequent searches for other stamps then will continue to the source of
                    the current stamp }
                ELSE
                    expand the copylists in the search_stamp; { expands CX,CY in wire-like
                    statements }
            FI;
        ROF;
    ROF;
END
```

subroutines :

```
STAMP_DESC *find_stamp();
int int_redtfr();
```

— routine *find_stamp() :*
This routine locates a stamp in the current macro, it searches by name.

— routine *int_redtfr() :*
same as *redtfr()*, but returns a constant representing the resultant transformation.

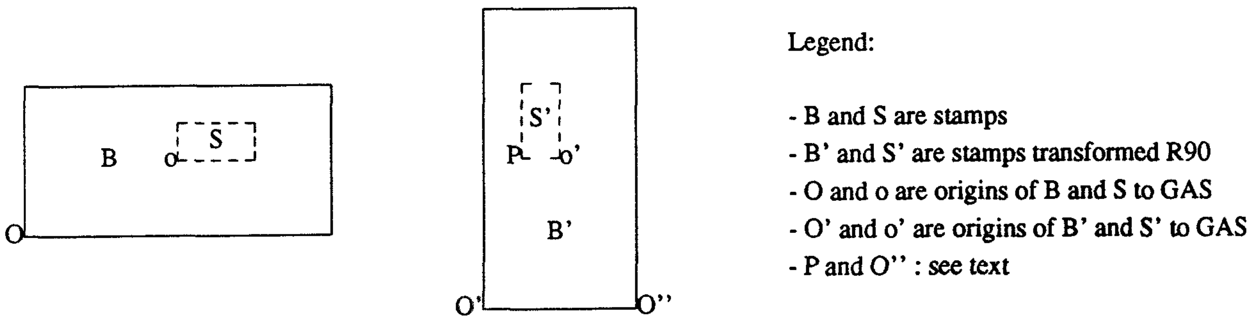
4.3.2 Resolution of the call statements in stamp definitions.

When the compiler reaches this point, all stamps in the parse tree are defined according to the <stdesc> rule of GADL. In the previous chapter it was briefly indicated how the compiler solves the calls to other stamps. Routine *trace_stamp_calls()* is called by *compile_macros()* to perform the resolution of calls.

synopsis:

```
void trace_stamp_calls();
```

description: The routine checks every stamp in every macro for the presence of one or more calls. If found then the called stamp will be searched for in the parse tree. The dimensions of the stamp found are used to create correction offsets for the position list in the call statement currently being resolved. To see why this is needed, consider a small example:



— First, consider S being defined as a part of B (that is, S being not explicitly called as a stamp). Then O is the origin of B, and o is not explicitly known. When B is transformed R90, internally the point o shifts to o'. Since S is an integral part of B, and thus S' is an integral part of B', there is no problem. O is transformed (to O'') but corrected to become the new origin O'.

— Now consider the other case, one of the statements defining B is a call to S at point o. This point thus is known explicit, and when R90 is applied to B like before it will become o'. The same transform applied to S will yield S' having P as its origin (it is corrected). Since the positions listed in a call statement are to be the anchor points of the origins of the called stamps, o' is wrong and it should be P. The additional correction to shift o' to P is the correction mentioned above.

If the called stamp turns out to be a copy of another stamp, the search continues to locate the "source" stamp. This is done based on the registration made in the substitution process. When the right stamp is found, the call is modified by subroutine *transform_stamp_call()*.

subroutines:

```
void transform_stamp_call();
STAMP_DESC *find_src_stamp();
MACRO_DESC *find_macro();
```

— routine *transform_stamp_call()*:
The routine simply sets the macroname and stampname fields of the call to the macro and stamp just found; furthermore it adjusts the positions in the position list with the correction offsets mentioned earlier, and it expands the copy list if it is specified with the call.

— routine *find_src_stamp()*:
This routine locates a stamp by a name and a transform given. The routine returns a pointer to the stamp which is *equivalent* to the stamp of the name given transformed by the transform given.

— routine *find_macro()*:
Locates a macro (by its name) in the parse tree.

4.3.3 Translation to the target database and computation of net identifications.

The last phase in the macro compilation process is performed by routine *translate_macros()*.

synopsis:

```
void translate_macros();
```

Due to the size of this part of the compiler, let us first consider the algorithm for the translation.

algorithm:

```
BEGIN
    make a list of connection pins global to the gate array; { pins of the MASTER - macro }
    allocate space for the MASTER macro;
    compile the MASTER stamp;
    FOR every macro DO compile that macro;
    ROF;
END
```

description:

The MASTER macro is compiled separately because the presence of a list with global pins is assumed for the compilation of the remaining macros. The body of the routine is a FOR loop; this may seem a contradiction with earlier announced recursive principle. In the next subsection, this structure will be dealt with in more detail.

subroutines:

```
MACRO *compile_mac();
void compile_stamp();
```

— routine *compile_mac()*:

This routine delivers a completely compiled macro. Its internals are detailed in the following algorithmic description.

algorithm:

```
BEGIN
    IF the given macro is not in the target database THEN allocate and name it;
    FI
    FOR every stamp in the given macro DO
        IF current stamp not in target database THEN compile the current stamp;
        FI;
    ROF;
END
```

explication:
The most important detail of this routine is the fact that in case it cannot find the macro in the target database, it immediately allocates a new macro without filling in the stamps, instead of first compiling all stamps and then allocating space for the macro (afterall, the macro structure is little more than a kind of header to the list of stamps). This approach is chosen as to mark the macros in the target database as "processed", so the recursion (in the next routine) will not again try to compile it.

subroutine:

```
void compile_stamp();
```

— routine *compile_stamp()*:

Will deliver a completely compiled stamp.

algorithm:

```
BEGIN
    collect all NWIRE,NVIA,PWIRE and NWIRE statements and
    translate them to the target format;
    allocate space for the target stamp;
    convert the NETs to the target format;

    fill the updates for layers, the legals and
    the transformed nets in the target stamp;

    FOR every call in the current stamp DO
        search called macro in target database;
        IF not found THEN compile the requested macro;
        FI;
        { at this point the requested macro and stamp must be compiled }
        locate the requested stamp in the target database;
        FOR all positions in this call DO
            add the call at the current position to the target stamp;
        ROF;
    ROF;
    make equivalence updates;
```

```
        make cost- and designrule updates;
        IF the current stamp has calls THEN compute net identifications;
        FI;
    END
```

explication:

The compilation the stamp has two parts : the first part can be done independent of the calls, and the remaining things have to be done after the calls are completed. The first part is quite simple, we translate all wire-like statements to the target format (exept for the EQ statements), ordering the NETs as they occur in the list of pins. Also, the legal statements will be translated.
Next, the calls (if present) are processed. In this processing, the recursion can occur : when a call to a not yet compiled macro is found, the routine *compile_mac()* is called to compile it. On return, the macro is compiled in its entirety, and the requested stamp can be located. Of course this only holds for a macro which is not the current macro, since calls to stamps inside the current macro are understood to be self-loops, which are not allowed. The calls are completed by linking them into the target stamp. When this point is reached, the equivalence-, cost- and designrule statements can be dealt with, because the library routines which update the respective fields in the stamp will consult the just completed calls to other stamps. To complete the compilation of a stamp, the net identifications must be computed if there are calls in the current stamp.

subroutines:

```
void get_npwires();
LAYER_UPDATE *make_layer_update();
LEGAL *make_legals();
NET *make_nets();
MACRO *compile_mac();
void make_eq_update();
void make_aux_update();
void make_netids();
```

— routine *make_eq_update()*:

Translates the positions of the equivalent statements from each net in the stamp to an intermediate format, then hands them to library routine *compile_stamp_eq_set()*. This latter routine will fill in the update-field of the stamp.

— routine *get_npwires()*:

Translates the NWIREs,NVIAs,PWIREs an PVIAs into the target format.

— routine *make_layer_update()*:

For each layer a target structure is allocated, layer characteristics are filled in according tc definitions in the target database.

— routine *make_legals()*:

For each position in the position list of a LEGAL statement, a structure in the target format is allocated, and the copy statement (if present) is converted to a range and filled in. This is repeated for each legal statement.

— routine *make_nets()*:

This routine orders the nets according to the following list:

- first, all nets corresponding to pins of the current macro, in that order,
- next, all nets having global defined names, in the order of the pins of the MASTER macro,
- the remaining nets.

Then it will translate them into the target format.

algorithm:

```
BEGIN
    FOR each pin in the pinlist of the current macro DO
        locate its matching net, and mark it;
        translate and add the net to the list in the target format;
    ROF;
    IF the current stamp is not the MASTER stamp THEN
        FOR each pin in the global pinlist { pins of the MASTER MACRO } DO
            locate its matching net {in the current stamp}, and mark it; translate and add the
            net to the list in the target format;
        ROF;
        FOR all remaining unmarked nets DO
            translate and add the net to the list in the target format;
        ROF;
    FI;
END
```

explication:
The nets are ordered according to the pinlists; this order must be enforced to facilitate the computation of net identifications.

— routine *make_eq_update()*:

Translates the positions of the equivalent statements from each net in the stamp to an intermediate format, then hands them to library routine *compile_stamp_eq_set()*. This latter routine will fill in the update-field of the stamp.

— routine *make_aux_update()*:

The cost and designrule updates are made by this routine.

algorithm:

```
BEGIN
    FOR every aux { cost or drl } statement DO
        IF the current statement is a COST statement
            THEN
                get the appropriate layer;
                determine orientation;
                FOR all positions in the positions list DO
                    compile_stamp_eq_set(); { a library routine }
                ROF;
            ELSE
                compile_drl();
        FI;
    ROF;
END
```

subroutines:

void compile_stamp_eq_set();
void compile_drl();

— routine *compile_stamp_eq_set()*:

This library routine computes the updates for the stamp. It does so by scanning the cost table for the update to be made. If found, the updates will get the corresponding index, otherwise a new entry is made and assigned to the new updates.

— routine *compile_drl()*:

As mentioned in the previous chapter, for the compilation of designrules the complementary rules have to be calculated. This routine does so, and translates the rules into the target format. This routine is called by *make_aux_update()* (compilation of stamps) as well as *compile_core_aux()*.

algorithm:

```
BEGIN
    get layerindex;
    make edge set of all SHADOW clauses and sort it;
    IF compilation of a stamp
        THEN
            FOR all positions DO
                compile_stamp_dr_set(); { library routine }
            ROF
        ELSE
            FOR all positions DO
                compile_core_dr_set(); { library routine }
            ROF
    FI;
    make layerindexes in the edge set absolute; { x and y are already absolute to (0,0) }
    FOR each edge in the edge set DO
        current_pos = current edge;
        current edge = (0,0,layerindex);
        FOR each edge in the set DO
            make the current edge relative to current_pos;
```

```
                { "current edge" in this inner loop }
            ROF;
            copy the edge set, and sort the copy;
            FOR each edge in the set DO
                make it absolute again { for subsequent iterations }
            ROF;
            IF stamp compilation
                THEN
                    FOR each position in the poslist DO
                        compile_stamp_dr_set() with respect to current_pos,
                        using the copied edge set;
                    ROF;
                ELSE
                    FOR each position in the poslist DO
                        compile_core_dr_set() with respect to current_pos, using the copied
                        edge set;
                    ROF;
            FI;
            restore the contents of current edge from current_pos;
        ROF;
    END
```

explication:
The complementary rules are computed with respect to a current position. Initially, the current position is taken from the positionslist, and the SHADOW clauses have coordinates relative to it. To compute the complementary designrules, the shadow edges are cyclically taken to be the current position, and substituted for that particular edge is the origin (0,0,layerindex). Next, all the edges are made relative to the current position, that is, the current position is subtracted from each. Then a library routine is called, and the positions it is given are taken from the positionslist, adjusted for the fact that now they have shifted by the offsets given in the current position (at a shadow edge). This process is repeated for each shadow edge.

subroutine:

```
DR_EDGE *sort_edges();
```

Sorts and copies the list of edges given into a new list; returns a pointer to the new list.

— routine *make_netids()*:

This routine computes the net identifications. These are numbers which identify nets of the current stamp with pins of the called modules

algorithm:

```
    BEGIN
        FOR each pin of the current stamp DO
            expand the pin's net;
        ROF
        FOR each called stamp DO
            FOR each pin of the current called stamp DO
                expand the pin's net;
            ROF;
            allocate a list for the net_ids of the called stamp to be stored;
            set all net_ids in the list to BLOCKED_VERTEX; { not connected yet }
            called_net=0;
```

```
                FOR each expanded net of the called stamp DO
                    current_net=0;
                    FOR each expanded net of the current stamp DO
                        IF a connection between the nets
                            THEN net_id[called_net]=FIRST_NETID+current_net;
                        FI;
                        increment current_net;
                    ROF;
                    increment called_net;
                ROF;
            ROF;
        END
```

explication:
The routine starts with expanding each net in the current stamp which belongs to a pin of the macro. The expansion is discussed later in more detail, but for now it suffices to say that each and every position a net occupies (including the positions generated by equivalences) is put into a list, and this list is sorted. Next, each call is processed subsequently. For each stamp called the net expansion procedure is applied to its nets. Having the lists of nets prepared, we can check connectivity by comparing pairs of positionlists, and this is done in the two innermost FOR loops. The comparison is quite simple: since each list of positions is sorted, it comes down to a comparison per component, and if not equal advancing the list having the smallest position. When a connection is found, the list of net_ids belonging to this call is updated.

subroutines:

```
int connected();
WIRE *expand_net();
```

— routine *connected()*:

As indicated previously, this routine walks along the two positionlists it is given, and it returns a 1 if a common position is found, and 0 in the other case. To compare positions the routine calls a subroutine *position_cmp()*.

— routine *expand_net()*:

Each net is represented by a list of WIREs. Now, each wire is expanded into a list which contains literally every position it occupies on the gate array. A list having all the positions of a net results.

algorithm:

```
BEGIN
    FOR each WIRE in the net DO
        IF the current wire has only one position
            THEN
                allocate that position in the list;
                get all the equivalent positions for the current position,
                and add them to the list;
            ELSE
                FOR each position in the wire, exept for the last position DO
                    put the current and the next position in the list, and also put in their
                    equivalent positions;
                    IF the wire segment between the current and the next position is
                    orthogonal
                        THEN
```

```
                                    FOR each position between the current and next position
                                    DO
                                        put the intermediate position and its equivalent
                                        positions in the list;
                                    ROF ;
                            FI;
                        ROF;
                FI
            ROF;
            sort the created list;
            remove multiple equal positions from the list;
        END
```

subroutines:

```
        POSITION *get_eqpos();
        POSITION *sort_position_list();
        POSITION *reduce_position_list();
```

— routine *get_eqpos()*:

In this routine, a library routine *stamp_scan_set()* is called to collect the equivalent positions for a given position. Next, the collected positions are inserted into the list given.

— routine *sort_position_list()*:

Sorts a list of positions using the heapsort method.

— routine *reduce_position_list()*:

Removes multiple equal positions from a given list of positions.

This concludes the compilation of stamps.

4.4 Technology compilation.

This part of the compiler translates DISTANCE and WIDTH statements of the MASTER block into the format of the target database. Due to the presence of library routines this is not at all difficult; a short discussion will suffice.
To compile the technology statements, the main program calls the routine *compile_technology()*.

synopsis:

```
void compile_technology();
```

description:
Will translate all the technology information to the target database.

algorithm:

```
BEGIN
    allocate space for the technology information;
    FOR each technology statement in the MASTER block DO
        get the technology object;
        IF the current statement is a DEFAULT statement
            THEN
                IF it is a DISTANCE statement
                    THEN store_default_distance()
                    ELSE store_default_width()
                FI;
            ELSE
                IF it is a DISTANCE statement
                    THEN
                        FOR each line specified DO store_distance()
                        ROF;
                    ELSE
                        FOR each line specified DO store_width()
                        ROF;
                FI; ROF;
        FI;
    ROF; compute_distances();
END
```

explication:
First, the routine makes space available for the storage of technology data by calling the library routine *store_tech_data()*.
Next, for each statement encountered the appropriate action is taken, which is clearly stated in the algorithmic description. The routine concludes by calling the library procedure *compute_distances()*. This latter procedure makes up the format of the target table to the requirements to the other GAS components.

subroutines:

```
void store_tech_data();
void store_default_distance();
void store_default_width();
void store_distance();
void store_width();
void compute_distances();
```

— routine *store_tech_data()*:

Allocates space in the target database for the technology data to be stored.

— routine *store_default_distance()*:

Does as its name says, stores default distances in the table.

— routine *store_default_width()*:

Same as above, for widths.

— routine *store_distance()*:

Assigns a distance to specified gridlines.

— routine *store_width()*:

Same as above, for width.

— routine *compute_distances()*:

Formats the distances table in the target database to the requirements of the other GAS components.

5. Conclusions

- The new definition of the GADL facilitates an easy and concise way to describe the features of gate arrays. The constucts provided for the specification of designrules and costs are quite simple, but powerful enough to model most constraints. Furthermore, the hierarchy in the language enables the user to describe a big gate array core and its accompanying macro library in a few pages of text.

- The storage of coordinates and a compact representation of tables instead of a storage of the full grid expansion greatly reduces the size of the GAS database.

- The new compiler will generate most of the data required by the other modules in the GAS. However, a great amount of consciousness regarding the syntax and semantics of the GADL inputfile is required from the user; the compiler has very limited capabilities for catching errors.
  Even a full featured compiler has limits in this respect; for example it has no way to determine if a given stamp does fit onto the underlying area of the image.

- Although the compiler is perhaps the fewest used tool in the GAS, further development regarding error detection and recovery is recommended; special attention should be paid to the handling of syntax errors by the parser, see for example [Schrein85].

Literature

[Frankle86]
Frankle, J. and R.M. Karp, *Circuit placement and cost bounds by eigenvector decomposition,* Proc. International Conf. on Computer Aided Design, pp. 414 - 417, Santa Clara, 1986.

[Huij90]
Huijbregts, E.P., and A.G.J. Slenter,*Flexible Module Generation For SoG Gate Arrays,* in: Proc. of second symposium on Design Methodolgy, Dalfsen (NL), April 1990.

[Jess86]
Jess, J.A.G. and A.G.J. Slenter, *The prototype of an open design system for gate arrays* in *ESPRIT '86, Results and achievements*, pp. 541 - 550, 1986.

[Jon84]
Jongen, R. *Simulated Annealing in Placement*
Master Thesis, Eindhoven University of Technology, 1984

[Lee61]
Lee, C.Y. *An algorithm for path connections and its applications*
IRE Trans. Electron. Computers,vol. EC-10, 1961, pp. 346-365.

[Lip86]
Lippens, P.E.R. *GADL, a gate array description language.*
Master Thesis, Eindhoven University of Technology, 1986.

[Otten84]
Otten, R.H.J.M. and L.P.P.P. van Ginneken, br *Floorplan design using simulated annealing*
Proc. International Conf. on Computer Aided Design, pp. 96 - 98, Santa Clara, 1984

[Rubin74]
Rubin, F. *The Lee path connection algorithm.*
IEEE Transactions on computers, vol. C-23, no. 9, Sept. 1974.

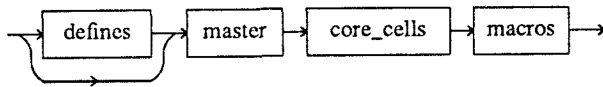
[Schrein85]
Schreiner, A.T. and H.G. Friedman Jr. *Introduction to compiler construction with Unix.*
Prentice Hall, Englewood Cliffs NJ 07632, 1985.

[Slen85]
Slenter, A.G.J. *Local routing of gate arrays*
Master Thesis, Eindhoven University of Technology, 1985.

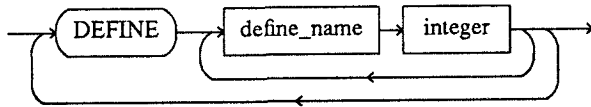
[Slen90]
Slenter, A.G.J. *A Generalised Approach to Gate Array Layout Design Automation*
PhD thesis, Eindhoven University of Technology, to be published.

Appendix A : Syntax diagram of GADL

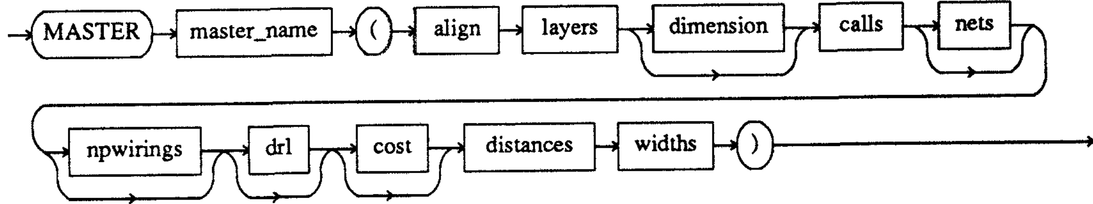
sentence:



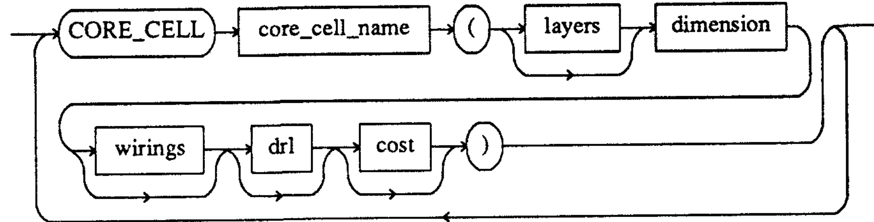
defines:



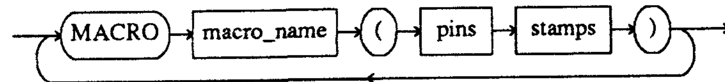
master:



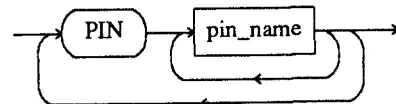
core_cells:



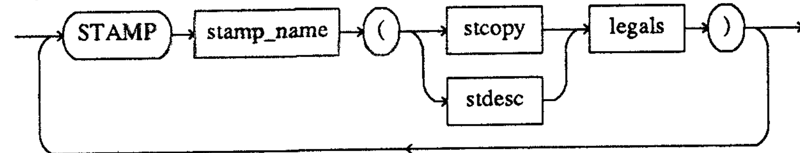
macros:



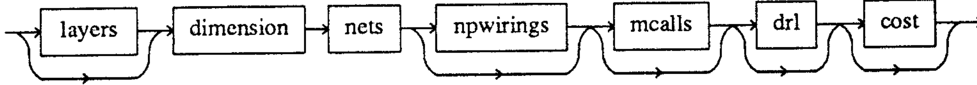
pins:



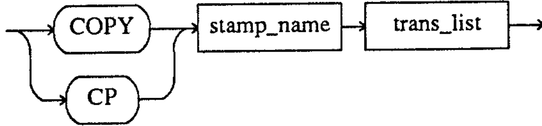
stamps:



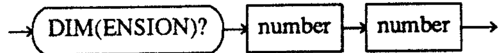
stdesc:



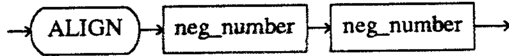
stcopy:



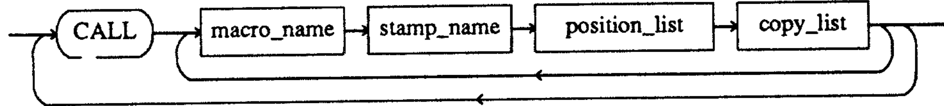
dimension:



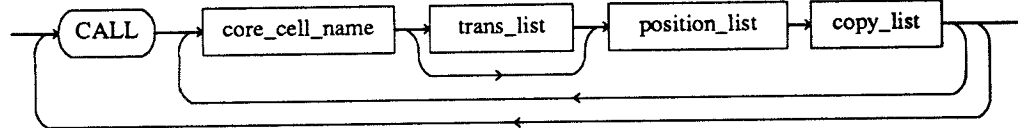
align:



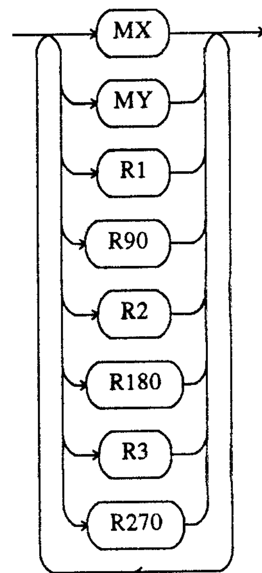
mcalls:



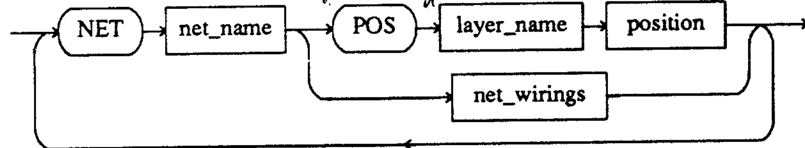
calls:



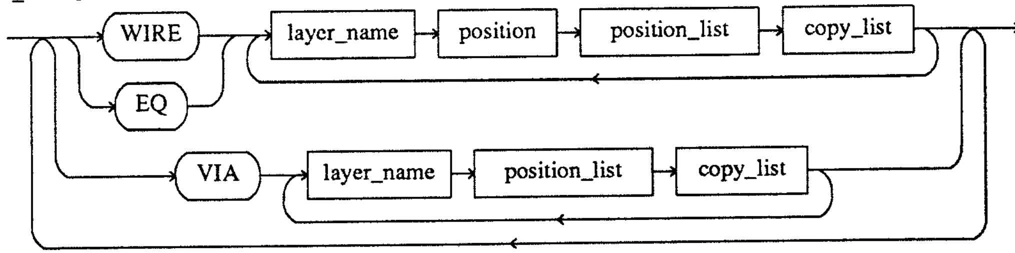
trans_list:



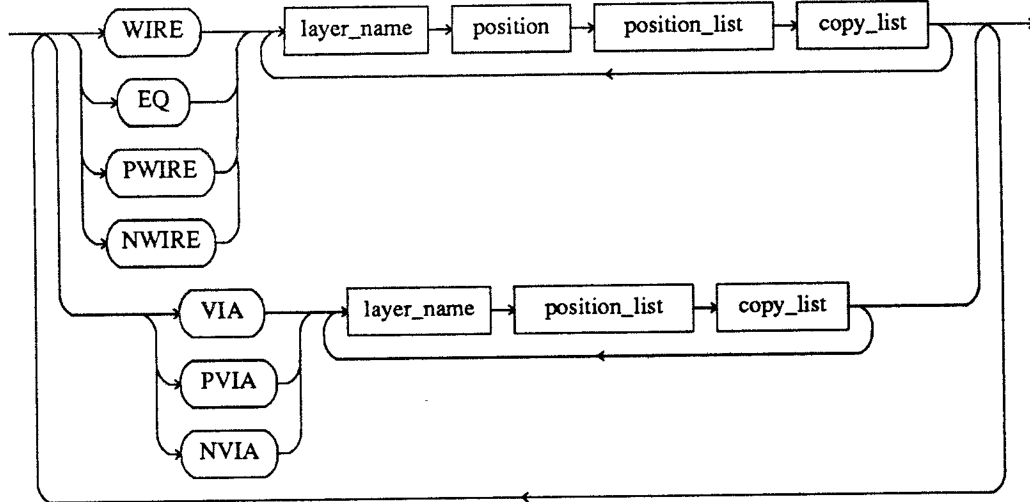
nets:



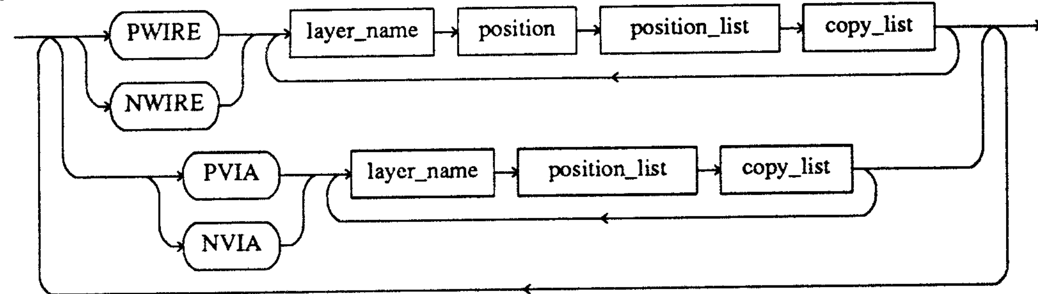
net_wirings:



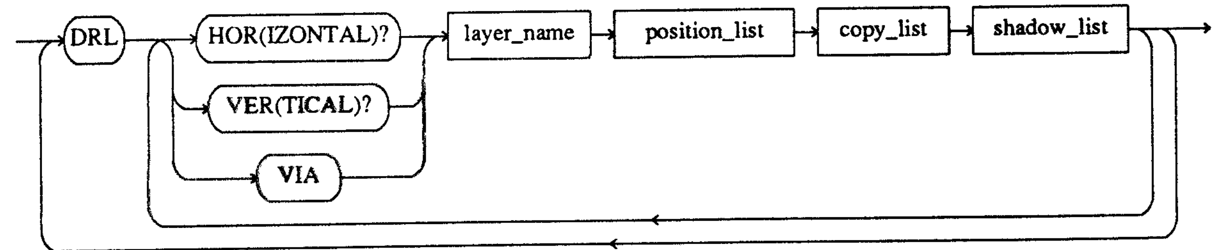
wirings:



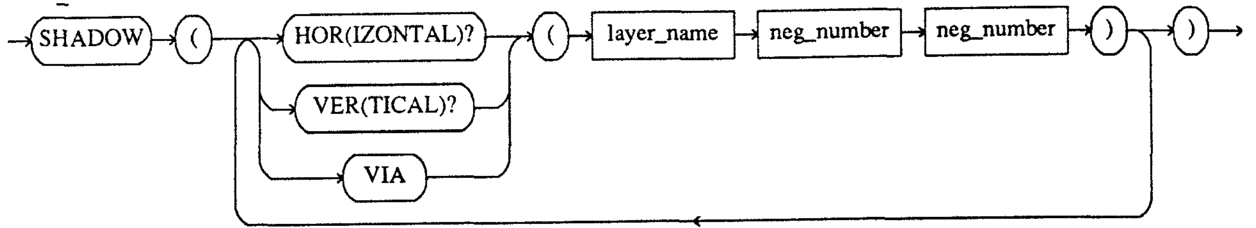
npwirings:



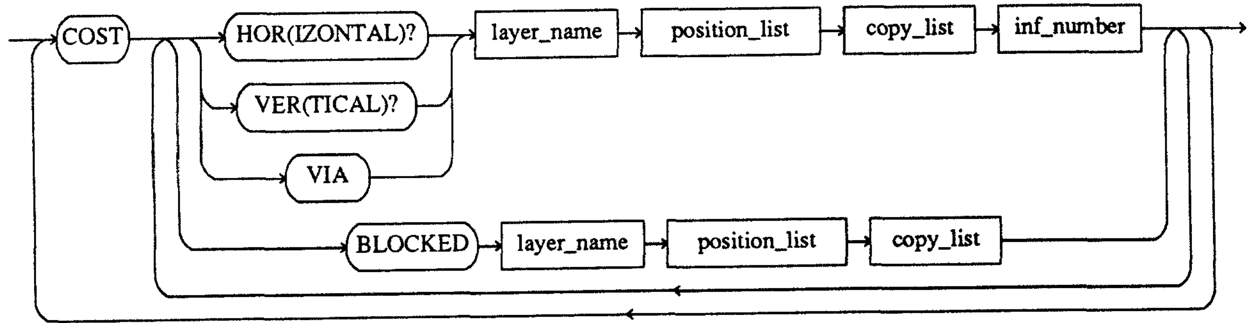
drl:



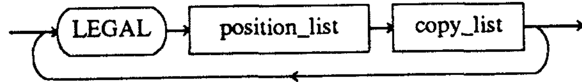
shadow_list:



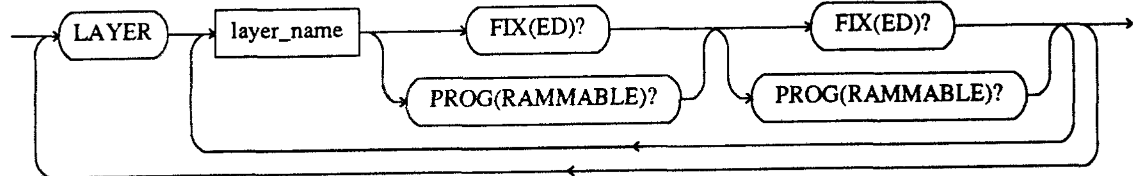
cost:



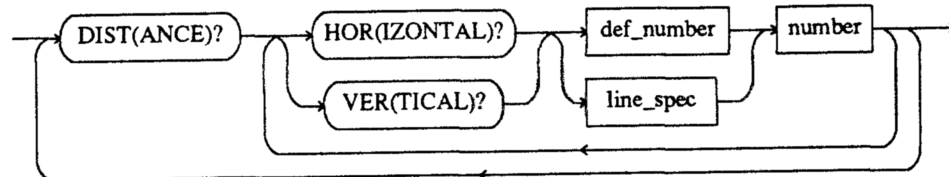
legals:



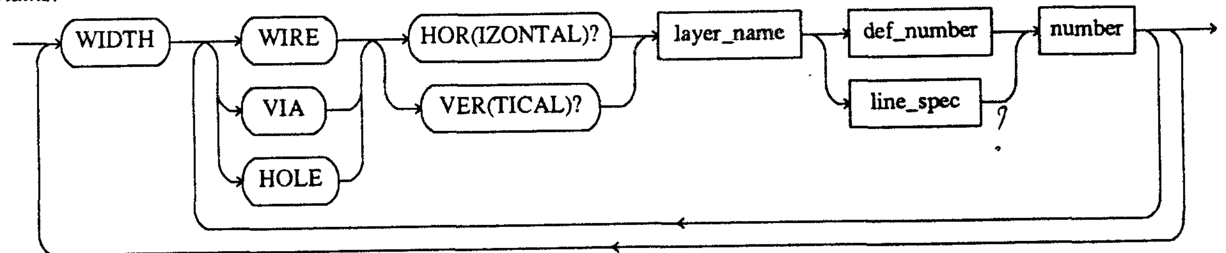
layers:



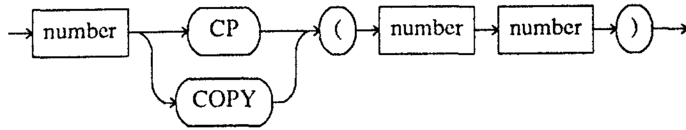
distances:



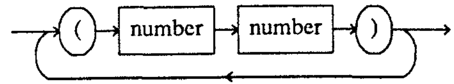
widths:



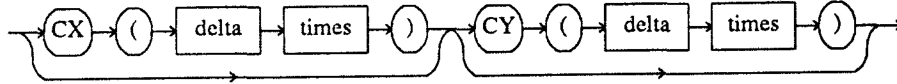
line_spec:



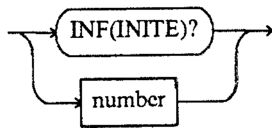
position_list:



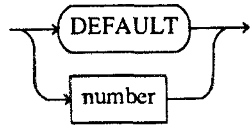
copy_list:



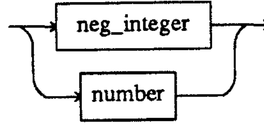
inf_number:



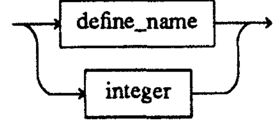
def_number:



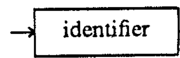
neg_number:



number:



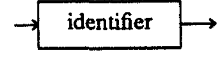
master_name:



core_cell_name:

macro_name:

stamp_name:

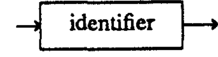


pin_name:

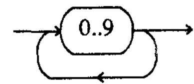
define_name:

net_name:

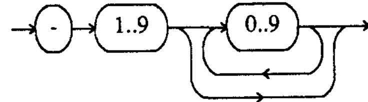
layer_name:



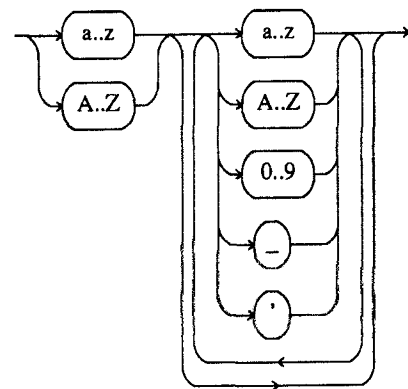
integer:



neg_integer:



identifier:



Appendix B: algorithm description meta language.

The language is a 'high level' meta language, it has only six statement kinds :

Action specifier:

```
< text >;
```

<text> specifies the action to be taken; it is expressed in natural language.

IF statement:

```
IF <test> THEN <statement(s)>
FI;
```

<test> and <statement(s)> are both formulated in a natural language. If the test is obeyed, the action(s) defined by <statement(s)> will be executed.

IF - ELSE statement:

```
IF <test> THEN <stat1>
       ELSE <stat2>
FI;
```

If <test> is obeyed, <stat1> will be executed, else <stat2> will be executed.

FOR statement:

```
FOR <set_description> DO <stats>
ROF;
```

<set_description> specifies a set of elements (could be anything). <stats> will be executed for every element in the set.

CASE statement:

```
CASE <var> OF
                         val_1 : <stats_1>
                         val_2 : <stats_2>
                           .
                           .
                         val_n : <stats_n>
ESAC;
```

<var> is an enumerated type of variable. If var has value <val_i>, the statements <stat_1>..<stat_i> will be executed.

WHILE statement:

```
WHILE <test> DO  <stats>
ELIHW;
```

<stats> will be executed as long as the <test> is obeyed.

Comment is enclosed in braces, and is not part of the algorithm. A description of an algorithm is a list of action specifiers enclosed in BEGIN and END.

Appendix C : Description of SBNF

SBNF as used in this report has the following rules:

{T}*	indicates repetition of T, zero or more times;
{T}+	indicates repetition of T, one or more times;
{T}? or [T]	indicate that T is optional;
<T>	makes T a non-terminal symbol;
{S/T}	is equivalent to: S{TS};