JOJO : an Interactive Domain and Terminal Generator

Kumeling, B.W.M.

Award date:
1984

UNIVERSITY OF TECHNOLOGY EINDHOVEN
DEPARTMENT OF ELECTRICAL ENGINEERING

Group Digital Systems

Graduation Report:

J O J O

An Interactive Domain and Terminal Generator

by

B.W.M. Kumeling

Coach: Ir. M. Stevens

Ir. C. Niessen (Natuurkundig Laboratorium N.V.Philips'Gloeilampenfabrieken

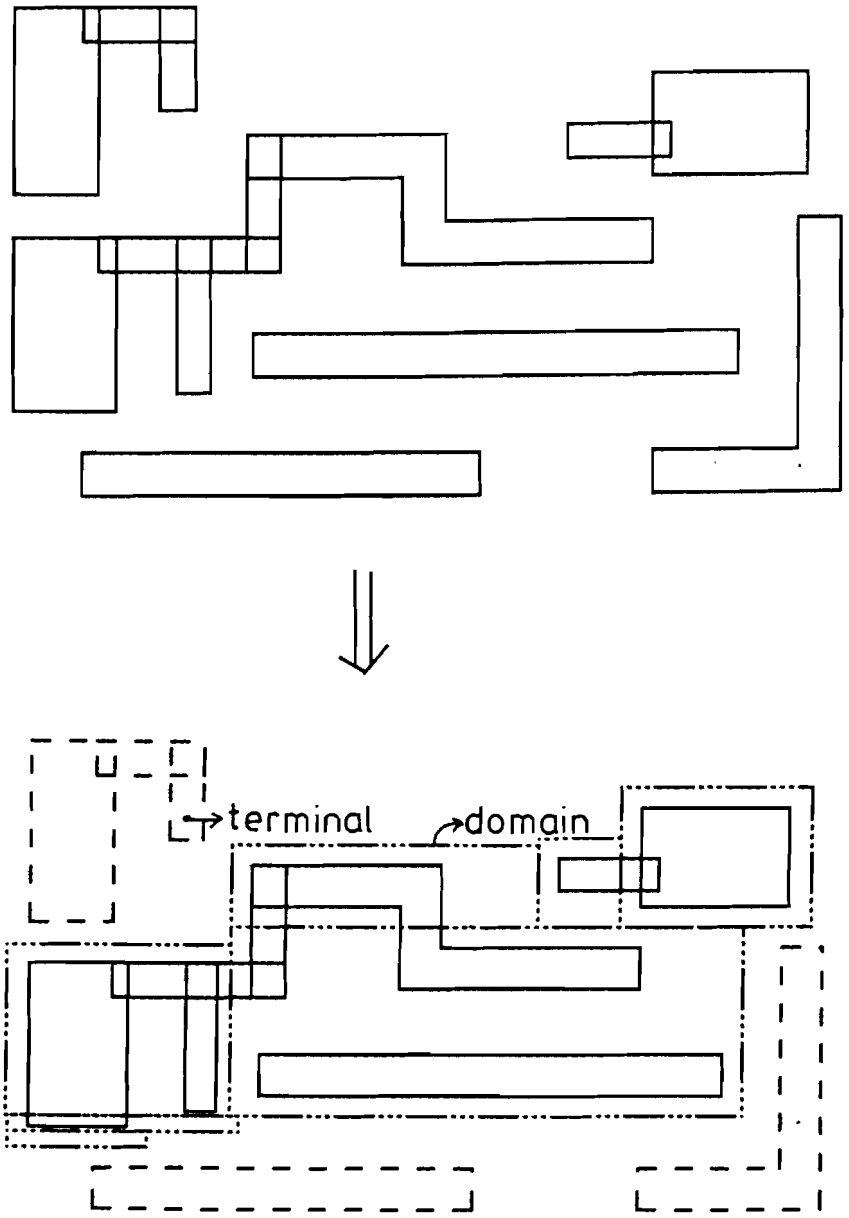
Period: september 1983 - augustus 1984

JOJO

An Interactive Domain and Terminal Generator

by

B. W. M. Kumeling



JOJO

An Interactive Domain and Terminal Generator

by

B. W. M. Kumeling

This report describes the work of the final project for the electrical-engineer degree of the University of Technology of Eindhoven, the Netherlands. The work was done in the period of september 1983 to july 1984 at the PHILIPS Research Laboratories. The work was done in the VOILA project, in which an interactive and hierarchical design-methodology is implemented. My task in the project was the investigation and implementation of methods, that can be used to make an abstraction of large amounts of layout-data. The abstraction is described as a simplified area, in VOILA called domain, and an interface, in VOILA called terminals. A study of hierarchies and existing methods of abstraction was made. An inventory of user requirements was made. An interactive approach was chosen to demonstrate the concepts of domain and terminal generation. A program was written in PASCAL on a VAX 11-780. It is embedded in the VOILA graphics editor.

Acknowledgement

The development of this report was greatly aided by C. Niessen. His careful reading and suggestions for revisions were thankful accepted. I also thank the members of the Nat. Lab. Voila-team: K. van Berkel, J. Engelen, K. Kuiper, B. Steinmueller, J. Walkier, and my TH-coach M. Stevens. Their patience in answering my questions greatly enhanced the progress of my work.

CHAPTER 1

INTRODUCTION

1.1 THE PROBLEM : 10 MILLION TRANSISTORS

VLSI technologies are so powerful that it is possible to integrate, in 1984, 1 million transistors on one single chip. For example the dense 1 Mbit dynamic RAMs, that were presented at the IEEE International Solid State Conference in february 1984 [ISC 84]. The design time of these memories is relatively short due to their extremely regular patterns.

The design time of complex signal- and micro-processors may be an order of magnitude larger. It is reported in [WEI 79] that the design time of the 16 bit microprocessor M68000 required 52 man-years for 68000 transistors. Lattin reports in [LAT 79] that the average productivity of a designer within a project, implementing random logic on a chip, is 10 transistors a day. A productivity of 6 transistors a day is reached in the design and layout of the M68000 project.

It is expected that an upper bound on the number of transistors on a VLSI chip will be 10 million. This means, that by an optimistic linear extrapolation, a team of 13 designers should have started a design near 1584; the year of the murder of Willem van Oranje!

The main problem with VLSI is the integrety of the design. Even the most talented and competent people make mistakes. The problem with errors is that they escalate in time; its recovery requires weeks or even months. A correction of an error may cause new errors elsewhere in the design. In

a complex design, more and more design-time is spent on the recovery of errors. Less time is available for the actual understanding of the inter-dependencies between the several parts of a system.

1.2 HOW COMPLEX IS COMPLEX?

P. Losleben gives in [LOS 80] an analogy of the increase in circuit complexity of 2 or 3 orders of magnitude. It is shown that, in the ultimate situation of 0.4 micron design rules, the area needed to draw the layout on a scale of 12500:1, will be equivalent to the size of 2 football fields! The idea of a team of designers, crawling about on the millimeter paper, indicates that different layout methods have to be found to master this complexity.

Why has this **VLSI complexity problem** and its management become so popular? Very complex systems are built routinely: examples are operating systems or airplanes; mankind is even capable of putting men (and women) on the moon! We have to make a historical review of the design of integrated circuits. LSI circuits were the result of a physical partitioning process: only a limited number of transistors could be placed on a single chip. The designer was forced to decide which sub-functions could be realized on a single chip. With VLSI technologies however, it is possible to integrate whole systems instead of sets of single transistors. The necessity for partitioning is no longer obligatory. The disappearance of off-chip drivers, the reduction of parasitic wiring capacitances and the decrease of the device-size has led to an enormous gain in performance. On the other hand, a complex and unstructured domain of silicon is created in which a designer has total freedom.

1.3 THE SOFTWARE CRISIS ANALOGY

There is a striking analogy with the development of large software systems in the 1960's. Mammoth systems were built, without knowledge of the applications. Requirements for the systems changed regularly. As a result, time-limits were exceeded and the costs were a multiple of what was planned. The performance of the first release was very poor. The problem was not the size of the program, but the programming discipline. The discipline of **Software Engineering** arose in the 1970's; it deals with the systematic approach of the development of large complex software systems. The accent is shifted away from the actual coding to the design of software. Structured methods are used to achieve complexity control and requirements are 'frozen' at a certain stage. Documentation and top-down implementation play an important role in software engineering.

There are many lessons, that can be learned from this Software crisis. C. Sequin explores in [SEQ 83] the nature of complexity in general and analyzes methods to deal with this complexity. In structured systems, modules have a strong internal cohesion, while the coupling between modules is as loosely as possible. In this way, complexity can be controlled; self-contained modules are easier to specify. These general techniques are also applicable to the design of complex VLSI systems. There are, however, a few extra problems:

* Instances of modules have to be placed in the 2-dimensional silicon plane. There is no procedure- or recursion mechanism that can be used.

* The number of pins to a chip is strictly limited. As an analogy, this would mean that a software module is only allowed to have two or three floating-point variables for input or output.

* Communication to other chips on a circuit board gives loss of bandwidth and requires extra power.

* A layout module has only a restricted number of neighbours. For software modules, the sequence and definition of procedures is of none concern.

Structuring in VLSI implies a good partitioning of the system. Systems should have a well-defined independent function with only a few exit points. They must consist of a limited number of sub-modules, that obey the same criteria. Clean and precise specification is necessary at all levels.

VLSI has a big advantage in comparison with software systems: VLSI systems can be designed for as much concurrent action as is possible.

Concurrency problems are not entirely solved; not even in software systems. Concurrency is a fast evolving research topic.

1.4 STRUCTURED DESIGN AND VLSI ENGINEERING

Hierarchical decomposition is the major tool in the design of complex systems [SIM 62]. A complex function is decomposed in less complex sub-functions. This process can be repeated several times until an elementary level of basic functions is reached.

Miller showed [MIL 56] that the number of elements on a hierarchical level should be no more than 7. This is due to the physiological fact that the short-term memory of a human being can hold 7 to 9 items at a time. This gives an indication of a good decomposition of a complex problem. There should not be less than 2 or 3 items on a hierarchical level: this results in a very deep and stiff hierarchy. More than 7 to 9 items lead to very complex inter-relationships and a flat hierarchy.

This decompositon process is often called the **top-down approach** of tackling the problem. The actual implementation occurs in the opposite direction: elementary functions are placed first and they are interconnected to make larger functions. This implementation process is also repeated several times until the complete function is built. This process is called **bottom-up implementation .**

Ideally, tasks such as: specification, simulation and test preparation are handled in a top-down manner. Tasks like partitioning, placement, routing and topological analysis are handled in a bottom-up manner. The top-down approach is called **functional design** and deals with design on a logical level. Bottom-up implementation is called **structural design.** Systems are actually built in a physical medium.

The problem with pure top-down design is that it is not known a priori how problems will be solved in sub functions. It is possible that similar problems arise in different sub-functions. In an early stage of design, this knowledge could have led to a different decomposition of the system. This means, that a good design travels several times up and down a hierarchical tree. With this insight, I decided to call my program : **JOJO**

The structured design is the natural evolution of an engineering discipline. Losleben writes:

> "The imposition of reduced freedom in the design process .... can no longer be avoided. ... most engineering disciplines involving the design of complex systems have seen a metamorphosis from a highly individualistic art form to a structured, highly disciplined, profession."

As an example : a space shuttle is made by a design-team and not by one single designer. The design-team works with well-defined interfaces and

uses a modular approach for solving the problem. In my opinion, VLSI engineering is also evolving to such profession.

1.5 HARDWARE DESIGN VS. VLSI LAYOUT

In hardware (printed circuit) design, systems are built with 'discrete' components. VLSI layout is an activity in which whole systems are integrated in silicon. These different approaches have different optimization criteria. In hardware design, a cost-function is used to minimize the number of gates (or active devices). This cost-function is very well suited for the design with standard MSI and LSI (TTL) components.

In **VLSI layout**, a different cost-function is used. As the number of components on a chip increases and the device-size decreases, the lenght and the number of interconnection wires will be the limiting factor; NOT the number of active devices! There are two main cost-functions in VLSI: **area** and **speed**. Long interconnect-wires result either in much power or slow operating speeds; both are not acceptable. The length of a wire determines how much time is needed to transfer data between two points. The most important problem is to find architectures, that need a minimal set of interconnect. When the design is finished, the most important optimization is the placement of modules in the layout, to achieve a minimal lenght of the wires.

Digital hardware design relies on **standardization** : a designer picks the exact pieces he needs from a catalog to implement a particular function. In VLSI, a library of standard functions can be used. The life-time of this library is very short; it has the same duration as a technological process (2-3 years). Technological changes, for instance, may lead to a complete re-implementation of the library (as happened in the change-over from NMOS to CMOS). Technological-independent library

functions however, loose efficiency in either speed or area. Parametrized functions, that can be adjusted to either area or speed, may give a solution. It is a powerful tool in combination with symbolic layout and compaction techniques [ELL 83].

In resemblance with hardware design, more and more irregular control functions are replaced by regular counterparts, as ROM or PLA.

1.6 THE DESIGN PROCESS OF AN INTEGRATED CIRCUIT

The design process of an integrated circuit is divided in several stages. In the first stage, a **functional specification** is made. This specification is a compromise between wild ideas and system demands. The behavior to the outside world and the internal operation are described in terms like: maximum power dissipation, minimum guaranteed access-time, number and assignment of pins of the package, kind and sequence of the operations, size of the operands, etc. etc. In this first stage, a decision has to be made in which technology the integrated circuit will be processed and which design method will be used. The first decomposition in smaller blocks is made and the functional specification is often accompanied by a, loosely drawn, **schematic diagram** of the system.

In the second stage, a transformation to a **logical description** is made. The large functions of the first stage are further decomposed in sub-functions. This process is repeated until the most elementary level of boolean gates or library functions is reached. Verification can be done either by **bread-boarding** (the actual implementation in hardware) or by computer simulation. The advantage of bread-boarding is the availability of a working prototype. It is however not applicable to VLSI circuits due to speed considerations: parasitic wire-capacitances cause reduced operating speeds. Furthermore, the documentation of changes in a

bread-board circuit demands a very severe (self-)discipline.

The most critical phase in the design of integrated circuits is the **layout-phase** : the drawing, digitizing and checking of the circuits in the different mask- or process layers. The layout phase is very costly: a huge investment in computer aids and in human interaction is required. Many methods are developed to help the designer with the drawing and checking of a layout. Generally spoken, a compromise is made between costs of production and costs of design. The most important criterion is the expected volume of production. For large production volumes it is desirable that the layout is as compact as possible. In this way, a maximum number of good chips on a wafer can be expected at a given process yield . An other criterion is the time to finish the design. As stated before, the design of complex signal- and microprocessors demands tens of man-years. Structured design methodologies are the key skills for future designs.

The last phase in the design of an integrated circuit is the production of the physical masks that are used in the wafer processing. The number of masks is dependent of the complexity of the technological process: this number can vary from 7, for a metal gate PMOS process, to 17 for an advanced CMOS process.

In summary: the design process of an integrated circuit can be split up in several phases:

idea -> functional specification -> network description -> layout -> masks -> IC,

in which the layout phase is the most critical one.

1.7 THE VOILA APPROACH

The structured-design approach is adopted in **VOILA** (acronym for VLSI Oriented Interactive Layout Aid). It creates an environment for the structured design of VLSI circuits and comprises the traject of:

network description --> layout --> mask data

In VOILA, the method of **hierarchical abstraction** is used as a way to control the complexity of a VLSI circuit. Niessen [NIE 83] defines abstraction as:

> "The method in which an object is replaced by a simplified one, that only defines the interactions of the object with its environment, while deleting the internal organization of the object. "

Hierarchical abstraction means that more than one level of abstraction is needed to achieve a reduction of the amount of data. The number of levels in a hierarchy is often called the **cardinality** of that hierarchy.

Hierarchical abstraction is the implementation of the **divide and conquer principle.** This well-known principle is applied in many human cultures; the organization of the Roman Warriors is a historical example. Division means that a problem is delegated to subproblems, that are independent of each other. Division implies formal methods for guaranteeing the locality of a module. Conquering means that the solutions of the subtasks are gathered by the next higher level in hierarchy. This implies, that there must be ways for modules to communicate with the 'outside world'.

VOILA does not assume a specific layout style. It has its main emphasis on interactive layout-design. Special care is given to an efficient

man-machine interface. Both top-down (floorplanning) and bottom-up layout design is possible. The modularity of the programs enables efficient applicaton-program development (as with JOJO). A layout editor is available in the VOILA tool-set.

The textual constructs for the description of layouts are formalized in a layout-language: **VLDL** , for VOILA Layout Description Language [BER 83]. To get an impression of the definition of VLDL and a description of an actual module, see appendices A and E.

All programs operate on one standardized datastructure: VIC (VOILA Intermediate Code). The VIC is a one-to-one projection on the memory of the computer of the textual constructs of the description of a module. This means that modules are described only once; instantiated modules are not expanded, as is the case in present-day layout programs. In this way, it is possible to store the whole data-structure of a VLSI chip in the memory of the computer. This enables fast search and display algorithms. Real-time editing is also possible. The standardized data-structure has an other advantage: no data-conversion is necessary for e.g. design-rule checking. Some design-automation tools are planned as well. Examples are cell-placement, channel-routing and PLA generation. The concept of VOILA, its features and its hardware requirements are described in a prospect [VOI 84].

The basic layout-unit in VOILA is the **module.** Both geometric and symbolic layout modules can be described in VOILA. Geometric modules are called **patterns** . Their geometric primitives consist of line-segments that have angles of multiples of 45 degrees, relative to x- or y-axis. Symbolic modules are called **cells** . On a symbolic level, only placements of subcells and their interconnect is relevant. Shapes are restricted to 90 degrees. Symbolic layout enables a considerable degree of abstraction from the technological details. A limited set of masks is used; it contains

e.g. no implantation masks. A symbol can hide a very complex subcell. On the other hand, symbolic layout gives a reduced degree in layout-flexibility.

In VOILA, the conquering principle is formalized in the **terminals** of a module. Terminals define all possible ways of communication with the environment. The division principle is implemented in the **domains** of a module. Domains claim layout-areas in the different process masks and shield the contents of a module from its environment. An other abstraction mechanism in VOILA are the **drawings** . Drawings allow the user to place graphical comment in the layout. They are described in color-masks, that will not be converted to process-masks.

Domains, terminals and drawings are the **abstraction of a module.** They are the means for hierarchical abstraction. Terminals and contents data are the **realization of a module.** Layout-elements of both types will appear on silicon.

The abstraction of a module has its greatest benefit in a hierarchical description of a system. When a module is designed, tested, and proven to be correct, only the abstraction of the modules is needed in the next higher level of hierarchy. In this way, large systems can be built with a minimum amount of detail to be considered at any time.

1.8 THE LINK OF DOMAIN AND TERMINAL GENERATION WITH VOILA

In VOILA, the description of the abstraction of a module requires an extra amount of work by the designer. In present-day layout-design, the designer has to define the contents (or realization) of a module. The question arose if it was possible in VOILA, to derive this abstraction data from already existing layout- and network data. Domains and terminals are in fact hidden in the contents of a module. This observation hits the

heart of the matter. A re-definition of my work is:

> Make a description of the domains and terminals of a module from already existing layout and network-data. The abstraction should be as simple as possible and should give as little as possible loss in flexibility.

This report gives an overview of some abstraction mechanisms. The properties of the VOILA domains, terminals and drawings are high-lighted and further explored from the syntax of VLDL. These properties are the boundary conditions for this work; they are the semantics of VLDL.

An interactive language is introduced for the generation of domains and terminals and some implementation aspects will be treated. The algorithms are embedded in the standard VOILA editor. They are written in PASCAL [WEL 82] on a VAX 11-780. Finally, some conclusions are drawn and directions for future investigations will be given.

CHAPTER 2

ABSTRACTION AND HIERARCHY CONCEPTS

2.1 PRESENT-DAY ABSTRACTION

Present-day CAD for layout-design already uses hierarchical layout-description languages (e.g. Circuitmask (CMSK) at Philips). Geometrical patterns can be nested in geometrical patterns. So what is the problem?

CMSK programs make no use of this hierarchical description, because they do not work with abstraction. CMSK expands all layout data up to the highest level in hierarchy. This is the reason that mask manipulation programs and design-rule checkers are the fastest growing group of consumers of available processing power. The next sections contain some case-studies of present-day ways of making abstractions.

2.1.1 Abstraction By Using Macros

In the so called Nat.Lab. LSI shop, synchronous NMOS circuits are developed with scannable delay flip-flops. At the input of each flip-flop, a logical and-to-or function is connected. Data is latched in a flip-flop at one edge of the global clock. Output data is available at the other edge of the clock. The flip-flops are placed row-wise to enable connection of power supplies etc. by abutment. The logical and-to-or function is placed in a wiring channel underneath that row of flip-flops. Fig 2.1

shows how a symbolic layout looks like. Wiring is placed on a coarse grid and is described in symbolic wires.

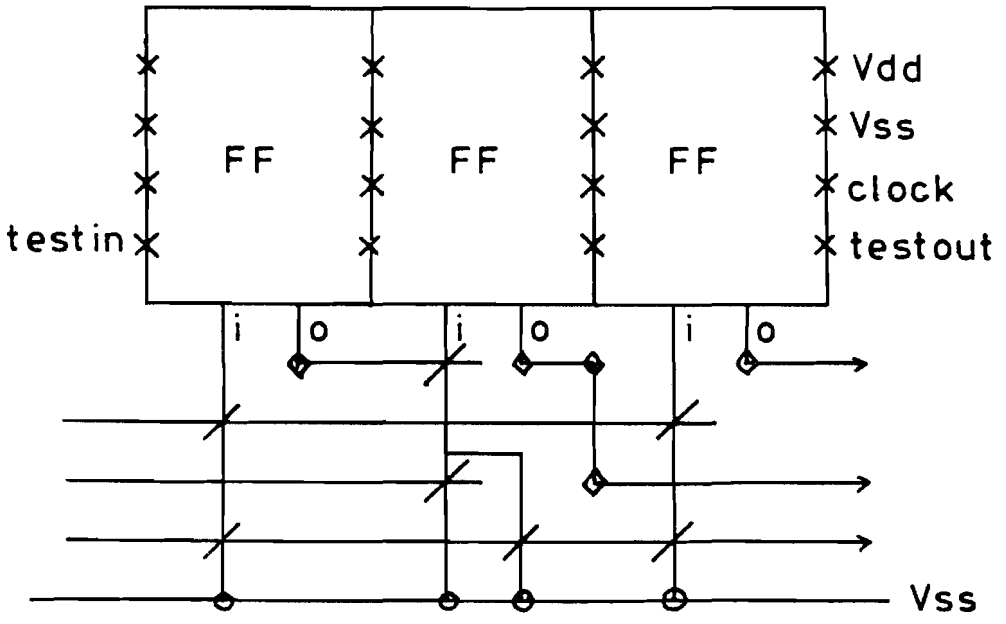


Fig. 2.1. Example of a layout in the LSI-shop.

In a testmode, the logic and-to-or circuit is disconnected and the input is connected to the adjacent output. All flip-flops are chained in this way to enable scanpath testing.

For geometrical checking, the following abstraction is applied: dummy cells are created manually to replace (checked) flip-flops. Only the peripheral of the dummy cell is identical with the flip-flop. Its content is less detailed.

A number of flip-flops (say 20) represent a subfunction which is fully checked with all detailed layout-rule checks. However, the detailed flip-flop was replaced by a dummy cell. Also for this subfunction, a dummy cell is created. This dummy cell contains only the area, occupied by this subfunction and connection-points on the edge of this area. The occupied area lies one gridpoint around the actual details and should be as rectangular as possible. The connection-points are situated in this grown area and they are represented by small rectangles. The area and the

connection-points are placed by hand (Fig. 2.2).

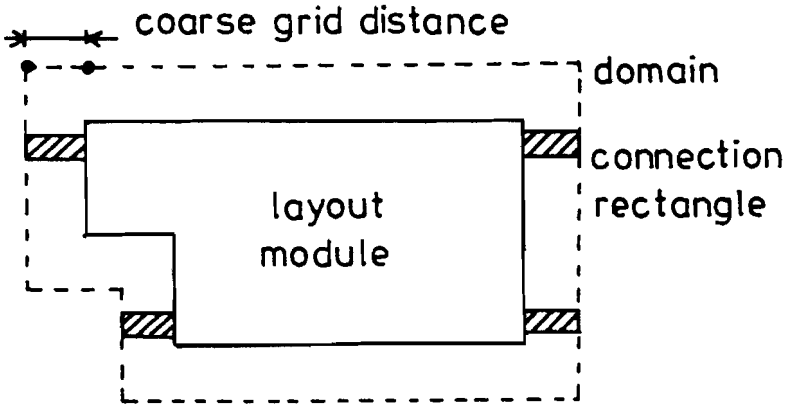


Fig. 2.2. Macro description of a module in the LSI-shop.

In the next step, only the 'grown' area is taken; the inner part in fig. 2.2 is empty. Ensuring correctness of the dummy cells is a designer's responsibility for which no CAD checks are available. The conventional design-rule checker only checks environment-details against the small connection-rectangles of the dummy pattern. Intrusion of wires in the occupied area is inspected on a plot of the layout, by eye.

In this way, a reduction of CPU-hours is achieved. At mask-generation time however, no more gain is achieved. The dummy patterns are replaced by the original ones and CMSK expands all the layout data. All problems are in fact shifted to the end of the design traject! This macro design method demands a very large self-discipline of the designer. It was used until august 1983. In current designs, which also contains CMOS, the program DELILA is used.

2.1.2 **Abstraction With Respect To Layout-verification.**

In the Nat. Lab. research group Digital Circuitry and Memories, a CAD tool-set has been developed to assist a designer in hierarchical verification of CMOS layouts. It also helps the designer in placing and checking of interconnection fields. With the help of a layout-extraction

program, a net description is made out of an interconnection field and a placement of submodules, in the description language of SIMON. The terminals in the layout have a one-to-one relationship with the inputs and outputs of a networkdescription of the same module, and of power-supply lines. The areas of submodules are recognised by the program with the help of a mask ZD, that is not projected on silicon. It covers most of the details of the module, except for a, on purpose, selected set of external areas. The external areas are used to generate the terminals of the module in an automatic way. External areas in the interconnection-masks IN, PS and OD are grown by a small factor and 'anded' with the domain description in ZD. With this overlap area, a terminal is made in the helpmasks ZI, ZP and ZO (Fig. 2.3).

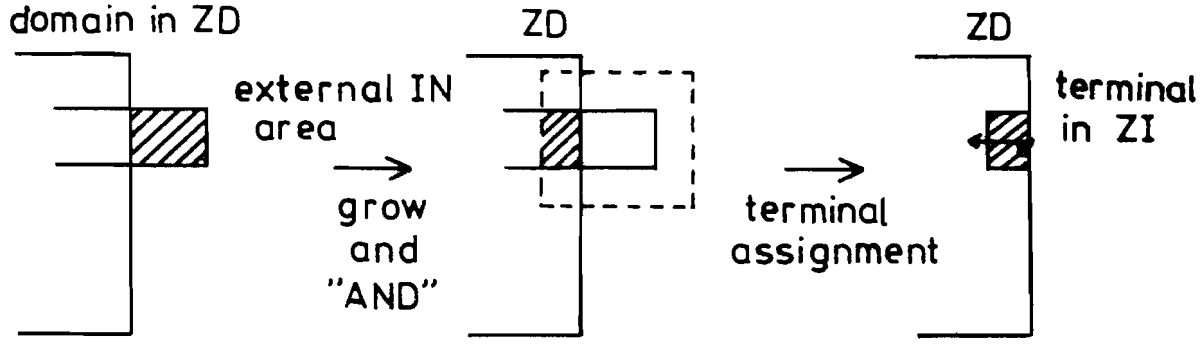


Fig. 2.3. Generation of terminal-points in ZI, ZP or ZO from domain (ZD) and overlap areas in Termdef.

The next step is the assignment of a name to these terminals. In an interactive session with the designer, a cursor is automatically adjusted on top of the generated terminals. The designer is asked to assign a name to that terminal, that is selected from the existing network-description. Terminals can be defined of type input, output or power. Input and output terminals are read from the i/o list of the network description. Power-supply terminals can also be named and declared of type power.

The placement of the domain area in mask ZD is, in this approach, seen as an intelligent action. Terminal generation in masks ZI, ZP and ZO is

done in an automatic way. Name assignment to terminals is also straightforward: only names from a network-description and power-supply lines have to be selected. An additional feature of the program is the placement of information in the plot of the module. This gives better readable plots and serves as a documentation aid.

2.1.3 **Abstraction In DELILA**

The program Delila is a CAD tool for the generation of symbolic layouts and is described in [STR 81]. The basic design module in Delila is the **UNIT.** Within a unit, free layout is possible and is checked with detailed layout-rule checks. The connection points to the unit have to be placed on coarse grid. A symbolic representation of the unit is made when the layout is finished. A chip is composed of (up to 200) units, interconnected by wires in maximal 3 connection layers. Wires, vias and active devices are represented as stick-diagrams on a coarse grid. Fig 2.4 shows a layout of a unit.

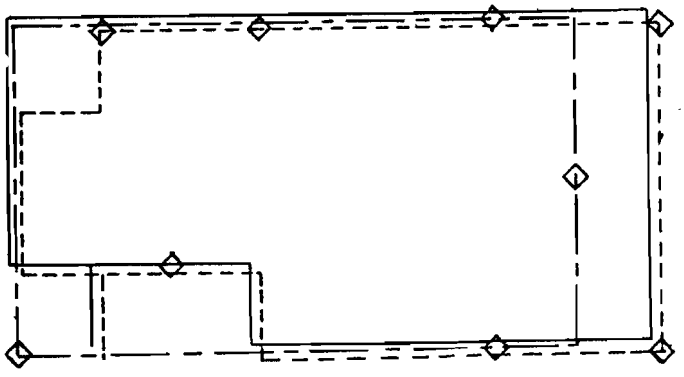


Fig. 2.4. Example of a Delila UNIT.

Both NMOS (interlace dynamically-latched logic) and Bipolar (IIL) layouts can be made with this CAD tool. A large library of standard functions is available.

The abstraction is described by so-called **protective outlines.** These are fences around a unit in the 3 connection layers, and in a special mask

XG. Connection to a unit is made to **contact-points** on the edge of a protective outline. In Delila, it is essential that every design unit has a network-description in SIMON and a CMSK layout-description of the same name. Units are not allowed to overlap. There must always be one grid distance between two adjacent cells. This is a simplified check compared to the check of all layout-details. Global wires can be placed by using through-routes or throuts. Contacts are placed on both sides of a unit. They may be internally interconnected when no protective outline between these 2 points is specified. The interlace logic is placed on coarse grid and is (logically) checked.

An interesting utility in Delila is the automatic generation of **SUPER-UNITS** . Its protective outlines are generated from the outlines of sub-units, the interconnect, the active devices and the vias. The information for the contacts to a super-unit is derived from the network-description. The algorithm for outline generation is straightforward. An inventory of occupied grid-points is made, followed by some operations. A, built-in, parameter is used to smooth the edges of the protective outline by searching indentations. E.g., the indentation area between 2 grid-points is assigned to the outline area. An other control parameter is the number of edges, that may appear in the outline description.

2.1.4 Some Observations

From this review of abstraction mechanisms in existing CAD tools, we can state that there are two general principles used in abstraction:

- The first one is a fence around the module, that may not be crossed by elements from the environment. In literature, many exotic names are given to this fence:

domain	Voila and Termdef
protective outline	Delila
bounding box	hierarchical design-rule checker [WHI 81]
protective frame	Berkeley [ELL 83]
shadow	MDS (IBM) [WOL 81]
bounded cell representation	HP [TUC 82]

- The second abstraction principle is the existance of openings in the fences. They enable the module to communicate with the 'outside world'. An other anthology:

terminal	Voila and Termdef
port	REST [MOS 81]
connection point	Mulga [WES 81] and at HP [TUC 82]
contacts	Delila
pins	

2.2 AN OTHER HIERARCHICAL CONCEPT

Rowson [ROW 80] reports a different approach for the description of hierarchies at Caltech University. A **separated hierarchy** is used, that consists of leaf-cells and composition-cells . Leaf-cells are the 'atomic' units, that contain real mask-data; they contain no copies of other leaf-cells. Composition-cells only contain names of leaf-cells and other composition-cells. The composition-cells contain no mask-data for the

interconnection of subcells. Instead, they contain an algorithmic description for the interconnection, that is called a composition rule. Rowson shows that hierarchies (and thus composition rules) can be described mathematically by using Lambda calculus. Their philosophy is to make VLSI chips by construction and not by afterwards-testing (as is the case in many American Industries).

2.3 EQUIVALENCES BETWEEN HIERARCHIES

When two persons are decomposing a complex problem, it is very unlikely that they will produce the same hierarchies. When layout and network are decomposed in different hierarchies, consistency checking can only be done when the two hierarchies are equivalent. Rowson distinguishes three types of equivalences between hierarchies: identical, functional and topological.

> Identical: "Two systems are equivalent only if every cell in every level of the hierarchical tree composes the same instances of the same cells with the same interconnection."

This is the strictest definition and is applied in VOILA. The hierarchical description of the layout-modules must be the same as the hierarchical description of the network. This is a necessary condition for incremental connectivity-checking.

> Functional: "Two systems are equivalent if, given the same input sequences, they produce identical output sequences."

This definition is too unrestrictive. As an example, a CPU can be implemented on silicon in two ways. The first one is the result of a functional decomposition. The CPU is subdivided in ALU, MMU, Registers

etc. and these functions are mutually interconnected by busses. The second implementation is the result of a decomposition per bit. A bit-slice is made in which the functions of the ALU etc. are distribed to each bit. Slices are connected by abutment. Both hierarchies are fundamentally different, but not according to the second definition.

Topological: "Two hierarchies are equivalent if the hierarchies that define them result in identical compositions of the identical leaves."

This means that the systems can be compared with each other, when the two hierarchies are "flattened-out"; i.e. described on one level with no hierarchy at all. The "flattening-out" process is implemented e.g. in layout-extraction programs.

CHAPTER 3

SEMANTICS OF THE VOILA ABSTRACTION

This chapter gives an overview of the properties of the VOILA abstraction: domains, terminals and drawings. The syntax definition of VLDL, that is necessary for the definition of the abstraction of a module, is drawn in graphs in appendix A. In this chapter, the VLDL syntax is interpreted and further explored. Some applications of the use of abstraction will be given.

3.1 PROPERTIES OF DOMAINS

The function of domains can be summarized as:

"Domains claim an area in the layout".

Domains can be defined in any process-mask with any geometric primitive. The set of areas in all these masks defines a 3-dimensional surface, in which the local subtask of the module is executed. Within each mask, a domain is described as a 2-dimensional area (Fig. 3.1). The domain area, in e.g. the PS mask, is called: "Domain in PS".

Domains of 2 separate modules may touch, but NOT overlap.

The use of domains requires 2 extra sets of design rules. They are called **modular design-rules** and can be derived from one set of technological rules. As an example, the modular design-rules for a

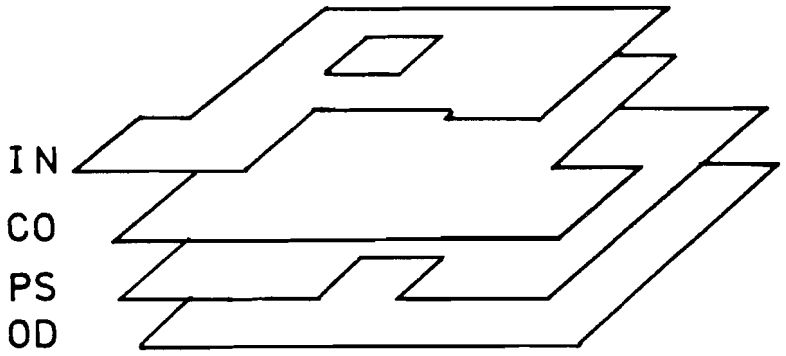


Fig. 3.1. A 3-dimensional representation of a domain in the several masks of an NMOS process.

clearance of 4 um between 2 IN elements are (Fig. 3.2):

1. IN detail to IN detail = 4 um.

2. IN detail to IN domain = 2 um.

3. IN domain to In domain = 0 um.

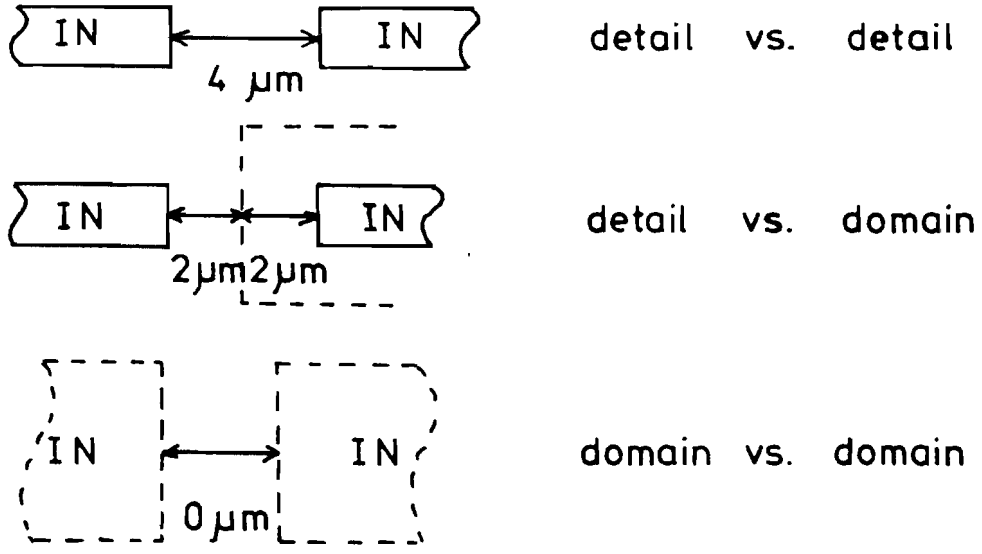


Fig. 3.2. A modular set of design-rules can be derived from one technological design-rule.

When the domain border is chosen as half the minimum clearance-distance, the minimum clearance rules are always obeyed. (Fig. 3.3).

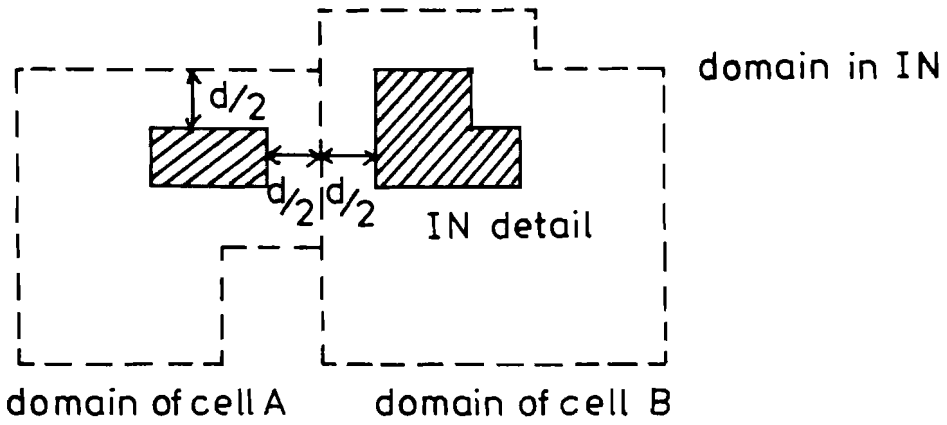


Fig. 3.3. Touching domains guarantee minimum clearance-distances (d) between two layout-elements.

Domains serve as an abstraction mechanism: they are never printed on silicon. It is even possible to define no domain at all for a module. The technological structures on the chip will remain the same, with or without a domain. The contents data of such module is no longer protected and is considered as global or common to the environment. The efficiency of the hierarchical verification-tools will decrease (and the necessary CPU-hours increase) due to expansion of all instances of the module.

Domains ensure that no elements from the environment intrude in the module. There is, however, one exception to this rule. A layout element is allowed to cross a domain border when a terminal lies near this border (Fig. 3.4).

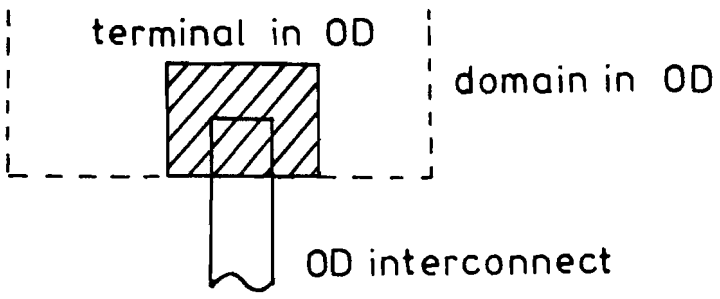


Fig. 3.4. Interconnection wires are allowed to cross a domain boundary at the place of a terminal.

Every domain border must have a Manhattan (or orthogonal) boundary. This eases the transformation of a geometrical pattern to a symbolical cell. The size of the domain area remains the same for both descriptions. Slanted boundaries in the contents part must be made orthogonal.

Domains may be non-connected. This means that unused areas can be filled by the environment. More specifically, this area can be used by the next higher module in hierarchy for interconnection purposes. Vias and wires can be placed in this area (Fig. 3.5), and this leads to dense packed circuits.

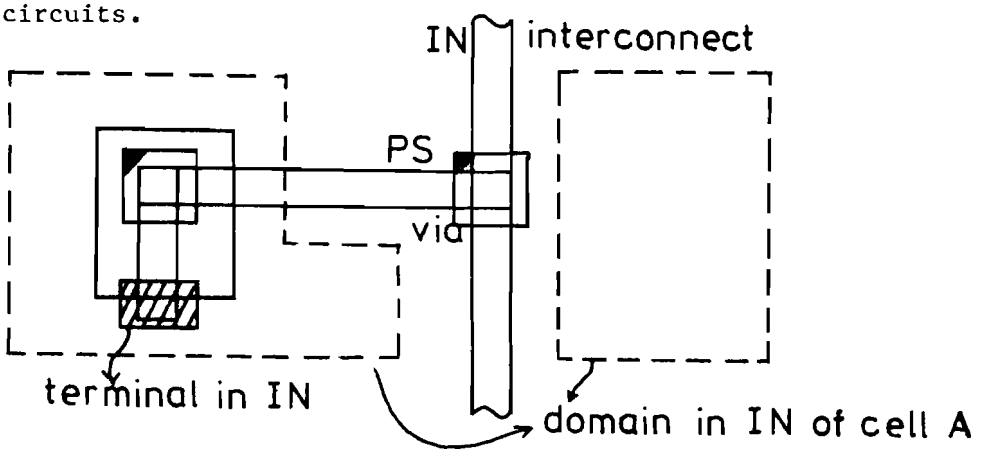


Fig. 3.5. Non-connected domain-areas allow ease of interconnect.

There are 2 extremes in the definition of a domain:

1. The most simple domain is a **bounding box.** This is a rectangle, that encloses all layout elements with a distance of 0.5*d. (d is the minimal, technological, spacing between 2 elements). A maximum level of abstraction is reached, resulting in a huge data-reduction and high efficiency of checking tools. The price to be paid is a considerable loss of layout area.

2. The most complex domain is built from the set of all layout details of a module in the following way. All details are approximated by Manhattan counterparts and expanded (grown) with a factor 0.5*d. This domain description gives no loss of silicon area, but no data-reduction is achieved. Data is in fact copied to another place. The price to be paid this time is a decrease of efficiency of CAD-tools and an enormous claim on computer memory.

Both alternatives are not acceptable; a compromise is somewhere in the middle and is the subject of this work.

3.2 PROPERTIES OF TERMINALS

The function of terminals can be summarized as:

"Terminals give all possibilities for a module to communicate with its environment."

Parasitic couplings are not taken into account. In contrast with domains, terminals are a part of the realization of a module and will be printed on silicon. This means that they don't have to be redeclared in the contents part of a module. They can be defined in any mask and with any geometric primitive. Terminals in VOILA are restricted to have Manhattan boundaries. Terminals can be regarded as 2 or 3-dimensional areas, that are shared by the module and its environment.

Terminals of a module can have their counterparts in the inputs and outputs of a network-description or in global power-supply lines. The location of terminals may be anywhere in the layout, relative to the domain border. This means that they are even allowed to cross that border. A useful terminal however, must be connectable from its environment. It is connectable if it is located on less than 0.5*d distance within a domain

area (Fig. 3.6).

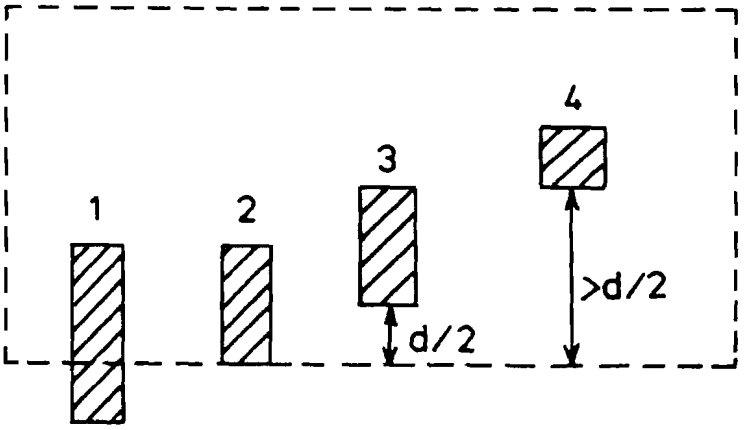


Fig. 3.6. Terminals may be defined anywhere, relative to the domain boundary.

This observation enables a large freedom in the definition of domains and terminals. Terminals can be defined before or after domains. The shapes of domains however, will be different in both cases. Domains protect the contents data of the module; not the terminals. Fig. 3.7 gives some examples.

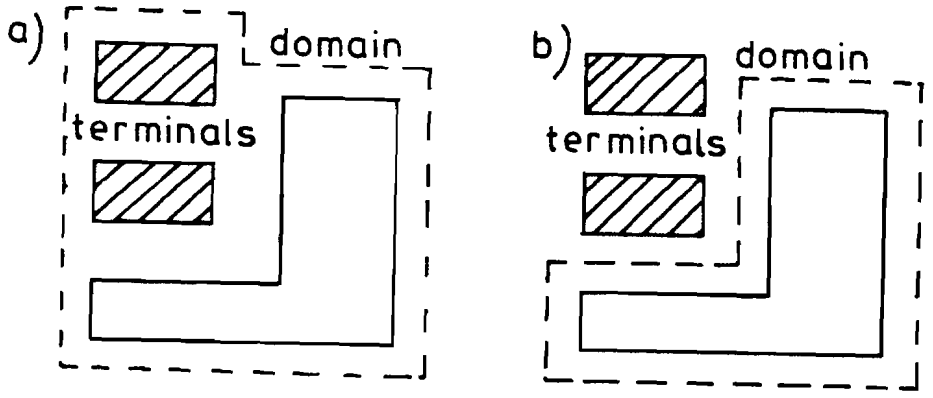


Fig. 3.7. Domain generation before terminal generation (a) leads to different domain shapes than domain generation after terminal generation (b).

When a terminal is fully surrounded by a domain area, and the distance to the domain boundary is greater than 0.5*d, it is not connectable in that mask. As an example, Fig. 3.8 shows that such terminal can only be

connected via the third dimension. The PS terminal makes contact with a metal wire via a contact-hole.

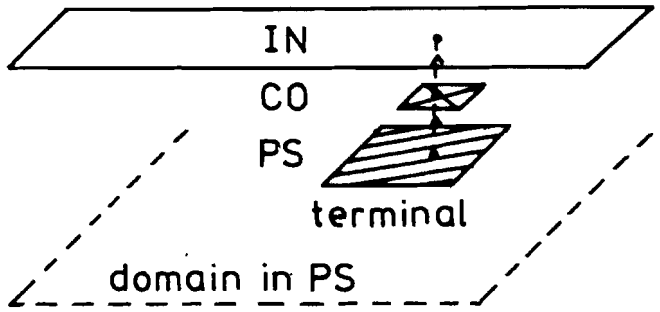


Fig. 3.8. The terminal in PS can only be connected via the third dimension (via CO and IN).

The terminal part of a module may be empty. In this way, the contents of a module can be fully isolated from the environment. Alignment and identification patterns are examples of modules without terminals. When no domains and no terminals are defined, all data of a module is considered as global.

Two applications can be given for terminals.

1. Terminals are means for the connection of a module

2. Terminals can be used as a geometrical coupling of common areas. Examples are the buried layer in IIL, that is used as ground, and the p- or n-well in CMOS; they must be connected to ground or VDD. In these applications, the use of terminals leads to dense packed circuits. Fig. 3.9 gives an example of conventional connection and connection with terminals.

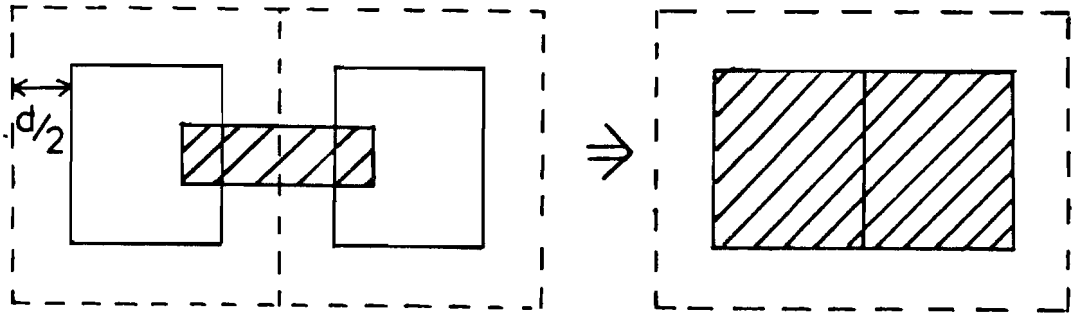


Fig. 3.9. Terminals, used as a geometric coupling, lead to dense packed circuits.

Two extremes can be distinguished in the definition of terminals.

1. The most simple terminal is a contact-point on the edge of a domain border.

2. The most complex terminal is the set of layout-elements, that belongs to one equipotential net. For deriving such terminal, knowledge of the connectivity of the technological process is indispensable.

Terminals should be as simple as possible to achieve a maximum of abstraction.

Terminals, connected to the same net are called electrical equivalent. They can be grouped in a **signal.** Terminals in a signal may be interchanged in the layout. The designer has the responsibility for the mutual interconnection of the terminals within a module. Signal terminals are called subcell-connected. Single terminals of the same name must be interconnected in the environment of the module; they are supercell-connected (Fig. 3.10).

```
signal
  dot( ps; x1, y1);
  dot( ps; x2, y2)
end : input;

 terminals are
 subcell-connected
```

```
dot( ps; x1, y1) : input;
dot( ps; x2, y2) : input;

terminals are
supercell-connected
```

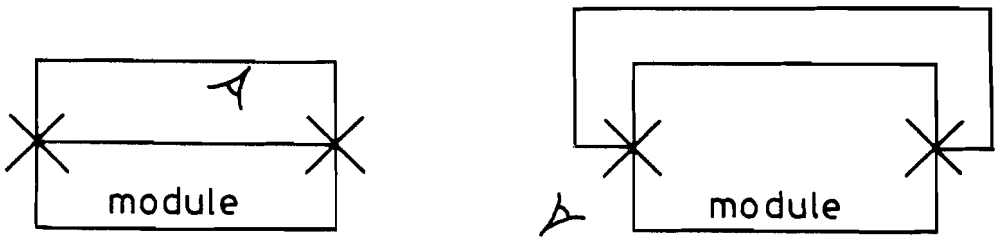


Fig. 3.10. Terminals can be subcell or supercell connected.

The freedom of making contact to a terminal can be restricted by assigning a class to a certain terminal. It can be compared with the TYPE mechanism of a variable in Pascal. Terminals of the same class may only be connected to each other. As an example, a terminal of class VDD may not be connected to a terminal of class VSS. A connectivity-checker can detect this catastrophic short-circuit long before a simulation of the circuit, that is fed with data from a layout-extraction program.

3.3 PROPERTIES OF DRAWINGS

Drawings present a module in a graphical way. Layout-primitives are described in color-masks, that will not be printed on silicon. Drawings allow the designer to place graphical comment in the layout; e.g. the names of interconnect wires. The identification of a module can also take place in the drawing part. What kind of information is necessary?

1. Information about the function of a module. The name of the module should appear in the drawing part.

2. Information about the placement of a module in the layout. The designer should have mirroring and rotation information at a first glance. A non-symmetrical symbol can be used for this purpose.

3. Information about the abstraction of a module. Domain and terminal data can be usedas graphical comment or careful selected symbols can be defined. E. g., in bipolar layouts, it is tradition to indicate a collector with a plus, an emitter with a small rectangle and the base with a cross.

In summary, the properties of the abstraction of modules in VOILA are the boundary conditions for the generation of domains, terminals and drawings of a module.

CHAPTER 4

SOME THOUGHTS ON ABSTRACTION

This chapter gives some indications of how domains and terminals can be generated. It is shown that the knowledge of the designer is indispensable. This leads to an interactive approach of assigning domains and terminals to a module. Finally, some user requirements are drafted.

4.1 THE SHAPE OF DOMAINS

Three criteria can be distinguished in the assignment of a domain to a module.

The first criterion is based on economical reasons. The shape of a domain area is dependent of the expected volume of production: large production-volumes require a dense packed chip-area. The domains of all modules must be placed tight around the modules. The shape of a domain is also dependent of the speed of introduction of an IC to the market. Fast market-introduction implies a short design-time. This means that floorplanning techniques must be used, in which the, rectangular, shape of domains is determined beforehand.

The second criterion is based on the number of layout-elements in a module. The shape of the domains of complex modules must be more rectangular than those of primitive modules, to achieve any data-reduction. The place of the module in the layout is also of interest. Modules, placed

in a regular matrix must have a more tight domain. Modules on the edge of a chip can have a more rectangular domain.

The third criterion is the the design-style (gate-array, standard-cell etc..) that is used. Standard-cell design e.g., requires a uniform height of the layout-modules. The domain shape must be rectangular.

In general, the description of the abstraction of a module should be as simple as possible. Holes and indentations in the domain description are not forbidden, but should be avoided. A good measure of the complexity of a domain is the number of edges of the domain area.

4.2 DOMAINS OF COMPLEX MODULES

For complex modules, manual generation of domain boundaries is a tedious job. A domain boundary can also be generated from existing mask-details. Layout-elements are 'grown' in size, merged and 'shrunk' back. The grow and merge operation have the effect that holes in the domain description are filled. The shrink operation takes care for a tight boundary around mask-details, at 0.5*d distance. A large value of the grow factor has the effect that a boundary polygon is generated. A small value of the grow factor will result in a more complex description. It will give the designer more flexibility in using unused areas in a mask.

Automatic generation of domains has a draw-back: it is possible that some terminals cannot be connected anymore in the same mask. The algorithm should only be applied in an interactive environment, in which the designer can choose an optimal shape of the domain.

4.3 GEOMETRIC AND SYMBOLIC DOMAINS

A domain is, according to its definition, an area, that is occupied by a layout-module and guarantees the locality of a module. This is the definition of a **geometric domain** and is a minimal definition. In the layout, there exist areas that may not be occupied by elements in a certain mask. Fig. 4.1 shows the situation, in which the OD-area is **forbidden area** for PS interconnect.

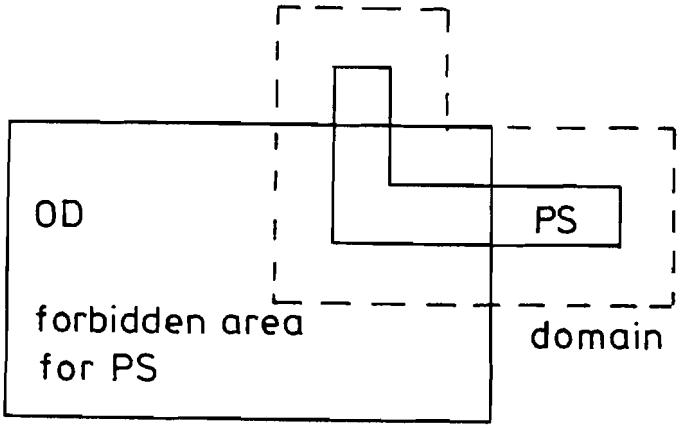


Fig. 4.1. A domain in PS should also contain forbidden areas for interconnect in PS.

PS details would disturb the structure within the module by creating a parasitic capacitor or transistor. A design-rule checker will give us a warning, but it would be convenient if this forbidden OD-area was visible in the PS domain. In this way, the designer only has to pay attention to PS interconnect and PS domains. Knowledge of possible interconnection areas can be achieved without knowledge of details in other mask-layers. The chance of making errors can be reduced. The domain area, that contains also forbidden areas of related masks, is called **symbolic domain.** In my restricted time, I have chosen to implement only the geometric domain generation.

4.3.1 Estimation Of Forbidden Areas

The forbidden areas can be derived from a **design-rule matrix.** The diagonal elements in this matrix indicate the minimum clearance between two elements in one mask-layer. The existance of an off-diagonal position indicates that a relationship exists between two elements in different mask-layers (Fig. 4.2.).

	OD	PS	IN
OD	*	*	-
PS	*	*	-
IN	-	-	*

Fig. 4.2. Example of a design rule matrix

The off-diagonal positions indicate forbidden areas for both masks. For example, a relation between OD and PS means that OD is forbidden area for PS interconnect, and PS is forbidden area for OD interconnect. The following algorithm can be used for the determination of a symbolic domain in e.g. the PS mask.

1. Make a geometric domain as mentioned in the previous chapters. The domain boundary lies at 0.5* distance(PS-PS) from the PS details.

2. As can be derived from Fig. 4.2, the PS mask has a relation to the OD mask. Generate a geometric domain of the OD mask with a domain boundary at 0.5* distance(OD-PS) from the OD details.

3. Merge the two areas. The result is a symbolic domain in PS, that covers all PS details plus the forbidden areas for PS.

4.4 DOMAINS OF CONTACT-HOLE AND INVERSELY PROGRAMMED MASKS

To ensure the locality of a module, all layout details should be covered with a domain area. Contact-holes are placed very sparse in the layout. A bounding box description as a domain, can give too much loss of layout area. A small value of the grow factor gives us a grown copy of all details. We have not achieved any data-reduction. The best alternative is to apply the grow/merge/shrink operation with a very large grow factor. This results in a bounding polygon description as domain for the contact-hole masks.

Domains of contact-hole and implantation masks can be grouped in a mask-list, to achieve more data-reduction. It contains, in fact, the logical OR of the domains of non-metallization masks.

Both in MOS and in Bipolar processes, it is common use that masks are used to define areas, where no implantation or diffusion may take place. Fig. 4.3 shows an example of a polysilicon area, in which a low-doped resistor area is connected with two low-ohmic contact-heads.

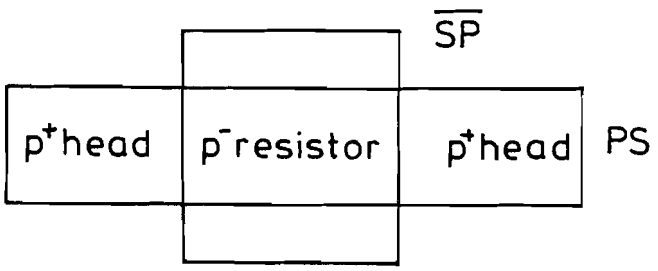


Fig. 4.3. Definition of a p-type resistor in polysilicon with a normal and an inverse programmed mask.

It can also happen that the, inversely programmed, masks are used to reduce the amount of layout-details. At mask-generation time, a negative exposed process-mask can be obtained. These two inversions give us the mask, that was originally intended.

The domains of inversely programmed masks are treated in the same way as

normally programmed masks. The absence of an implantation step determines the internal structure of a module and must hence be kept local to the module!

4.5 AUTOMATIC VS. INTERACTIVE TERMINAL GENERATION

As stated before, terminals are hidden in the contents part of a module. If the positions of the terminals have to be generated in an automatic way, knowledge of the internal structure of a module is necessary. The structure of a module can be captured in a network graph. Two graphs are needed; one is derived from the network description, and another is derived from the actual layout of the circuit. The graph from the layout is derived from a hierarchical description of the layout-module. For primitive patterns, the graph is constructed from a transistor description of the module, that is made by a layout-extraction program. A comparison of both network topologies is necessary to search for possible terminal locations. Topologies, derived from layout and network, can differ; one or more topology-transformations may be needed. Some class of circuits can only be compared by computer simulation. Examples are circuits with analog tricks: driver-circuits in NMOS are described as inverting (or non-inverting) gates. The actual implementation in the layout can contain up to 10 transistors.

Suppose that the topologies are the same. The question arises which part of the equipotential net should be declared as terminal? This problem is very complex and can only be solved in some cases with rigid assumptions. Terminals could, for instance, be chosen as contact-points on the edge of a domain boundary. This means that a subset of a net is declared as terminal. For primitive patterns, small rectangles can be used, as in the LSI-shop. Terminals are often used to share busses and

global power-supply lines. Lines, running across a domain boundary, from one end to another can be used. In standard cell design, with a uniform height of the modules, it is known in advance where connection is made to a module. Contact to another module is made by abutment. This knowledge can only be applied in this special case, but no general statements can be derived from it.

It is clear that the knowledge of a designer is necessary. He has, in fact, the whole connectivity of the module in his mind. He knows exactly the positions, where a module will be connected. In contrast to a designer, a computer program only knows the interior of a module and is forced to use brute computing-power to get insight in its structure. Terminal generation should be done in an interactive way; the computer program asks the designer the right data at the right moment. The designer however, holds full responsibility for making the terminals. The program can only give warnings and directions.

4.6 STANDARD CONTACT PATTERNS (OR VIAS)

A commonly used layout-practice is the use of standard contact-patterns (or via cells). Within such pattern, all layout-details satisfy the design-rules. This has to be checked only once. All instanciated patterns are correct. In VOILA, a connection to a terminal can also be made with a via module. This is a special module, that contains no contents part. All layout-data is placed in the terminal-part and is, according to the definition, part of the realization of a module. It is not advisable to specify a domain for a via module. The domain of the via module and the domain of the module to be connected are not allowed to overlap. Furthermore, this would prohibit domain generation before terminal generation. A disadvantage could be that no slanted boundaries may appear

in the terminal part.

4.7 THE USER REQUIREMENTS

As can be summarized, both domains and terminals must be made in an interactive way. The designer knows the internal structure of a module and knows how the module will be connected to the 'outside-world'. This knowledge is very difficult to formalize and would require a large overhead. At all stages, the user must have full control over the generation process. The computer program can only give directions and propositions. The propositions can be changed by giving extra information.

When the user asks for a tight domain, all areas, that can be used for interconnection purposes may not be joined to the domain area. On the other extreme, it must also be possible to make a bounding box in a certain mask.

In case of terminal generation, existing layout-elements from the contents part must be selected. These elements must be named and declared as terminal. It must also be possible that terminals can be generated without the existance of a network description. The user has the responsibility for generating domains and terminals.

CHAPTER 5

SPECIFICATION OF DOMAIN AND TERMINAL GENERATION

This chapter gives an overview of the environment, in which JOJO is embedded. The interfaces to both VOILA programs and the users are described. But first, a specification of the domain and terminal generation is given.

5.1 SPECIFICATION FOR THE DOMAIN GENERATION

Domains of modules must be defined mask-by-mask. Only those masks can be selected, that have not yet a domain description. The shape of the domain boundary can vary from a **minimum circumscribing rectangle** (or bounding box) to a, Manhattan shaped, boundary polygon. The polygon is generated by a 'breath' operation: all relevant layout-details are made orthogonal first, then 'grown' in size, merged and 'shrunk' back. The user has a free choice for the value of the grow factor.

A lower bound on the value of the grow factor is estimated by the following reasoning:

> All areas, that cannot be used for interconnection purposes, must be joined to the domain area of that module.

The minimum distance between two layout-elements, that allows the placement of an interconnect wire of minimum width is: two times the clearance distance plus the minimal width of that wire. Fig 5.1 shows that the

minimal grow-factor must be half this distance.

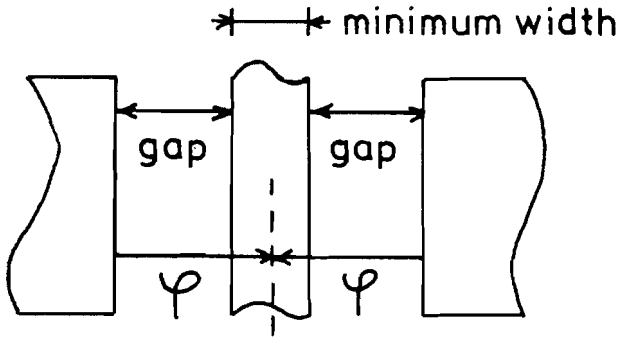


Fig. 5.1. A lower bound on the value of φ is determined by the gap-distance plus half the minimal-width.

The values of clearance distance and minimal width are derived from the design-rules of a technological process. These design-rules must also be available in the program. There is no limit on the upper bound of the value of the grow-factor. The shrink value is related to the grow value: the difference is always 0.5*d. In this way, domains are always placed on a minimal distance from the layout-elements, regardless of the value of the grow factor.

Special care must be given to the shrink operation. The grow and merge operation take care of an overlap of mask-details. This new, connected area must stay connected in the shrink operation.

The designer is also able to alter the domain description, that is a result of a 'breath' (grow-merge-shrink) operation. The domain description can be made more abstract (or rectangular) by adding rectangles. Holes can be filled and indentations can be smoothed (Fig. 5.2).

The last part of the generation of a domain area in a mask is an update of the VIC data-structure.

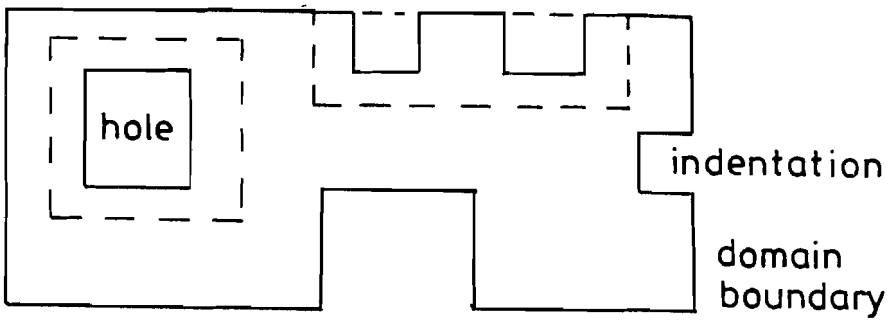


Fig. 5.2. Addition of rectangles after a 'breath' operation increases the abstraction-level.

5.2 SPECIFICATION FOR THE TERMINAL GENERATION

The terminals of a module are, in essence, made by pointing them out in the layout. Layout-data is moved from the contents part of a module to the terminal part. The user can help the program in two ways: he can give a network-description of the module and he can give layout-elements the name of a terminal. A network-description gives the names of the inputs and outputs of a module. Names of power-supply lines must be added to the i/o list; they are not enumerated in the network-description. Only those terminals can be defined, that have not yet a terminal description. All orthogonal layout-elements, with the same name as the terminal, are possible candidates for the terminal-assignment. Unnamed layout-elements with orthogonal shape can also be selected. When no network-description is available, the user can also define a new terminal name. It is treated in the same way as a terminal name from the i/o list.

After this object-selection, a simple edit session, consisting of creation and deletion of elements, can be started. In some cases, only a part of a polygon is sufficient as terminal description. The designer can add rectangles and dots to the terminal description. In this way, the realization of a module can be altered. This gives no loss of generality; wires and polygons are in fact built with sets of rectangles. Superfluous

elements can also be deleted.

At last, the set of selected and created elements can be made permanent by an update of the VIC data-structure. Elements, that were originally selected from the contents part of a module, must be deleted to avoid multiple declaration.

5.3 INTERFACE TO VOILA

The environment for the interactive generation of domains and terminals is depicted in fig. 5.3.

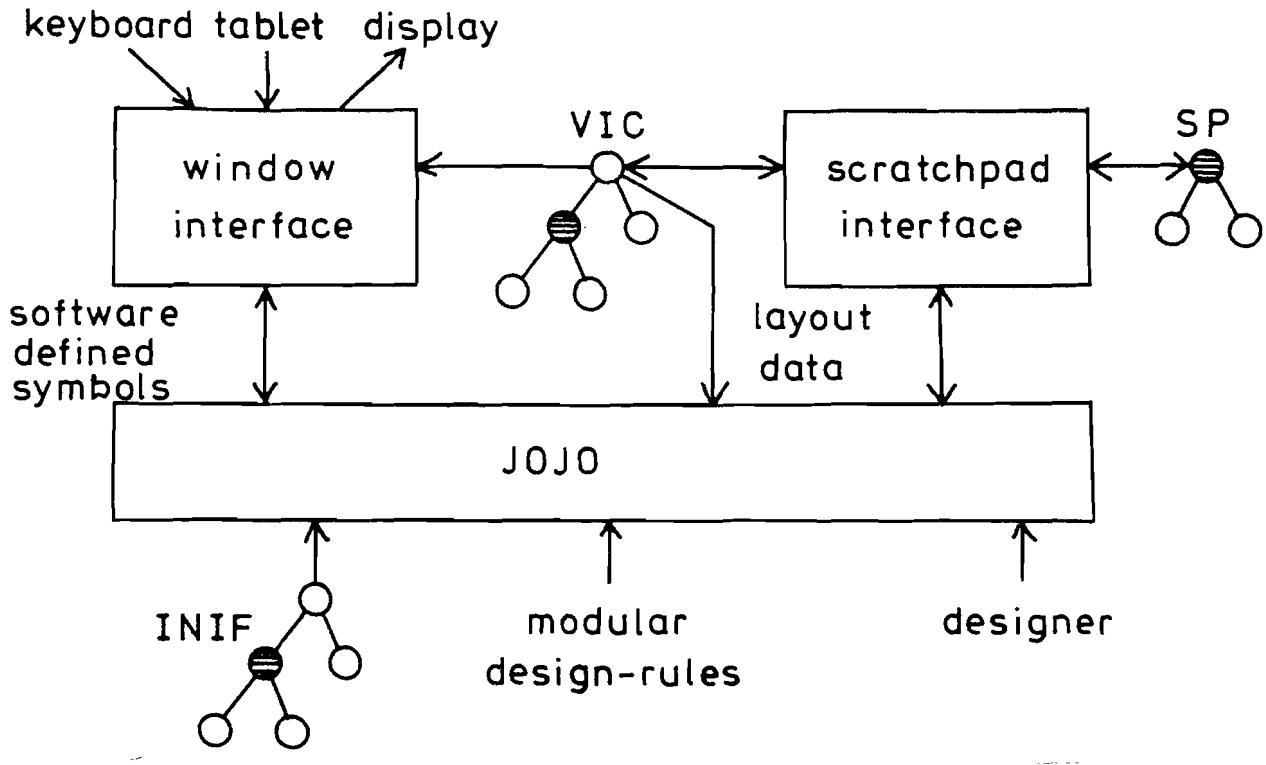


Fig. 5.3. Relation of JOJO with other VOILA programs.

The next sections describes the relevant VOILA data-structures and interface programs.

5.3.1 The VIC Data Structure

The layout-data is stored in the VIC Data-Structure. The VIC is defined recursively, as can be seen in the syntax diagrams in appendix A. Recursion does not occur by accident: a hierarchical defined layout contains modules, that are nested in modules etc... . Recursion allows us to write very compact search algorithms. There are many utilities in the VIC for creation and deletion of records, and for the implementation of fast search algorithms, by using hash-tables and circumscribing rectangles.

5.3.2 The Scratchpad Interface

The update of the VIC is controlled by the scratchpad interface. The scratchpad is an intermediate storage-medium for layout elements. The contents of the scratchpad is high-lighted on the layout screen of the editor for direct user-feedback. A set of utilities is available for copying data from VIC to scratchpad, for modification of objects in the scratchpad and for storing updates back to the VIC. Using these utilities guarantees a correct VIC and supports the integrety of the design. Real-time pictures are possible.

5.3.3 The Window Interface

This interface makes the interpretation of commands independent of the hardware. Command-menus and layout windows can be created and displayed automatically. Commands are translated to software-defined symbols. These symbols are interpreted by a **command interpreter** , according to a command language definition. The interpretation of commands results in operations on the VIC; possible commands are create, delete, copy, move etc... Selected commands on the menu-screen are high-lighted for direct

user-feedback.

5.3.4 The INIF Data Structure

The logical description of a module is also translated to a binary data-structure, called INIF for Inverted Network Intermediate Form. This data-structure is needed for the names of the inputs and outputs of a layout module. The INIF is not a one-to-one translation of the network-description. It is net-oriented, which means that inputs and outputs of modules of the same net are grouped. As with the VIC, a set of utilities is available to create and delete records, and to implement fast search-algorithms.

5.3.5 The Modular Design Rules

An internal data-structure is built by JOJO, that contains the technological design-rules. Two functions operate on this data-structure: the first one is called WIDTH, that computes the minimal allowable width of an element in a certain mask. The second function is called DISTANCE; it computes the minimal clearance between two elements in the same and in different masks.

5.4 INTERFACE TO THE USER

To give the user a guide for the generation of domains and terminals, an interactive language is developed. Its syntax and semantics are described in the next chapter. It contains some recepies in which the user can give commands and point to existing layout elements. The commands are entered with a mouse on a graphical tablet. The input data from the tablet is translated by the window-interface program to software-defined symbols,

that are interpreted according to the syntax of the interactive language. A screen is reserved for command-menu display and selected commands are high-lighted on the screen. A second screen is reserved for display of the layout and of a real-time cursor. The input from graphical tablet is also translated to coordinate values in the layout. The user must start JOJO from the VOILA graphics editor.

5.5 PROPERTIES OF THE INTERACTIVE LANGUAGE

The language, for the definition of domains and terminals, is defined as an LL(1) language. A parser of this language is based on the principle of **one symbol lookahead without backtracking** . This means, that the next symbol to be read, gives enough information to determine the next state of analysis. None of the steps have to be revoked later on. The parsing of an LL(1) language is straight-forward and is described in full detail in [WIR 76].

The **parser** of this language can be generated in a stepwise-refinement way. A procedure is made for every production rule. In an interactive design-environment, a syntax violation may never lead to a stop of the analysis of the language. An **error-recovery mechanism** is implemented in the parser. Parsing constructs are embedded in while loops that will be executed until the right alternative is chosen by the designer. In the mean time, an overview is given of the possible alternatives of the language.

As an example, the parsing of the following syntax description:

```
< Command >   :==  SCRATCH  |  KEEP
```

is given by :

```
PROCEDURE Command;

VAR
   oke : boolean;
```

```
BEGIN
   oke := false;
   WHILE not oke DO
   BEGIN
      IF sym = SCRATCH THEN
      BEGIN
         oke := true;
         writeln('Scratch command accepted');
         Getsym;
      END
      ELSE
         IF sym = KEEP THEN
         BEGIN
            oke := true;
            writeln('Keep command accepted');
            Getsym
         END
         ELSE
         BEGIN
            writeln(' Please give SCRATCH or KEEP:');
            Getsym
         END
   END    {while}
END       {Command };
```

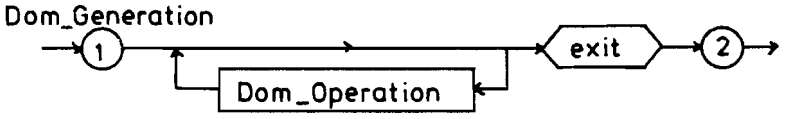
The procedure Getsym reads the next input symbol. It is available in the variable sym.

CHAPTER 6

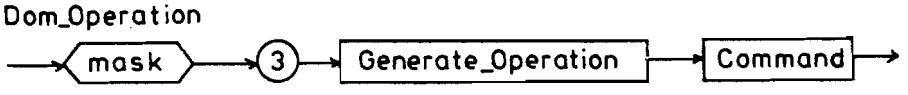
IMPLEMENTATION ASPECTS

This chapter contains a description of the syntax and semantics of the interactive domain and terminal generation language. It gives, in the mean time, a short description of JOJO.

6.1 IMPLEMENTATION OF THE DOMAIN GENERATION

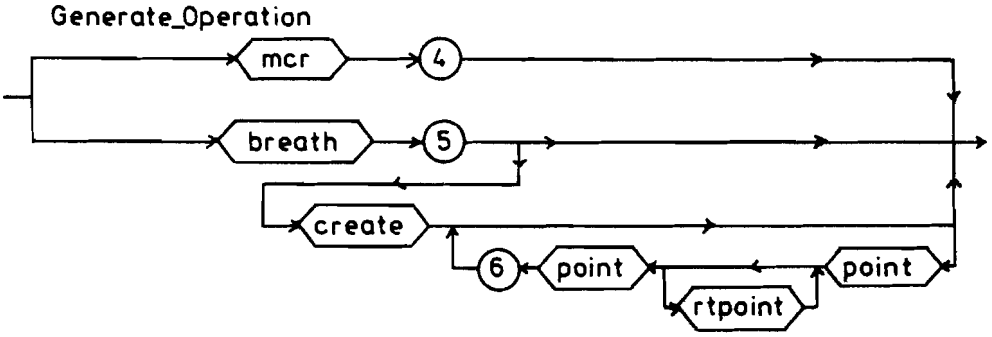


Before the domain generation is started, some preprocessing work must be done. An overview of the masks, that have no domain description, is given on the menu-screen (1). Already existing domains must be modified in the VOILA graphics editor. The layout-rules are read from an external disk-file. In the next step, zero or more domains can be generated for one of the listed masks on the menu-screen. When the domain generation is left, an EXIT command must be given. A warning is generated, when some masks have no domain description (2).



The actual domain-generation operation consists of the selection of a mask on the menu-screen (MASK). The internal book-keeping is updated in

(3). It is followed by a part, in which the user can define which kind of domain shape must be generated. It is finished with a part in which a command must be given.



The generate operation is started by giving the commands MCR or BREATH. They generate a minimum circumscribing rectangle or a, more tight fitting, set of rectangles. The set of rectangles defines the contents of a polygon.

Starting with an MCR command, a scanner of the VIC is started in (4). It expands wires, polygons, array constructs and instantiated sub-modules. The two diagonal edge-points of the bouding box are estimated and the resulting rectangle is expanded with a factor of 0.5*d. It is written to the scratchpad and is automatically high-lighted.

When the user gives a BREATH command, a scanner is started, that selects all relevant layout-data from the VIC (5). The layout-elements are made orthogonal first. Appendix D contains the source listing in pseudo-code of the implementation of the scanner. The contents-data of sub-modules, that have a domain description, does not have to be expanded. Expansion of the terminals of the sub-modules is however necessary; terminals are a part of the realization of a sub-module that must be protected by the domain of the next higher module in hierarchy. The contents-data of sub-modules must be expanded when no domains are specified. This process must be repeated until a, deeper nested, module with a domain description is found.

Domains of sub-modules must be treated in a different way than elements from the contents part. The grow and shrink values must be the same. When the two values are not the same, a nested set of hierarchical defined domain areas would result in non-connectable terminals (Fig 6.1).

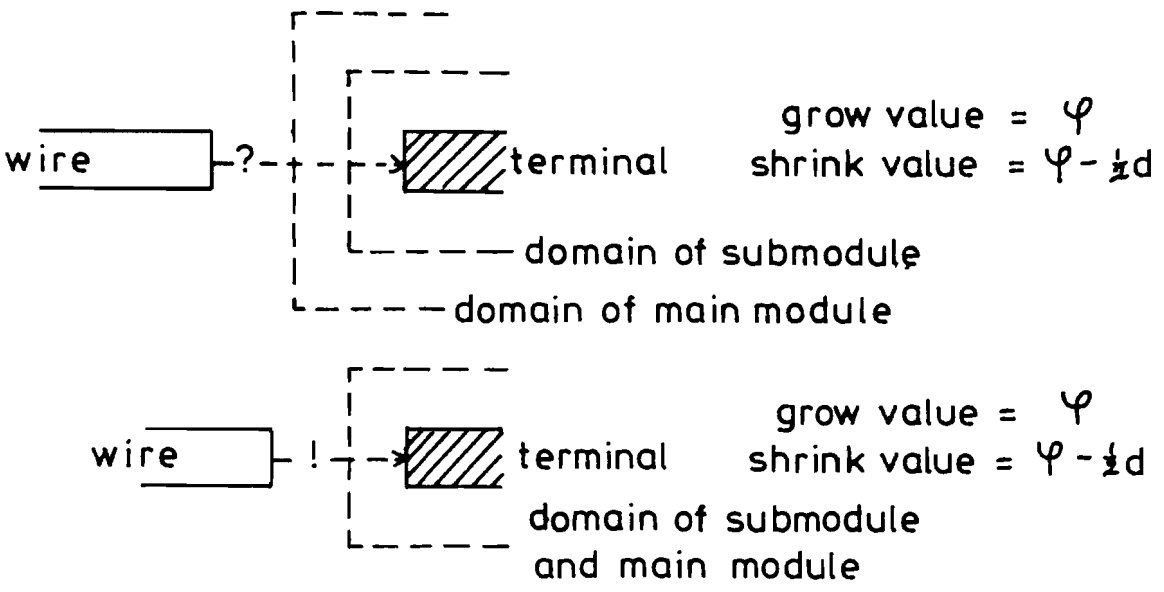
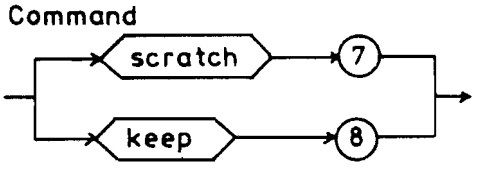


Fig. 6.1. Different grow- and shrink-values of domains may lead to non-connectable terminals.

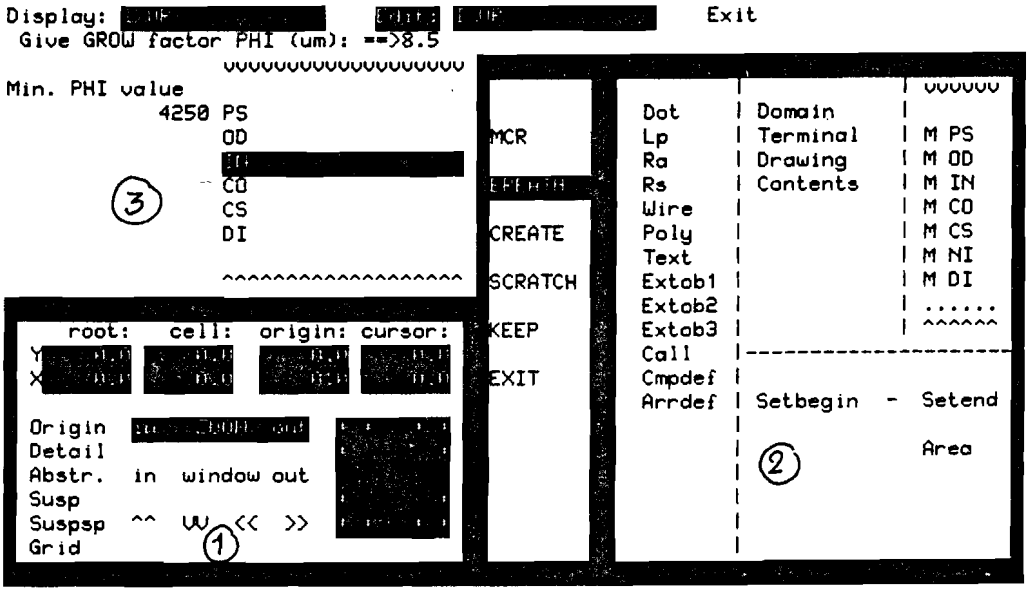
After a BREATH operation, the designer is also able to add rectangles to the domain description, by giving a CREATE command. Rectangles are displayed real-time (RTPOINT) and are, when finished, moved to scratchpad (6). Addition of rectangles allows the generation of a more abstract (or more rectangular) domain shape.



The description of the domain area is now stored in an intermediate data-structure: the scrathpad (SP). It can be made permanent by an update of the VIC with a KEEP command (8). The name of the mask is removed from the menu-screen. A SCRATCH command will clear the contents of the

scratchpad (7).

As an example, the following picture was taken from the menu-screen at domain generation time.



Three main parts can be distinguished in this picture:

1. The part, marked with 1 gives an overview of the VOILA display facilities. Coordinates of the origin of layout-modules and of the cursor are displayed. Furthermore, window and zoom facilities are available.

2. The second part gives an overview of the context setting. Operations on layout objects can be restricted to domain, terminal, drawing or contents part or a combination of the four parts. The same counts for mask-selection, although this is not used in the domain generation. At last, a set of object kinds can be defined on which operations must occur.

3. The third part shows the situation, in which for the IN mask, a bounding polygon had to be generated. This was done via the BREATH command. The minimum allowable grow factor is derived from the layout-rules (4250 nm or 4.25 um) and is displayed in the upper lekt corner of the screen. We have reached the situation, in which the user has entered a grow-value of 8.25 um via the key-board.

6.2 DOMAIN GENERATION ALGORITHM

The commands MCR and BREATH start a scanner of the VIC data-structure. In appendix D, a source listing in pseudo-code is given to illustrate the operations, that must be performed. It also shows the recursive structure of the scanner.

The algorithm for the determination of a domain area of module M in mask N uses two lists (A and B) for the storage of layout-data. The two lists are filled in the following way:

1. Take the contents-part of module M. Expand array-constructs, wires and polygons. Slanted boundaries must be made orthogonal. Assign these elements to list A.

2. In case of an instantiated sub-module, check for the existance of a domain area of this sub-module. If it exists, add domain-data to list B.

3. Expand all terminals of the sub-module. Terminals are a part of the realization of a sub-module and must be expanded. Add this data to list A.

4. If there is no domain-description of a sub-module, expand all contents-data of this module and add it to list A.

At point 4, we have reached in fact the situation of point 1. We have created a recursive algorithm for the determination of domain-areas. The scanner for the domain-generation uses global variables to keep books of which layout part has to be expanded and to which list the data must be sent. The next table shows a hierarchical defined module M, in which three sub-levels can be distinguished. An overview is given of the two lists, to which the domain, terminal and contents data must be sent.

part	module M	sub module	sub-sub module	sub-sub-sub module
domain	-	-	list B	-
terminals	-	list A	list A	-
contents	list A	list A	-	-

The 'breath' operation is implemented as:

1. Grow data from list A with a factor PHI

2. Grow data from list B with a factor PHI - 0.5*d

3. Merge the two lists

4. Shrink the data of the resulting list with a factor PHI - 0.5*d

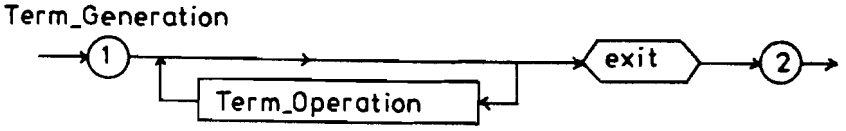
A grow operation can be done without knowledge of layout-details in the environment. This operation results in overlapping rectangles. A merge operation is, in this algorithm, not more than a concatenation of lists A and B. A shrink operation cannot be done without knowledge of environment-details. A shrink operation is a local operation that is even capable of deleting some layout-data (Fig. 6.2).

Fig. 6.2. A shrink operation on mask-data is a local operation and can let some areas disappear.

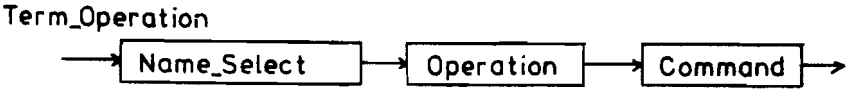
Fortunately, the shrink operation is a standard VOILA utility, that is also used in the extension of mask-lists. The standard shrink-utility can only handle orthogonal data, but this was obligatory in the definition of the abstraction of a module.

6.3 IMPLEMENTATION OF THE TERMINAL GENERATION

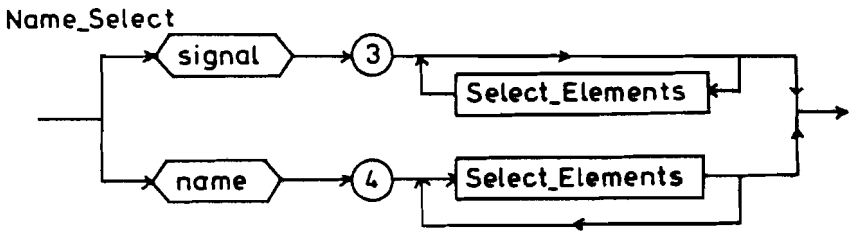
The syntax and semantics of the terminal generation language are described in this section.



Some preprocessing work must be done before the actual terminal generation is started (1). The names of input and output terminals are read from a network description, if it is specified by the user. Names of power supply lines are added to the list. The names of those terminals, that have not yet a terminal description, are listed on the menu-screen. In the next step, zero or more terminals can be generated. An EXIT command must be given when the terminal generation is left. A warning is generated, when there are still some terminals not defined (2).



The actual terminal assignment consists of the selection of a terminal name, followed by an operation, that allows simple editing on selected layout-data. It is finished with a section where a command must be given.



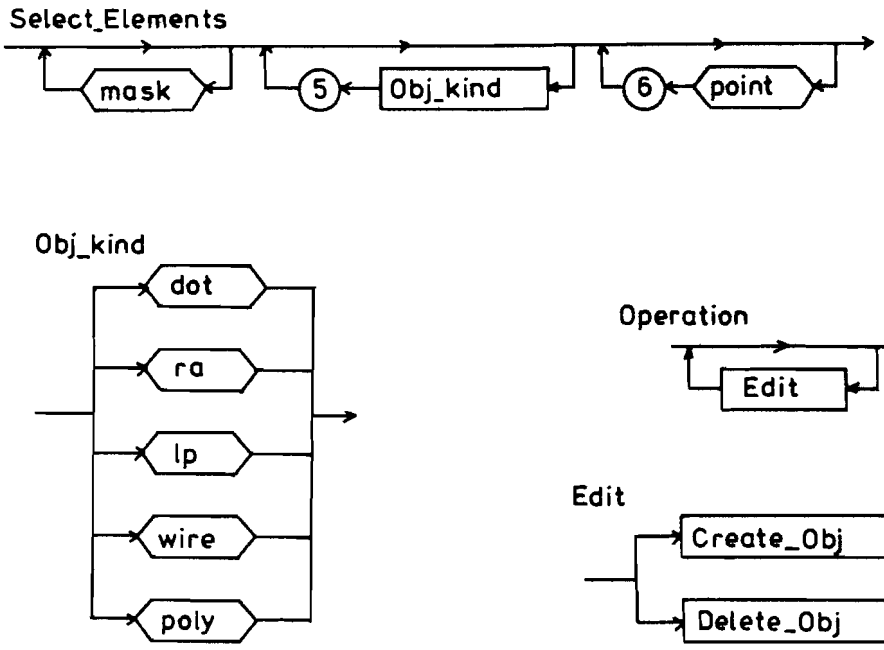
The selection of a terminal name can be made in two ways:

1. A SIGNAL command allows the selection of a name from the list on the menu-screen. It originated from either network description or power-supply lines. Layout-elements, with the same name as the selected terminal, are moved to the scratchpad (3). The user can also point to zero or more layout-elements in the layout.

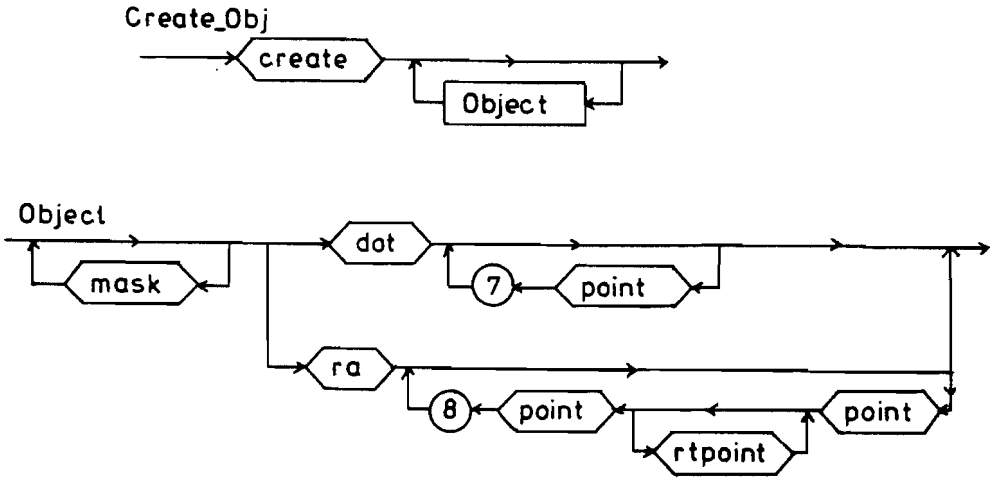
2. A NAME command reads a terminal name from the keyboard, in case of a non-existing or non-complete network-description (4). The user must select one or more layout-elements on the layout-screen. They are also copied to the scratchpad.

The selection of layout-elements requires a context, that defines the kind of element, that must be selected. The context consists of a mask, the object-kind and a layout-point. Only one single mask can be selected from the menu-screen. From the layout-element types, available in VOILA, the following five types are allowed: dot, line-piece, rectangle, wire and poly. The user must give a layout-point to start a scanner of the VIC (6). Only elements, with orthogonal boundaries are moved to the scratchpad. The name of the actual terminal is assigned to all scratchpad elements.

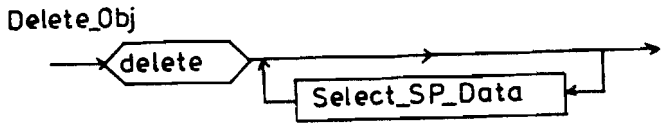
After the element selection, zero or more edit-sessions can be started.

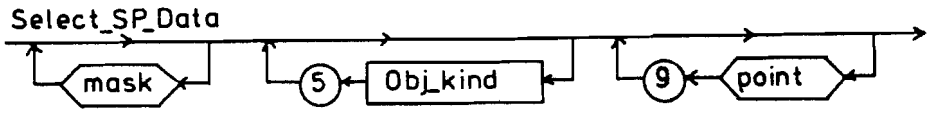


An edit-session consists of a simple creation or deletion of orthogonal elements in the scratchpad.

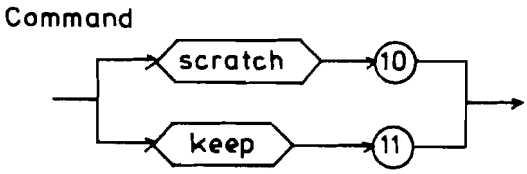


The creation mode of the edit-session is started with a CREATE command. Zero or objects can be created in a certain MASK. The designer has the ability to alter the realization of a module by creatin rectangles (RA) or dots (DOT). The newly created elements are displayed real-time (RTPOINT) an finally written to scratchpad (7 and 8).





A delete command allows the user to remove superfluous elements from the sratchpad. The selection of elements from the scratchpad requires a similar context as the selection from the VIC (Select_Elements).

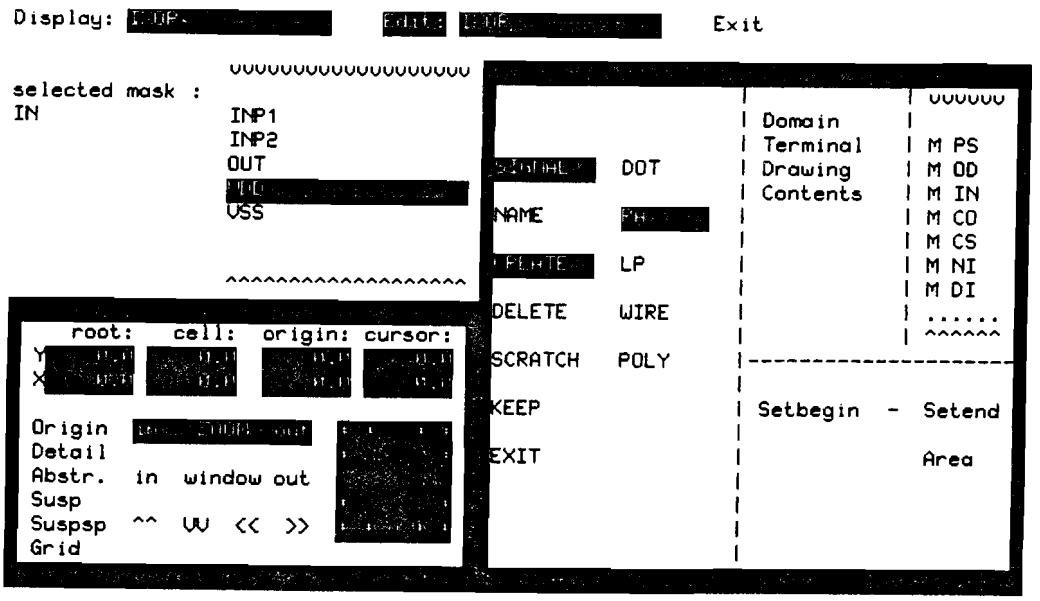


The scratchpad data can be made permanent by an update of the VIC with a KEEP command (11). The number of elements in scratchpad is counted. When there is more than one element, a signal compound is built around all elements. This indicates subcell-connectedness. Terminal elements are 'denamed' first, and the name of the actual terminal is assigned to the signal compound. Selected elements from the contents part will be deleted to avoid multiple declarations. The terminal name is removed from the menu-screen.

The SCRATCH command clears the contents of the scratchpad (10).

As an example, the next picture was copied from the menu-screen at terminal generation time.

The picture shows the situation, in which a SIGNAL command was given and the terminal name VDD was selected. Layout-elements were moved to scratchpad and the designer decided to start a CREATE session. Rectangles (RA) in mask IN can be added to the contents of the scratchpad.



CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

7.1 CONCLUSIONS

* The layout-phase is the most critical phase in the design of integrated circuits. Hierarchical layout-methodologies must be applied to tackle the VLSI complexity problem and to reduce the design time. Hierarchical abstraction, as is supported by VOILA, reduces the amount of detail, that must be handled at any time. Many lessons can be learned from the Software-Crisis in the 1960's. VLSI layout has a few extra problems due to the two-dimensional character of the silicon medium, in which the circuits have to be realized.

* In traditional hardware design, a minimization of the number of active devices is made. In VLSI layout, an optimization of area and speed is necessary; interconnect is more expensive than a few extra devices.

* The domains and terminals are hidden in the contents description of a module. Automatic terminal assignment is a very complex problem and can only be applied in some special cases. Automatic domain generation can lead to non-connectable terminals. The generation of domains and terminals should be done in a combination of an interactive and automatic way; the program gives propositions an the designer can alter these propositions by giving more information. The designer must have full control over the generation process.

* A language is developed for the interactive generation of domains and terminals. It contains some recepies, that must be applied and leads to an update of the VIC data-structure. The user can enter commands on a graphical tablet with a mouse and point to existing layout-data.

  The interactive domain generation allows the generation of a minimum circumscribing rectangle (bounding box) and a bounding polygon. The bounding polygon is made with a 'breath' algorithm: layout-elements are made orthogonal first, then 'grown' in size, merged and 'shrunk' back. The designer can choose the shape of the domain by giving a value for the grow factor and by adding rectangles to the result of the 'breath' operation.

* The terminals are generated, in essence, by selecting existing layout-elements from the contents part and moving them to the terminal part of a module. The user can help the program by giving a network-description of a module, in which the names of input and output terminals are comprised. Names of power-supply lines must be added to

the list; they do not appear in the i/o list of the network-description. The user can also help by assigning names of terminals to existing layout-details.

* The domain and terminal generation languages are defined as LL(1) languages. The parsers of these languages are equiped with an error-recovery mechanism. This is indispensable in an interactive design environment; syntax errors may not lead to a stop of the syntax analysis. Parsing constructs are embedded in while loops, that will be executed until the right alternative is chosen.

* The domain and terminal generation are started from the VOILA layout editor. In this way, standard utilities could be used; e.g. definition of commands on the menu-screen, hardware independent input and output of menus and layout data and the use of an intermediate storage medium, the scratchpad.

* The performance of JOJO, in case of the domain generation is hard to measure. This is due to circumstances, that cannot be controlled by JOJO. The first one is the existance of domains of sub-modules: they determine if the details from the contents part of these sub-module have to be expanded or not. The second one is the number of elements (or complexity) of the domain description of sub-modules. A bounding-box description greatly speeds up the generation process. The performance of JOJO, in case of the terminal generation, cannot be measured at all, because it is a completely interactive process.

7.2 RECOMMENDATIONS

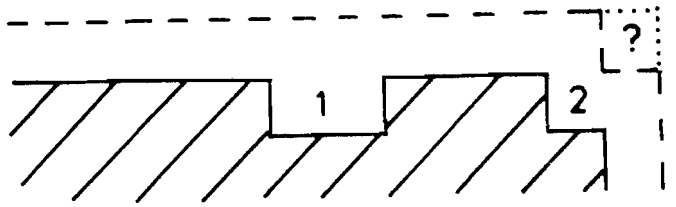
* It seems convenient to me that a utility should be provided, for the selection of terminals and interconnect-wires by name. The selection by context (mask plus object-kind) is in some cases insufficient. At the same time, an overview of existing terminal-names should be given with an indication of sub- or supercell connectedness.

* In the terminal generation, an extension of the names of the i/o list from the network-description with 'VDD' and 'VSS' is implemented. There is however, in the VIC a technology record, in which all global signals are enumerated. The extension of the terminal list could in fact be copied from this global signal-list.

* It is very difficult to determine the terminals of a PLA module in retrospect. A PLA contains no hierarchy at all: all standard library patterns are placed in two 2-dimensional planes; the AND and the OR plane. It would be convenient for the terminal generation if the coordinates of the inputs and outputs were output of the PLA generator. The designer has in fact put this information a priori in the PLA generator.

* The terminal-class concept is not implemented in the first version terminal generation. It was not implemented in the first version of VOILA. This powerful concept, and possible extensions should also be implemented in the next versions of the terminal generation. The designer must have the possibility to enter terminal-class names on the keyboard and assign them to previously defined terminals.

* A lot of effort is put in the smoothing of indentations in the domain-boundary.



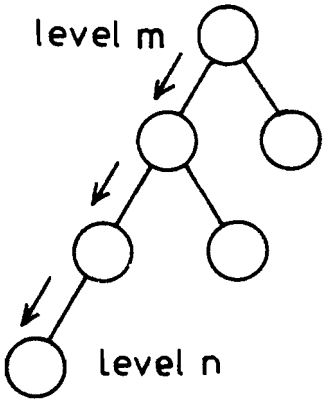
It is, with a 'breath' algorithm, impossible to assign area 2 to the domain area. A large value of the grow factor will result in the disappearance of area 1 but not of area 2. A possible answer to this problem is the estimation of a circumscribing rectangle and the percentage of area 2 in regard to the circumscribing rectangle. Surfaces, below a certain percentage, could be assigned to the domain area.

* To enable the generation of symbolic domains, it must be possible to perform some boolean operations on the mask-data first, before a 'breath' algorithm is started. It seems also convenient to me to define an area, in which a certain grow-operation could take place. In this way, the user can trim the domain boundary locally, e.g. the right side of the domain can be rectangular and the upper side must be made with a 'breath' algorithm with a small grow factor.

* In the actual implementation of JOJO, an ad-hoc solution for the object-selection is chosen. The object-selection has difficulties with the selection of elements, that are described in an array construct. Elements are in fact copied to the terminal part, so in some cases a double declaration of elements is made. The powerful object-selection mechanism from the VOILA editor should be used instead. The problem of array-splitting is solved here. An other advantage is that a user does not has to work with a different object selection mechanism in domain

and terminal generation.

* Hierarchical defined layouts have also a draw-back: a module must always be connected via the next higher module in hierarchy. This is quite reasonable for input and output terminals. For global power-supply lines, it is not. If the power-supply line of a primitive module in level n, must be connected to a module in level m, all interconnect elements have to be copied in the intermediate m-n levels!



This restriction in VOILA does not occur in programming languages. The scope of global variables covers an unnumerable number of sub-levels, if the variable has not been redeclared in the mean time. It seems convenient to me that a hierarchical scope of a terminal or global signal could be defined to overcome the m-n multiple declaration.

* Wires, polygons and slanted rectangles are 'rounded' to a single rectangle. An other approach could be the approximation by an orthogonal polygon, in which the sides have a staircase shape.

* No attention is given to the interactive drawing generation. The user should have possibilities to place graphical comment in the layout of a module. Some utilities of adding information about a function of a module, about the placement in the layout and perhaps about the abstraction should be provided.

* This report contains no contributions to increase the design testability in hierarchical defined systems. This is perhaps a nice subject for further investigations.

* In case of the automatic terminal assignment, knowledge of the internal structure, and hence the connectivity within a module, is necessary. The connectivity is different in different technological processes. In JOJO, the terminal generation is implemented in a pure interactive way. There must also be ways to apply some automation in this area. This is perhaps also a subject for further investigations.

Eindhoven, 23 august 1984

REFERENCES

[BER 83] : van Berkel, K., "VLSI-layout Design requires a New Language", Proceedings ICCD 1983/ VLSI in Computers, Nov. 1983.

[ELL 83] : Ellis, S.A. et al., "A Symbolic Layout Design System", Proceeding of the International Symposium on Circuits and Systems (ISCAS), 1982, pp 670-675

[ISC 84] : Suzuki, S. et al., "A 128 k Word * 8 b DRAM", IEEE Int. Solid State Circuits Conference, Feb. 1984, pp 106-107.
Yamada, J. et al., "A Submicron VLSI Memory with a 4b-at-a-Time ECC circuit", ISSCC Feb. 1984, pp 104-105.

[LAT 79] : Lattin, W., "VLSI Design Methodology, the Problem of the 80's for Microprocessor Design", 16th Design Automation Conference Proceedings, San Diego, June 1979, pp 548,9.

[LOS 80] : Losleben, P., Chapter 4: Computer Aided Design for VLSI , in Very Large Scale Integration (VLSI): fundamentals and applications, ed. Barbe, D.F., Springer Berlin 1980, Springer Series in Electrophysics, 5.

[MIL 56] : Miller, G. A., "The magical number seven, plus or minus two : Some Limitations on our capacity for processing information", Psychol. Review, Vol. 63, 1956, pp 81-97.

[MOS 81] : Mosteller, R. C., "REST, A Leaf Cell Design System", in VLSI 81, pp 163-172.

[NIE 83] : Niessen, C., "Hierarchical Design Methodologies and Tools for VLSI Chips", Proceedings of the IEEE, Vol. 71, No. 1, January 1983, pp 66-75.

[ROW 80] : Rowson, J.A., "Understanding Hierarchical Design ", Technical Report (Ph. D. Thesis), April 1980, Computer Science Department, California Institute of Technology

[SEQ 83] : Sequin, C., "Managing VLSI Complexity", Proceedings of the IEEE, Vol. 71, No. 1, January 1983, pp 149-166.

REFERENCES

[SIM 62] : Simon, H. J., "The Architecture of Complexity", Procedure of the American Philosophical Society, Vol. 106, No. 6, Dec. 1962.

[SIM 84] : SIMON Usermanual 1984, Rel. 2.20 March 1984, Philips-Elcoma Nijmegen, Valvo RHW Hamburg.

[STR 81] : Stratford, R.I., "Delila - Design of Large Integrated Circuits", European Conference on Electronic Design Automation, Sept. 1981, pp 106-109.

[TUC 82] : Tucker, M. and Scheffer, L., "A Constrainted Design Methodology for VLSI", VLSI Design, May/June 1982, pp 60-65.

[VOI 84] : Niessen, C. et al., "CAD for VLSI layout becomes reality" PHILIPS internal prospect.

[WEI 79] : Losleben, P., "Computer Aided Design for VLSI", Chapter 4 in VLSI, ed. Barbe 1980, ref. 2:
Weimann, W., "VLSI Design at Motorola", 1979 Design Automation Workshop.

[WEL 82] : Welsh, J. and Elder, J.: Introduction to Pascal, second edition, Prentice-Hall 1982.

[WES 81] : Weste, N. and Ackland, B., "A Pragmatic Approach to Topological Symbolic Design", in VLSI 81, pp 117- 129.

[WHI 81] : Whitney, T., "A Hierarchical Design Analysis Front End", in VLSI 81, pp 217-225.

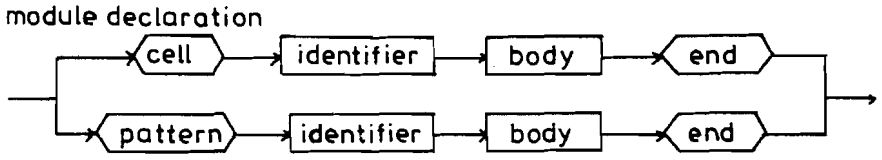
[WIR 76] : Wirth, N., chapter 5 in Algorithms + Data Structures = Programs, Prentice-Hall, New Jersey, 1976.

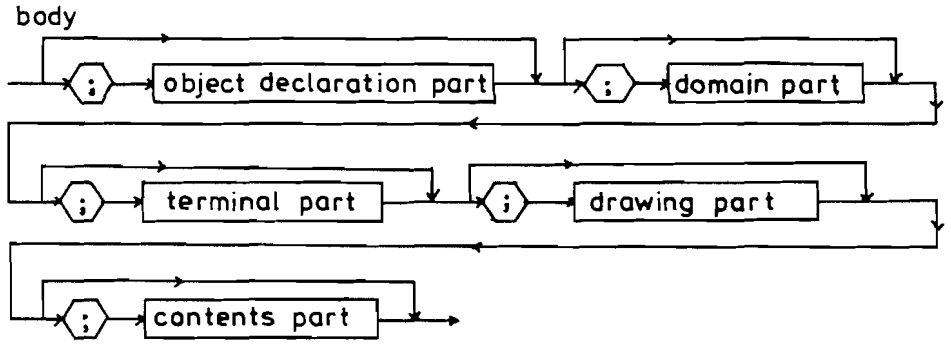
[WOL 81] : Wolff,J.A. and Farr, E.C., "MDS - A Design Automation System based on Macros", European Conference on Electronic Design Automation, Sept 1981, pp 272-276.

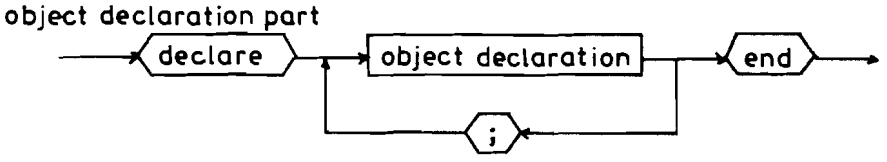
APPENDIX A

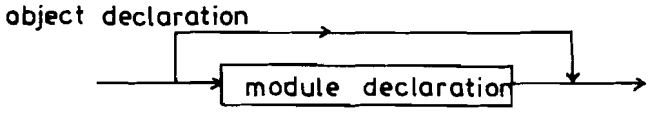
SIMPLIFIED VLDL SYNTAX GRAPHS

This appendix contains a selection of those VLDL constructs, that are necessary for the generation of domains and terminals. At some points, some simplifications were made. So these syntax diagrams should not be regarded as official VLDL syntax diagrams.

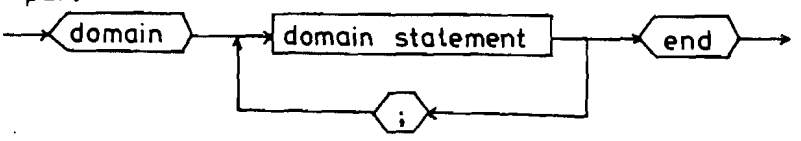




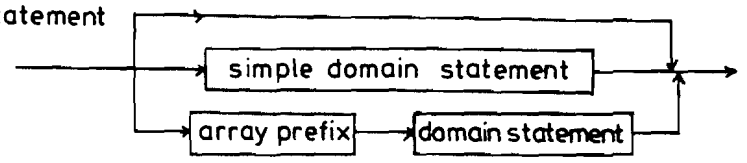




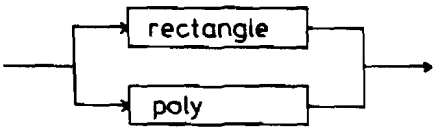
domain part



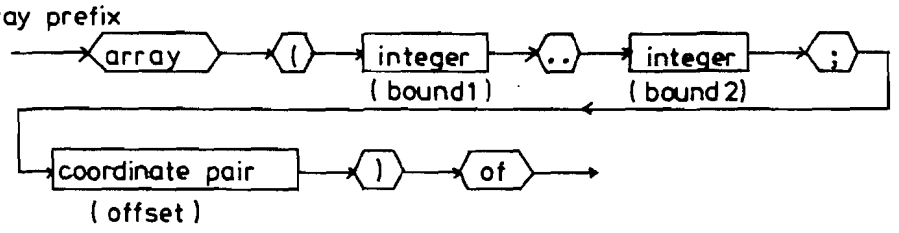
domain statement



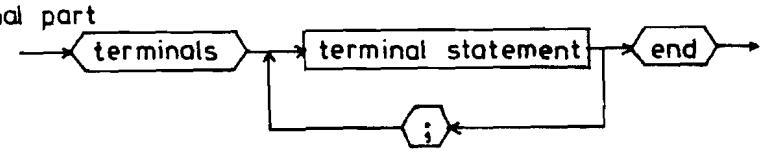
simple domain statement



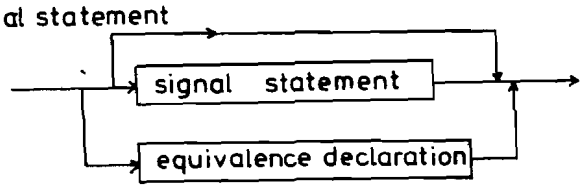
array prefix



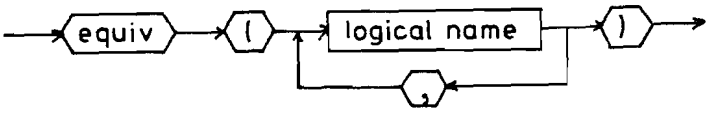
terminal part

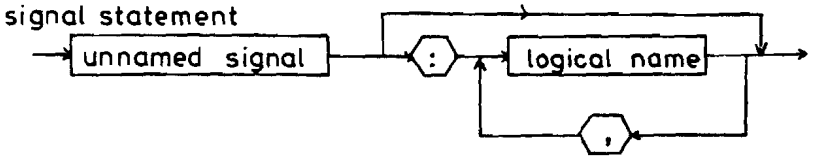


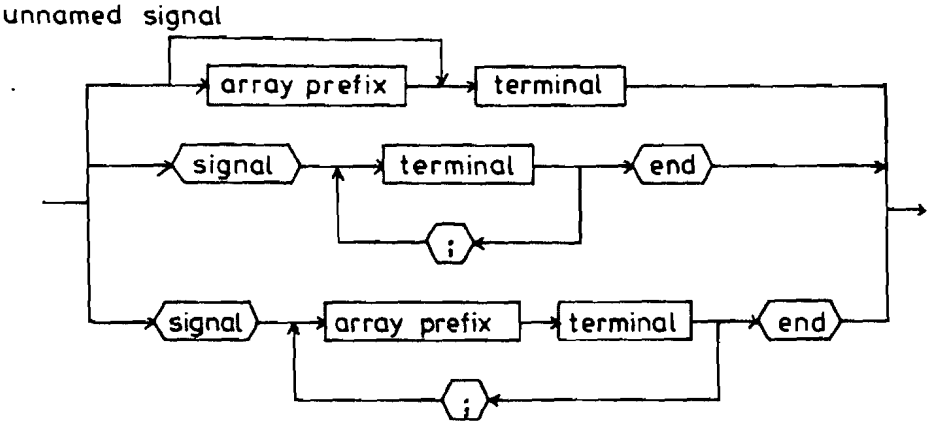
terminal statement

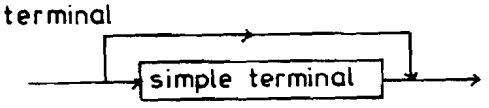


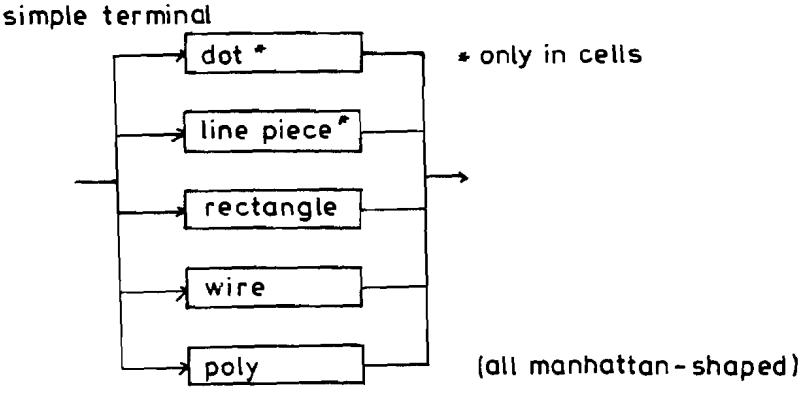
equivalence declaration



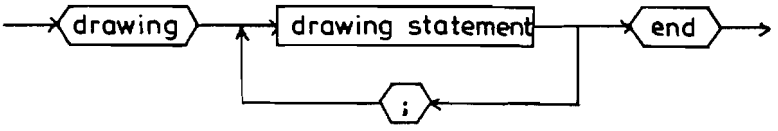

signal statement
unnamed signal
:
logical name
,


unnamed signal
array prefix
terminal
signal
terminal
end
;
signal
array prefix
terminal
end
;

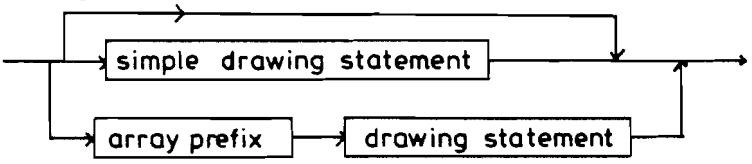

terminal
simple terminal


simple terminal
dot *
line piece*
rectangle
wire
poly
* only in cells
(all manhattan-shaped)

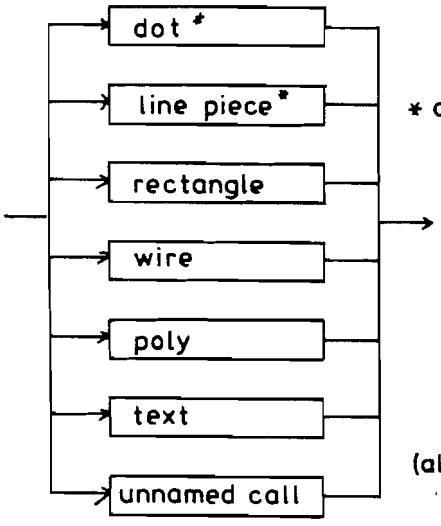
drawing part



drawing statement

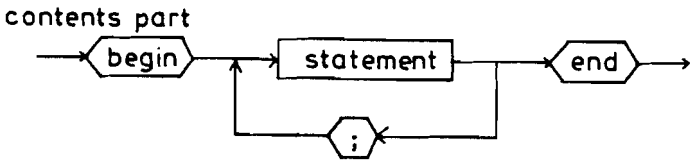


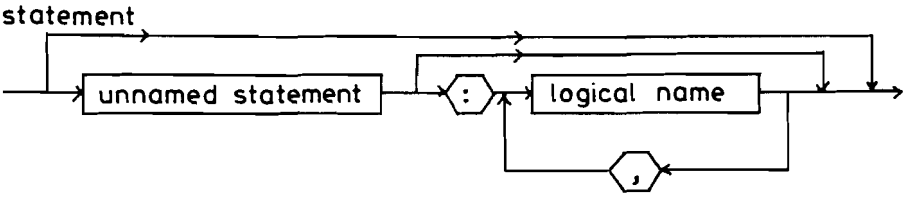
simple drawing statement

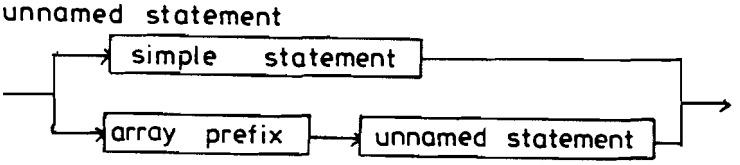


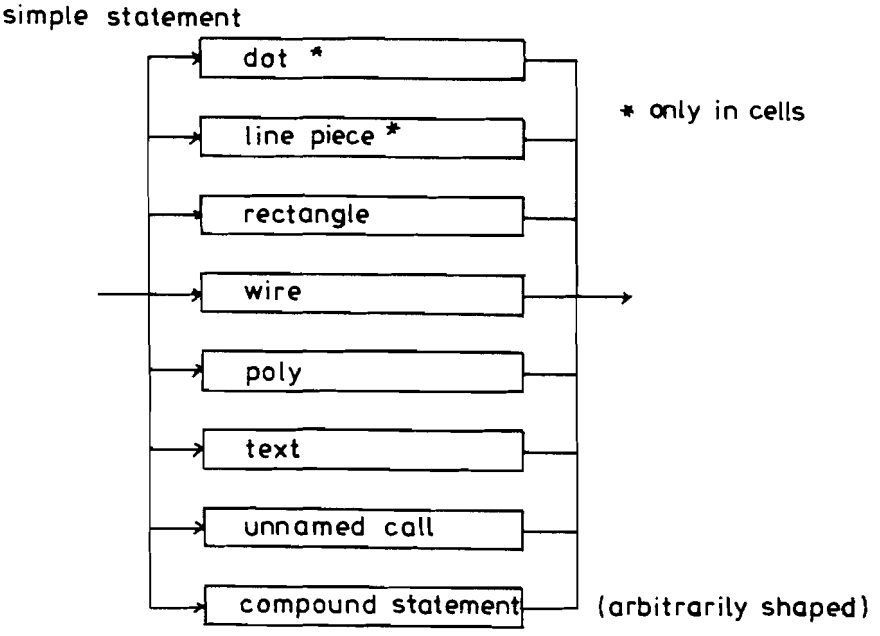
* only in cells

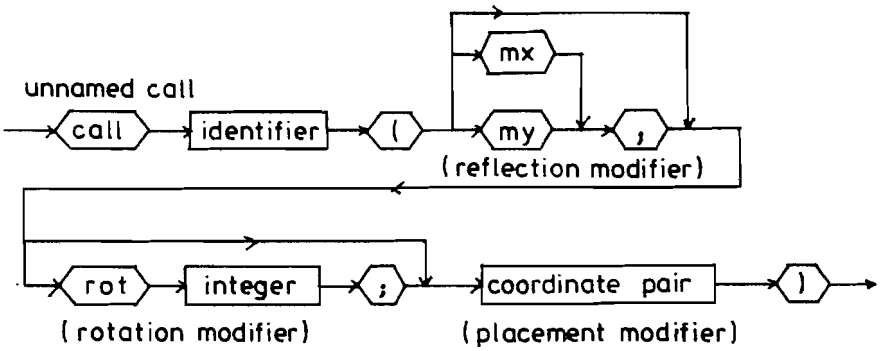
(all manhattan-shaped)

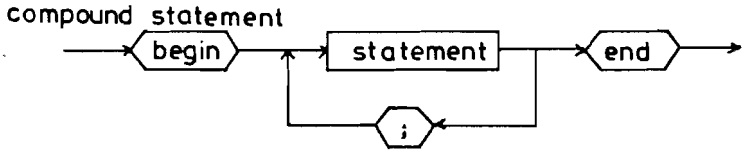

contents part
begin
statement
end
;

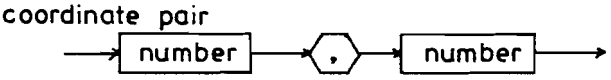

statement
unnamed statement
:
logical name
,

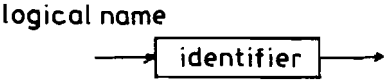

unnamed statement
simple statement
array prefix
unnamed statement

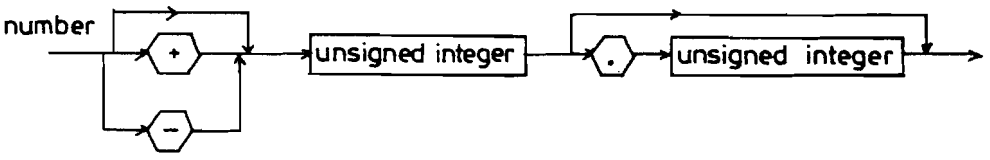

simple statement
dot *
line piece *
rectangle
wire
poly
text
unnamed call
compound statement
* only in cells
(arbitrarily shaped)

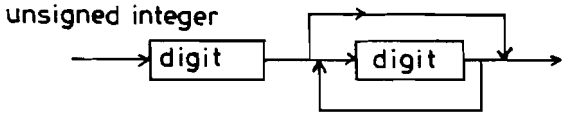

unnamed call
call
identifier
(
mx
my
,
(reflection modifier)
rot
integer
;
coordinate pair
)
(rotation modifier)
(placement modifier)

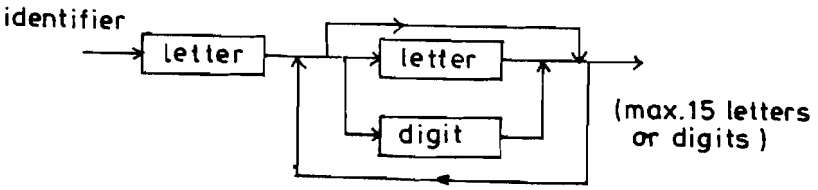

compound statement
begin
statement
end
;

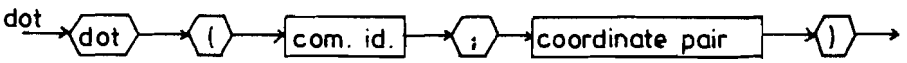

coordinate pair
number
,
number

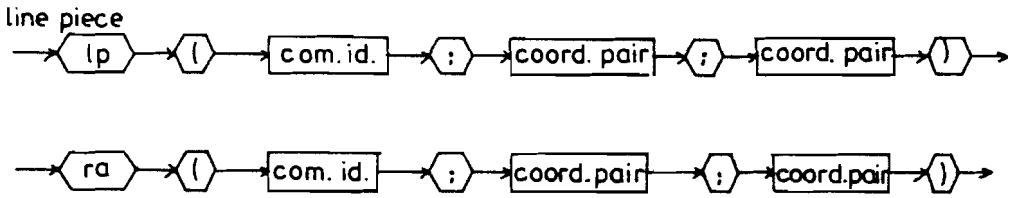

logical name
identifier

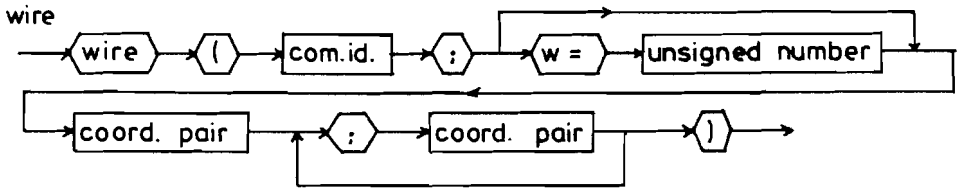

number
+
-
unsigned integer
.
unsigned integer

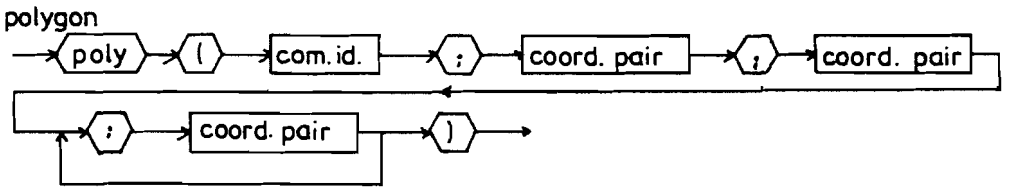

unsigned integer
digit
digit

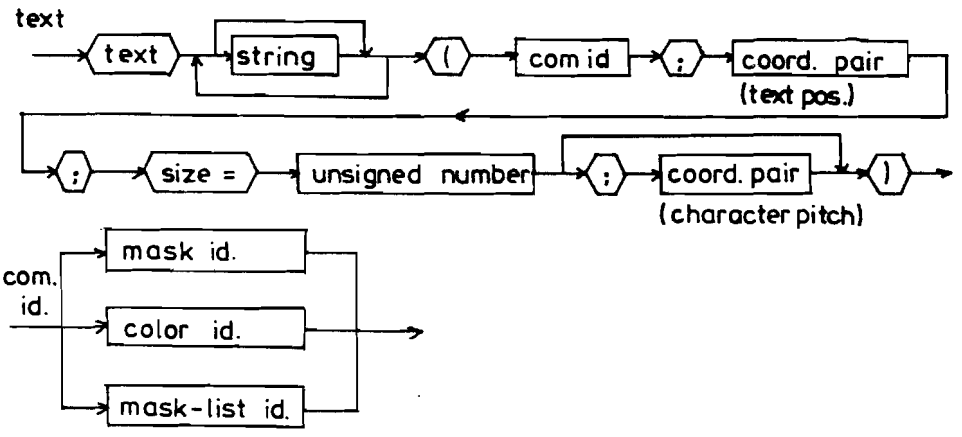

identifier
letter
letter
digit
(max.15 letters
or digits)


dot
dot
(
com. id.
;
coordinate pair
)


line piece
lp
(
com.id.
:
coord. pair
;
coord. pair
)
ra
(
com. id.
:
coord.pair
;
coord.pair
)


wire
wire
(
com.id.
:
w =
unsigned number
coord. pair
:
coord. pair
)


polygon
poly
(
com.id.
:
coord. pair
;
coord. pair
:
coord. pair
)


text
text
string
(
com id
:
coord. pair
(text pos.)
:
size =
unsigned number
:
coord. pair
)
(character pitch)
com.
id.
mask id.
color id.
mask-list id.

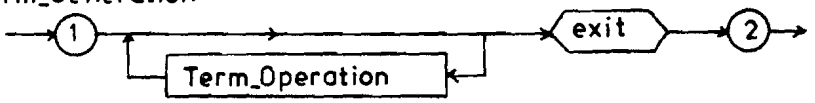
APPENDIX B

JOJO SYNTAX DEFINITION AND GRAPHS

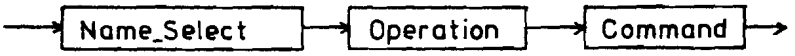
The syntax definition of the terminal generation is defined as:

```
< Term_Generation > :== { < Term_Operation > } EXIT

< Term_Operation >  :== < Name_Select > < Operation > < Command >

< Name_Select >     :== SIGNAL { < Select_Elements > } |
                        NAME  < Select_Elements > { < Select_Elements > }

< Select_Elements > :== { MASK } { < Obj_Kind > } { POINT }

< Obj_Kind >        :== DOT | RA | LP | WIRE | POLY

< Operation >       :== { < Edit > }

< Edit >            :== < Create_Obj > | < Delete_Obj >

< Create_Obj >      :== CREATE { < Object > }

< Delete_Obj >      :== DELETE { < Select_SP_Data > }

< Object >          :== { MASK }  [ DOT { POINT } |
                                    RA { POINT { RTPOINT } POINT } ]

< Select_SP_Data >  :== { MASK } { < Obj_Kind > } { POINT }

< Command >         :== SCRATCH | KEEP
```

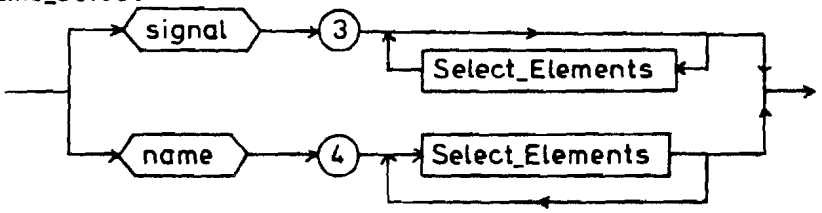
Term_Generation



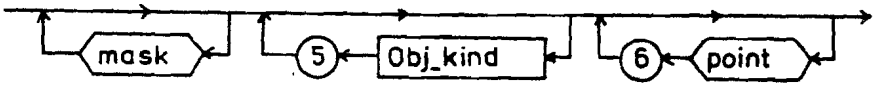
Term_Operation



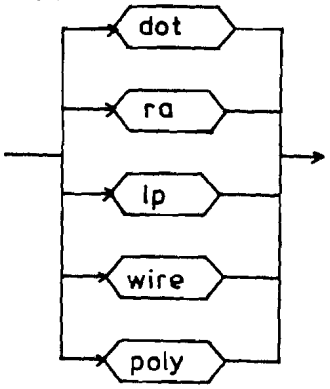
Name_Select



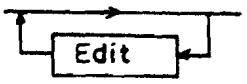
Select_Elements



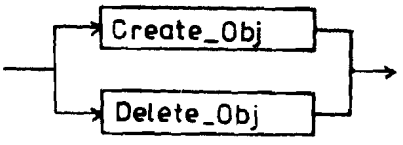
Obj_kind



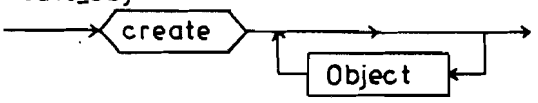
Operation



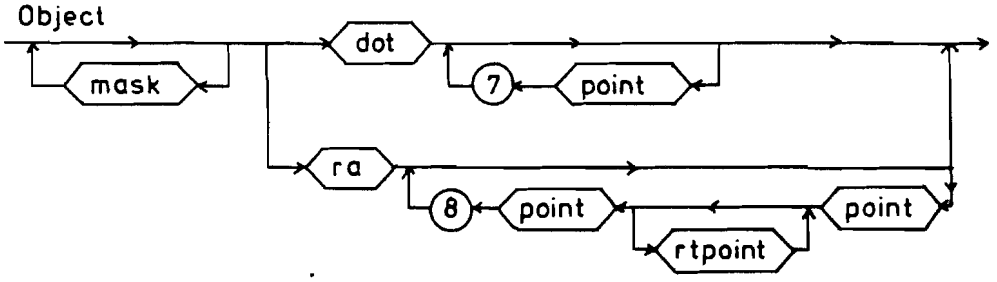
Edit



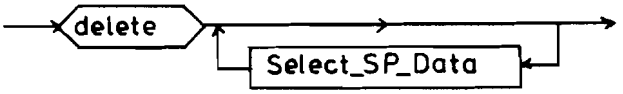
Create_Obj



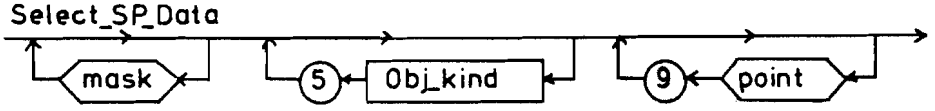
Object



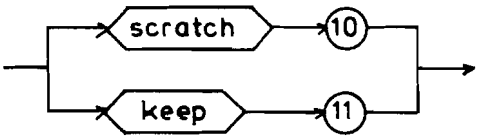
Delete_Obj



Select_SP_Data



Command



The numbers 1 to 11 in the syntax-diagrams relate to the actual implementation of the terminal-generation programs.

Overview of actual procedures:

1. Initialize_Term_Gen

   This procedure initializes the global data and reads a file with the network-description of the layout-module. The inputs and outputs are read and the power-supply names VDD and VSS are added to this list. A comparison with the existing terminal-names is made. Only those masks are sent to the menu-screen, that have not yet a terminal-description. A list of named layout-elements is made for internal book-keeping.

2. Term_Close_And_Stop

   Generates a warning on the menu-screen when there are still some terminal-names left, that have not yet a terminal description.

3. Move_Signals_To_SP

   Moves layout-elements, with the same name as the name of the selected terminal, to the scratchpad. It is displayed and high-lighted automatically.

4. Get_Name

   Reads a terminal-name from key-board in case of a non-existing or non-complete network-description. The name is added to the VIC. A field is created on the menu-screen and is high-lighted.

5. Set_Object_Kind

   Adjusts the internal book-keeping with the actual object-kind. This object-kind is a part of the context for object-selection or object-creation.

6. Search_Object_And_Add_To_SP

   Starts the scanner of the VIC data-structure. Contents- details, that coincidence with a given layout-point are named (the name of the actual terminal) and moved to scratchpad.

7. Add_Point_To_SP

   In the creation mode, a terminal point in a symbolic cell is created, named and moved to scratchpad.

8. Make_RA_And_Add_To_SP

   A rectangle is created as terminal. Its size is displayed real-time. When the second diagonal point is given, the rectangle is named and moved to scratchpad.

9. Delete_SP_Object

   Selected and created objects can also be deleted from scratchpad. A context of object-kind, mask and layout-point must be given.

10. Clear_SP

    Executes the 'scratch' command. The scratcpad is cleared and the internal book-keeping is adjusted.

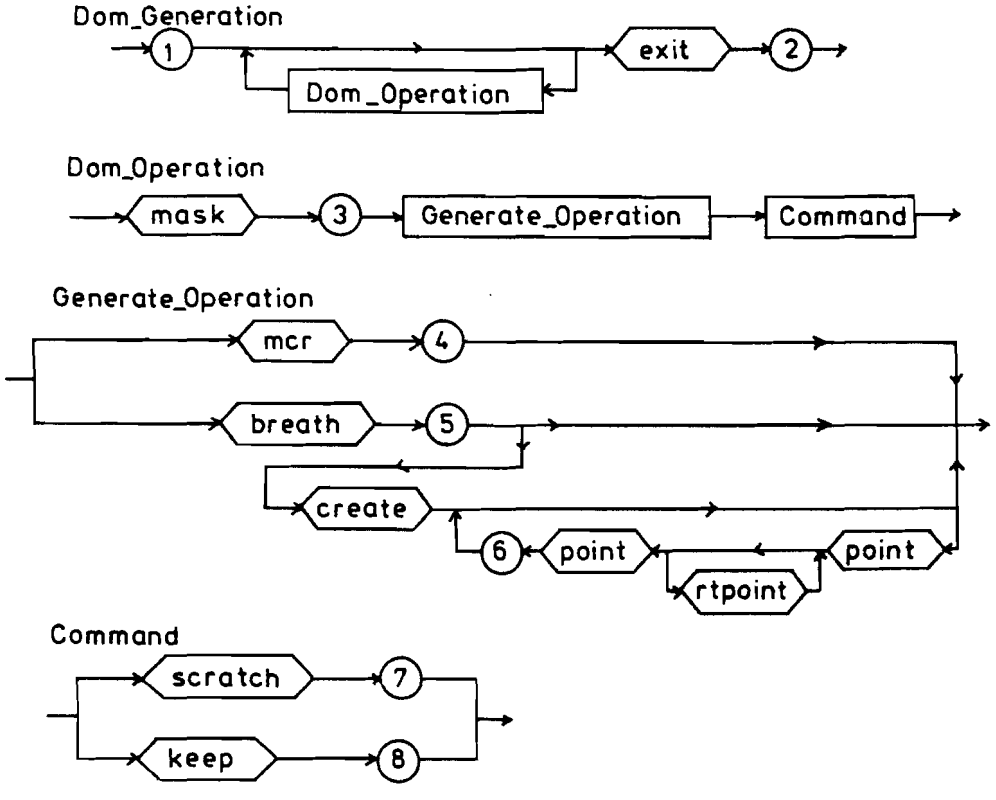
11. Move_Contents_Data_To_Term_Part

    Checks for the number of elements in the scratchpad. A signal-compound (signal ... end;) is generated when there is more than one element. Data, that originally resided in the contents-part is removed from this part. Scratchpad data is moved to the VIC data-structure. The actual terminal-name is removed from the screen.

The syntax definition of the domain generation is defined as:

```
< Dom_Generation >       :== { < Dom_Operation > } EXIT

< Dom_Operation >        :== MASK < Generate_Operation > < Command >

< Generate_Operation >   :== MCR |
                             BREATH [ CREATE { POINT { RTPOINT } POINT } ]

< Command >              :== SCRATCH | KEEP
```

The numbers 1 to 8 in the next syntax diagrams refer to the places in the language, in which the actual domain-generation programs are implemented.



Overview of actual procedures in the domain-generation

1. Initialize_Dom_Gen

   This procedure initializes the global data. It makes a list of masks on the menu-screen, that are present in the contents-part, but NOT in the domain-part of the module. An internal representation of the design-rules is made. Symbols for the domain generation are created on the menu-screen.

2. Dom_Close_And_Stop

   Generates an error message on the screen when there are still some masks, that have not recieved a domain area.

3. Find_Ordinal_Number

   Makes a transformation of the symbol to the internal VIC mask-number.

4. Search_MCR_And_Add_To_SP

   Starts the VIC-scanner, that adjusts the two diagonal edge-points of a minimum circumscribing rectangle. The resulting bounding-box is moved to scratchpad.

5. Breath_And_Add_To_SP

   Starts the same VIC-scanner, but makes now two lists. The first one contains the layout-details and the terminals. The second one contains the domains of sub-modules. It calculates and displays the lower-bound value of the grow factor PHI and asks for an actual grow factor on the terminal. The two lists are grown (with different grow-factors), merged and shrunk (with the same factor). The last operation is done with a standard shrink-algorithm. Data is moved to scratchpad and is high-lighted.

6. Make_RA_And_Add_To_SP

   Displays newly created rectangles on a real time base. When the second diagonal point is given, the rectangle is moved to scratchpad.

7. Clear_SP

   Executes the 'scratch' command. The scratchpad is cleared.

8. Store_SP_To_VIC

   Executes the 'keep' command. The scratchpad data is made permanent by writing the domain data to the VIC. The mask-name is deleted from the menu-screen.

APPENDIX C

EXAMPLE OF A DOMAIN-GENERATION PARSER

This appendix gives an example of the parsing procedures for the domain-generation. They were used for testing purposes with input and output from keyboard-terminal. The actual programs interface with a window-manager program, that takes care of the automatic translation of coordinates on the tablet to layout-coordinates and symbols on the menu-screen. Numbers, placed in brackets, indicate the places in the parser in which the actual procedures are called. The numbers are the same as indicated in the syntax diagrams in the previous appendix.

The next program parses the domain-generation language.

```
PROGRAM Dom_Generation(input,output);

{ This program parses the domain generation language }

TYPE
   symtype = (SCRATCH,KEEP,EXIT,MASK,NCR,BREATH,
              CREATE,POINT,UNDO);
VAR
   sym : symtype;
   PHI : integer;



   PROCEDURE Getsym;

   BEGIN
      write('Give symbol:');
      readln(input,sym)
   END;



   PROCEDURE Command;

   VAR
      oke : boolean;

   BEGIN
      oke := false;
      WHILE not oke DO
      BEGIN
         IF sym = SCRATCH THEN
         BEGIN
            oke := true;
            writeln('Scratch command accepted');
            Clear_SP;   {7}
            Getsym;
         END
         ELSE
            IF sym = KEEP THEN
            BEGIN
               oke := true;
               writeln('Keep command accepted');
               Store_SP_To_VIC;   {8}
               Getsym
            END
            ELSE
            IF sym = UNDO THEN
               oke := true
            ELSE
            BEGIN
               writeln(' Please give SCRATCH or KEEP:');
               Getsym
            END
```

```
   END     {while}
END        {Command };


PROCEDURE Generate_Operation;

VAR
   oke : boolean;

BEGIN
   oke := false;
   WHILE not oke DO
   BEGIN
      IF sym = MCR THEN
      BEGIN
         oke := true;
         writeln('MCR command accepted');
         Search_MCR_And_Add_To_SP;    {4}
         Getsym
      END
      ELSE
         IF sym = BREATH THEN
         BEGIN
            oke := true;
            writeln('BREATH command accepted');
            writeln(' Give GROW factor PHI: ');
            readln(PHI);
            Breath_And_Add_To_SP;    {5}

            writeln('You can give CREATE to add rectangles');
            Getsym;
            IF sym = CREATE THEN
            BEGIN
               Getsym;
               REPEAT
                  Getsym;
                  writeln('Rectangle will be generated');
                  Make_RA_And_Add_To_SP;    {6}
                  Getsym
               UNTIL sym <> POINT
            END
         END
         ELSE
            IF sym = UNDO THEN oke := true
            ELSE
            BEGIN
               writeln(' please give MCR or BREATH');
               Getsym
            END
   END      {while}
END;        {Generate_Operation}


PROCEDURE Dom_Operation;
```

```
   VAR
      oke : boolean;

   BEGIN
      oke := false;
      WHILE not oke DO
      BEGIN
         IF sym = MASK THEN
         BEGIN
            oke := true;
            Find_Ordinal_Number;    {3}
            Getsym;
            Generate_Operation;
            Command
         END
         ELSE
         BEGIN
            writeln('Give MASK to enter domain generation or EXIT');
            Getsym
         END
      END {while}
   END;   {Dom_Operation}



BEGIN
   Initialize_Dom_Cen;    {1}
   Getsym;
   WHILE sym <> EXIT DO
   BEGIN
      IF sym = UNDO THEN
      BEGIN
         writeln(' a UNDO operation could be performed now');
         Getsym
      END
      ELSE
      BEGIN
         Dom_Operation;
         Getsym
      END
   END;
   Dom_Close_And_Stop    {2}
END.  {Dom_Gen}
```

APPENDIX D

EXAMPLE OF A SCANNER OF THE VIC DATA-STRUCTURE

This appendix contains a scanner of the VIC data-structure, that can handle recursively defined layout-modules. The scanner itself is a recursive algorithm, that is used in the generation of domains. Both bounding-boxes and boundary-polygons can be handled by setting the global variable 'breath' to false or true. The scanner is written in pseudo-code. The actual scanner is somewhat larger.

```
PROCEDURE Search_VIC_Data(pntr      :pointer;
                          transform :modifier);

    { pntr      : module begin pointer
      transform : basic transformation vector}

    { Search_VIC_Data uses the following recursive
      calling scheme:
      Search_VIC_Data calls Analyze_VIC_Record and
      this procedure calls either Analyze_VIC_Record,
      Search_VIC_Data or it processes a leaf record.

      This recursive search-procedure uses three boolean variables
      to keep books of which part of the VIC has to be expanded.
      The next diagram gives an overview of which parts have
      to be expanded in a hierarchical description of the layout.
```

part	main module	sub module	sub-sub module	sub-sub-sub module
domain	-	-	*	-
terminals	-	*	*	-
contents	*	*	-	-

```
      The terminal-part of the last sub-module in which a domain
      is found has to be expanded.
      Boolean variables:
      - submodule  : indicates that we are in the main module
                     or in sub... modules.
      - domain     : indicates that a domain in a sub module
                     is found
      - last_module: indicates that no further sub module
```

```
                    has to be expanded.
      The boolean variable breath indicates if a minimum circum-
      scribing rectangle must be made (false) or if a
      grow/merge/shrink operation has to be executed (true) }

PROCEDURE Estimate_MCR(rect : rectangle);

 { This procedure examines the rectangle and adjusts the minimum
   circumscribing rectangle of the module. }

BEGIN END;


PROCEDURE Add_Rectangle_To_List(rect : rectangle);

  { This procedure adds an orthogonal rectangle to a list.
    This list is used for the grow operation on mask-data }

BEGIN END;


PROCEDURE Analyze_VIC_Record(VAR pntr : pointer);

   { This procedure analyzes an actual VIC record }


   PROCEDURE Analyze_Dot(pntr : pointer);

   VAR
      p     : point;
      delta : integer;
      rect  : rectangle;

   BEGIN
      Get_Dot(p);
      IF correct mask THEN
      BEGIN
         delta := width(mask) div 2;
         Grow_Point(p,delta,rect);
         Transform_Point(p);
         IF breath THEN
            Add_Rectangle_To_List(rect)
         ELSE
            Estimate_MCR(rect)
      END
   END;       { Analyze_Dot }



   PROCEDURE Analyze_Rectangle (pntr : pointer);

   VAR
```

```
   rect     : rectangle;
   delta    : integer;


BEGIN
   Get_Rectangle(rect);
   IF correct mask THEN
   BEGIN
      IF slanted rectangle THEN
         Orthogonal_Rectangle(rect);
      IF line piece THEN
      BEGIN
         delta := width(mask) div 2;
         Expand_Line_Piece(rect,delta);
      END;
      Transform_Rectangle(rect);
      IF breath THEN
         Add_Rectangle_To_List(rect)
      ELSE     { MCR }
         Estimate_MCR(rect)
   END
END;        { Analyze_Rectangle }



PROCEDURE Analyze_Polygon (pntr : pointer);

VAR
   deltaw      : integer;
   rect        : rectangle;
   p1,p2       : point;
   pol         : polygon;



BEGIN
   Get_Polygon(pol);
   IF correct mask THEN
   BEGIN
          { test for wire }
          { the hartline is traced and is blown up }
      IF wire THEN
      BEGIN
         deltaw := width(mask) div 2;
         p1     := begin_point;
         p2     := p1;
         WHILE not endpoint DO
         BEGIN
            p1 := p2;
            Transform_Point(p1);
            p2 := next_point;
            Transform_Point(p2);
            Expand_Line_Piece(p1,p2,rect,deltaw);
            IF breath THEN
               Add_Rectangle_To_List(rect)
            ELSE
```

```
                Estimate_MCR(rect);
            END;
        END { wire };

            { test for polygon }
            { the boundary of the polgon is scanned }
        IF polygon THEN
        BEGIN
            p1 := first_boundary_point;
            Transform_Point(p1);
            Find_MCR_Of_Polygon(p1,rect);
            WHILE not endpoint DO
            BEGIN
                p1 := next_point;
                Transform_point(p1);
                Find_MCR_Of_Polygon(p1,rect)
            END;
            IF breath THEN
                Add_Rectangle_To_List(rect)
            ELSE
                Estimate_MCR(rect)
        END  { polygon }
    END
END;        { Analyze_polygon }



PROCEDURE Analyze_Part ( VAR pntr : pointer);


BEGIN
    CASE part OF

        domain_part :
            BEGIN
                IF submodule and not domain THEN
                BEGIN
                    domain := true;
                    WHILE not endpointer DO
                    BEGIN
                        Get_Next_Record(pntr);
                        Analyze_VIC_Record(pntr)
                    END
                END
                ELSE
                    pntr := endpointer;
            END;

        terminal_part :
            BEGIN
                IF submodule and not last_module THEN
                BEGIN
                    IF domain THEN last_module := true;
                    WHILE not endpointer DO
                    BEGIN
```

```
                    Get_Next_Record(pntr);
                    Analyze_VIC_Record(pntr)
                 END
              END
              ELSE
                 pntr := endpointer
           END;

        contents_part or compound_part :
           BEGIN
              IF not last_module THEN
              BEGIN
                 WHILE not endpointer DO
                 BEGIN
                    Get_Next_Record(pntr);
                    Analyze_VIC_Record (pntr)
                 END
              END
              ELSE   { restore global variables }
              BEGIN
                 pntr        := endpointer;
                 domain      := false;
                 last_module := false
              END
           END;

     END; { CASE }
END; { Analyze_Part }



PROCEDURE Analyze_Instance (pntr : pointer);

     VAR
        copysubmodule  : boolean;
        copydomain     : boolean;
        copylastmodule : boolean;
        q              : pointer;
        trans          : modifier;

     BEGIN
        IF not last_module THEN
        BEGIN
              { save global data }
           trans := transform;
           q := pntr;
           Accumulate_Modifier(transform);
           copysubmodule  := submodule;
           copydomain     := domain;
           copylastmodule := last_module;
           submodule := true;

           Search_VIC_Data(q,transform);
                { restore global data }
           submodule   := copysubmodule;
           domain      := copydomain;
```

```
            last_module := copylastmodule;
            transform   := trans;
         END
      END; { Analyze_Instance }



   PROCEDURE Analyze_Array (VAR pntr : pointer);

      VAR
          q         : pointer;
          count     : integer;
          first,
          second    : integer;
          trans     : modifier;
          offset    : point;

      BEGIN
                     { save modifier }
         trans  := transform;
         first  := first_bound;
         second := second_bound;
         offset := array_pitch;
         count  := first;
         q      := array_pointer;
         WHILE count <= second DO
         BEGIN
            pntr := q;
            Analyze_VIC_Record(pntr);
            Accumulate_Modifier(transform);
            count := count + 1
         END;
                    { restore modifier }
         transform := trans
      END { Analyze_Array };



   BEGIN  { Analyze_VIC_Record }

      CASE record_type OF
         dot       : Analyze_Dot       (pntr);
         rectangle : Analyze_Rectangle (pntr);
         polygon   : Analyze_Polygon   (pntr);
         part      : Analyze_Part      (pntr);
         instance  : Analyze_Instance  (pntr);
         array     : Analyze_Array     (pntr);
      END { CASE }

   END { Analyze_VIC_Record };



BEGIN        { Search_VIC_Data }
```

```
   WHILE not endpointer DO
   BEGIN
      Get_Next_Record(pntr);
      Analyze_VIC_Record(pntr)
   END
END;          { Search_VIC_Data }
```

This procedure is initialized and called from a main program by :

```
   domain      := false;
   submodule   := false;
   last_module := false;
   breath      := true       { grow/merge/shrink }
                  or false; { minimum circumscribing rectangle }

   Search_VIC_Data ( pointer_to_module , null_modifier );
```

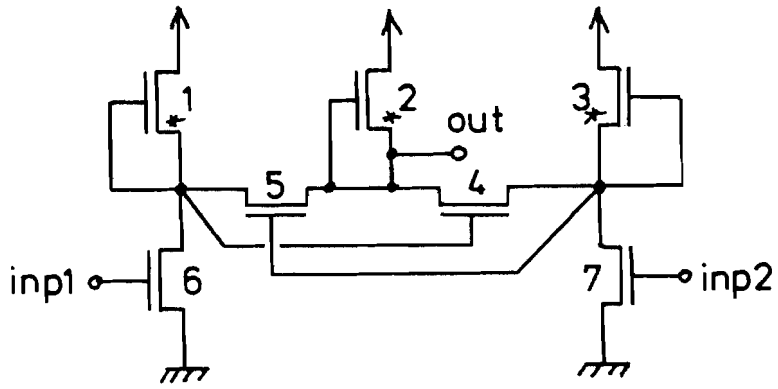
APPENDIX E

SOME EXAMPLES OF DOMAIN- AND TERMINAL-GENERATION

This appendix contains the VLDL description of an NMOS layout-module, called IXOR (Inverting eXclusive OR). Examples of domain-generation before terminal-generation, and terminal-generation before domain-generation will be given for the process mask PS. But first an example of module IXOR in VLDL format without any definition of domains and terminals:

```
pattern ixor;
  begin
    ra(ps; 7, 5; 30, 7.5): inp1;
    poly(ps; 40, 5; 51, 5; 51, 21; 48.5, 21; 48.5, 7.5; 40,
             7.5): inp2;
    ra(ps; 17, 12; 45, 14.5);
    ra(ps; 3, 22.5; 8, 33.5): out;
    ra(ps; 7, 31.5; 13.5, 33.5): out;
    ra(ps; 11.5, 27.5; 13.5, 33.5): out;
    ra(ps; 3, 10; 9, 20);
    ra(ps; 12.5, 11; 14.5, 20);
    ra(ps; 40, 23.5; 49, 29.5);
    ra(ps; 35, 24.5; 41, 26.5);
    wire(ps; w = 2; 9, 19; 17.5, 19; 17.5, 25);
    ra(ps; 16.5, 23.5; 28, 26);
    ra(ps; 25.5, 18.5; 28, 26);
    ra(ps; 25.5, 18.5; 40, 21);
    ra(od; -3, 24.5; 15.5, 30.5);
    ra(od; 9.5, 21.5; 15.5, 30.5);
    ra(od; -3, 12; 15.5, 17);
    ra(od; 10, 10; 15.5, 14);
    ra(od; 10, -0.5; 28, 4);
    poly(od; 42, -0.5; 42, 35; 47, 35; 47, 18; 56.5, 18; 56.5,
            -0.5);
    ra(od; 10, 3; 28, 11);
    ra(od; 20, 8.5; 38, 29.5);
    ra(in; -3.5, 11; 2.5, 37): vdd;
    ra(in; 8.5, 21; 16, 27);
    ra(in; 16, 17.5; 19.5, 24);
    ra(in; 31, 21; 44, 24);
    ra(in; -3.5, 34; 48, 37): vdd;
    ra(in; 9, -1.5; 57.5, 4.5): vss;
    ra(in; 51.5, -1.5; 57.5, 20): vss;
    array(0 .. 1; 0, 15) of
    begin
```

```
        ra(in; 41, 15; 48, 21);
        ra(co; 42.5, 16.5; 46.5, 19.5)
      end;
      ra(in; 19, 15; 25, 21);
      ra(od; 20, 16; 24, 20);
      ra(co; 20.5, 16.5; 23.5, 19.5);
      ra(in; 28.5, 21.5; 34.5, 27.5);
      ra(od; 29.5, 22.5; 33.5, 26.5);
      ra(co; 30, 23; 33, 26);
      ra(co; -2, 12.5; 1, 16.5);
      ra(co; -2, 25; 1, 30);
      ra(co; 10, 22.5; 14.5, 25.5);
      ra(co; 11, 0; 27, 3);
      ra(co; 43, 0; 52, 3);
      ra(co; 53, 7; 56, 17);
      ra(cs; 11, 9.5; 16.5, 18.5);
      ra(cs; 10, 26; 15, 32);
      ra(cs; 33.5, 23; 39.5, 28);
      ra(cs; 40.5, 10.5; 46.5, 16);
      ra(di; 1, 10.5; 11, 32);
      ra(di; 40.5, 21.5; 48.5, 31.5)
    end;
    comment 'ixor'
  end;
```

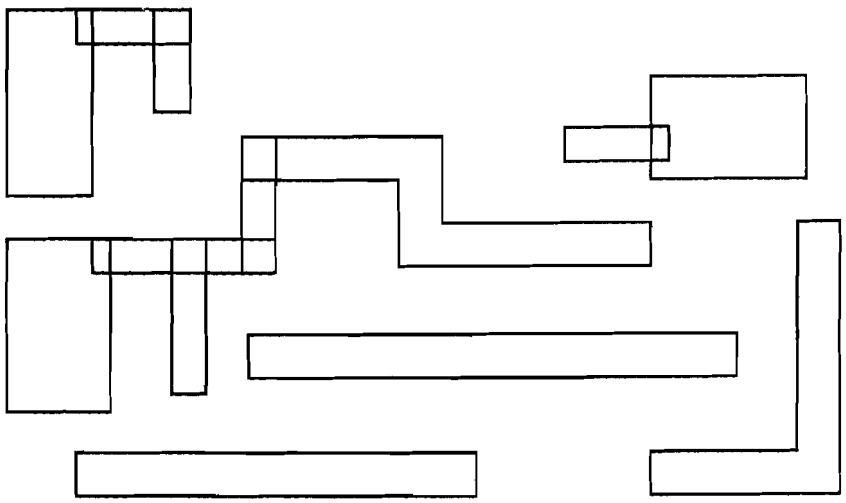


network graph of IXOR

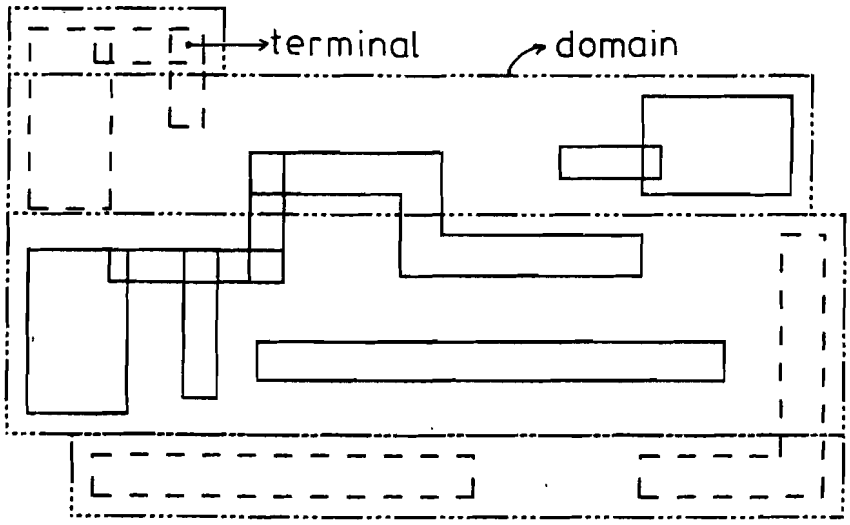
A network description of the module in NDL is:

```
        macro

macro   ixor   i(inp1,inp2)  o(out)
out     flog   i(inp1#inp2)
        mend
```

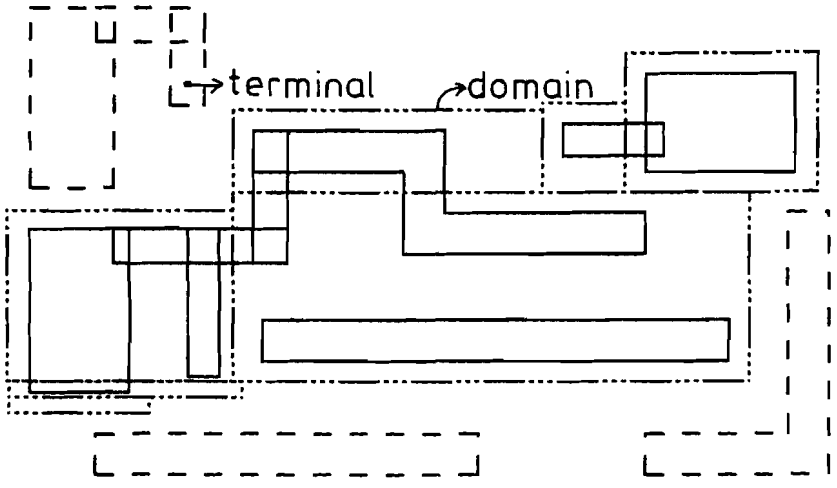
A layout of this module is :



The next example is the result of the generation of domains before the assignment of terminals in the PS mask of module IXOR.



The next example is the result of opposite operation: terminals are assigned before the generation of domains.



The next listing gives an example of result of the generation of domains before the assignment of terminals.

```
pattern ixor;
  domain
    ra(di; 0, 9.5; 49.5, 33);
    ra(cs; 9, 8.5; 47.5, 33);
    ra(co; -3, -1; 57, 35.5);
    ra(in; -4.9, 9.6; 58.9, 21.4);
    ra(in; 7.6, -2.9; 58.9, 9.6);
```

```
    ra(in; -4.9, 36.6; 49.4, 38.4);
    ra(in; -4.9, 21.4; 49.4, 36.6);
    ra(od; 7.85, -2.65; 58.65, 20.15);
    ra(od; -5.15, 9.85; 7.85, 20.15);
    ra(od; 39.85, 20.15; 49.15, 37.15);
    ra(od; -5.15, 20.15; 39.85, 31.65);
    ra(od; -5.15, 31.65; 17.65, 32.65);
    ra(ps; 1.75, 8.75; 52.25, 22.25);
    ra(ps; 5.75, 3.75; 52.25, 8.75);
    ra(ps; 38.75, 22.25; 50.25, 30.75);
    ra(ps; 33.75, 22.25; 38.75, 27.75);
    ra(ps; 1.75, 22.25; 33.75, 27.25);
    ra(ps; 1.75, 27.25; 14.75, 34.75)
  end;
  terminals
    ra(ps; 7, 5; 30, 7.5): inp1;
    signal
      ra(in; 51.5, -1.5; 57.5, 20);
      ra(in; 9, -1.5; 57.5, 4.5)
    end: vss;
    signal
      ra(in; -3.5, 34; 48, 37);
      ra(in; -3.5, 11; 2.5, 37)
    end: vdd;
    signal
      ra(ps; 11.5, 27.5; 13.5, 33.5);
      ra(ps; 7, 31.5; 13.5, 33.5);
      ra(ps; 3, 22.5; 8, 33.5)
    end: out;
    poly(ps; 40, 5; 51, 5; 51, 21; 48.5, 21; 48.5, 7.5; 40,
             7.5): inp2
  end;
  drawing
    ra(bb; -3.5, -1.5; 57.5, 37);
    text 'IXOR'(dr; -3.5, -1.5; size =  14)
  end;
  begin
    ra(ps; 17, 12; 45, 14.5);
    ra(ps; 3, 10; 9, 20);
    ra(ps; 12.5, 11; 14.5, 20);
    ra(ps; 40, 23.5; 49, 29.5);
    ra(ps; 35, 24.5; 41, 26.5);
    wire(ps; w = 2; 9, 19; 17.5, 19; 17.5, 25);
    ra(ps; 16.5, 23.5; 28, 26);
    ra(ps; 25.5, 18.5; 28, 26);
    ra(ps; 25.5, 18.5; 40, 21);
    ra(od; -3, 24.5; 15.5, 30.5);
    ra(od; 9.5, 21.5; 15.5, 30.5);
    ra(od; -3, 12; 15.5, 17);
    ra(od; 10, 10; 15.5, 14);
    ra(od; 10, -0.5; 28, 4);
    poly(od; 42, -0.5; 42, 35; 47, 35; 47, 18; 56.5, 18; 56.5,
            -0.5);
    ra(od; 10, 3; 28, 11);
    ra(od; 20, 8.5; 38, 29.5);
    ra(in; 8.5, 21; 16, 27);
    ra(in; 16, 17.5; 19.5, 24);
```

```
      ra(in; 31, 21; 44, 24);
      array(0 .. 1; 0, 15) of
      begin
        ra(in; 41, 15; 48, 21);
        ra(co; 42.5, 16.5; 46.5, 19.5)
      end;
      ra(in; 19, 15; 25, 21);
      ra(od; 20, 16; 24, 20);
      ra(co; 20.5, 16.5; 23.5, 19.5);
      ra(in; 28.5, 21.5; 34.5, 27.5);
      ra(od; 29.5, 22.5; 33.5, 26.5);
      ra(co; 30, 23; 33, 26);
      ra(co; -2, 12.5; 1, 16.5);
      ra(co; -2, 25; 1, 30);
      ra(co; 10, 22.5; 14.5, 25.5);
      ra(co; 11, 0; 27, 3);
      ra(co; 43, 0; 52, 3);
      ra(co; 53, 7; 56, 17);
      ra(cs; 11, 9.5; 16.5, 18.5);
      ra(cs; 10, 26; 15, 32);
      ra(cs; 33.5, 23; 39.5, 28);
      ra(cs; 40.5, 10.5; 46.5, 16);
      ra(di; 1, 10.5; 11, 32);
      ra(di; 40.5, 21.5; 48.5, 31.5)
    end;
    comment 'ixor'
  end;
```

APPENDIX F

EXPLANATION OF ABBREVIATIONS

VOILA : VLSI Oriented Interactive Layout Aid.
Name of the Philips project for the creation of a new VLSI design environment.

VIC : VOILA Internal Code. The one-to-one translation of the textual layout-description to a binary data-format, that is kept in core.

VLDL : VOILA Layout Definition Language. Defines the textual constructs for the description of a VLSI layout.

INIF : Inverted Network Interface Format. Contains the net-oriented description, in binary format, of the network-description of a layout-module.

NDL : Network Description Language. Defines the textual constructs for the logical description of a module, e.g. SIMON [SIM 84].

CMSK : Circuitmask. Name of the existing layout description and manipulation program.

JOJO : Name of the program for the interactive generation of domains and terminals.

INDEX OF KEYWORDS