The Integrated Circuit Design Book

P. Dewilde

Editor

Delft University Press
The Integrated Circuit Design Book

Papers on VLSI Design Methodology from the ICD-NELSIS Project

P. Dewilde

Editor

Delft University Press/1986
PREFACE

The Pre-edition of the ICD book has been prepared at the occasion of the general meeting of the ICD project held in Delft in January 1986. The purpose of it is to present interested readers with a reasonably general and readable overview of what is quickly becoming the "Integrated ICD system". It represents the work of about 100 engineers in various capacities: researchers, developers, software developers, designers. The system contains many original discoveries, new methods, unorthodox ways of doing things. Aside from the development of original tools, the transfer to commercial products has been foreseen: within the ICD project the responsibility to move ideas to products is organised in a unique way - one of the partners is a software house whose main responsibility is the coordination of efforts and the production of a viable commercial system. The main purpose of the book is to provide interested people and users with an introduction to the system’s operational principles and theory. Many parts of the system will not be covered here, namely those parts that are purely operational or classical: users’ interfaces, manuals and of course sources.

The ICD system is a very large system: it contains more than twenty megabyte of source code. That in itself is not a quality: the sheer vastness of a system may make it impractically difficult to master or understand. In our case that is not so. Except for some simulators, the system consists of a large collection of rather small functions each with their own capabilities, and which interface in a local way on logically structured data. The system is vast because it contains many functions, not because it is a massive whole. The main themes in ordering the data and mastering the complexity are: hierarchy and multiview. The hierarchy-multiview matrix of datafiles contains the work of the designers in a logically organized way so that each - rather small - design file is accessible at any level of the hierarchy and in any view, to see, manipulate, modify and use. In each view tools are aware of the structure of the data and can make use of it: neighboring points in the matrix can be converted into each other, either by verification tools, which typically will produce a higher view from a lower one or with synthesis tools which will automatically generate a lower view from a higher one.

The many tools presented contain as many sophisticated algorithms. The main reason the partners have embarked on this project, is to have the opportunity to realize many of their novel ideas as a total system, ready for distribution and use, yet supporting truly hierarchical multiview VLSI design. Highlights are:

- an original dataschema incorporating hierarchy, views, versions and a multiuser environment based on a semantic datamodel.
- a novel method for verification and extraction of layouts based on the original stateruler scan technique.
- a new parametrized design language.
- a design-rule compiler.
- an original concept of mixed-mode simulation (the Piecewise Linear Simulator).
- new tools for cell-generation and chip assembly achieving designs which are correct by construction.
- the development of a workstation architectures suited for VLSI design.
- a powerful tool for silicon-compilation based on the isomorphism between an algebra describing placement and the functional algebra representing systems’ behaviour.
It has been possible to realize all that in a relative short time due to a number of external and internal factors. As external factors the support of both the Commission of the EEC (Microelectronics program MR09) and the Dutch Government (NELSIS program) must be mentioned. Thanks to this support, and the own resources of the participating universities and companies, enough manpower and volume could be mustered to achieve the critical mass needed. As internal factors I would like to mention a number of important factors. The first one is the fact that all partners succeeded in attracting personnel of very high quality to work on the various tasks. In the light of the many original contributions that came out of the project, the many publications, and the high quality of most of the software, one can only marvel how it turned out to be possible to assemble so much enthusiastic talent. The second internal factor has been the systematic usage of good system standards. First of all there has been the early choice for UNIX as portable Operating System. Although this would be considered nowadays as a trivial choice, it cannot be sufficiently emphasized how important it turned out to be and how much it contributed to the overall success, part of the reason being that the interest of efficiency the multiview hierarchical dataschema has to be mapped on the file system. Our analysis that a healthy design system should be build around a portable hierarchical filemanagement system with a hierarchical process structure has now been vindicated. Aside from its portability, UNIX contributed so many high quality software tools which in many instances greatly reduced software development efforts and increased support and orderliness thereby saving many manmonths of development time. Other standards are now gradually - through experience - taking shape. Several internal dataformats have been defined. GKS is used as rallying graphics standard, although many alternative, device oriented systems coexist (in its present versions it is rather slow, but for many applications fast enough, and its value as a documentation standard cannot be underestimated - where high speed is necessary one must resort to dedicated solutions, which often can be build with the GKS definition as starting point). A standard for object-oriented programming is in the makings. A third internal factor has been the smooth management of the project and the good understanding between the partners. A key factor here is the avoidance of time consuming and unnecessary bagling over details. In my opinion, the management both of ICD and NELSIS has been exemplary with few time lost in unnecessary meetings, but a very matter-of-fact attitude to the solution of essential problems whenever they arise.

The end result is here for all to see. We have aimed at a book that at one hand would be reasonably complete as far as information is concerned, and at the other hand be easily readable. There was not enough time to make all the contributions uniform in style. Moreover, the first phase of the ICD project is barely ending, and we do not have the pretentian to present the reader and user with a fixed-for-all-time system. Although a certain completeness emerges at the present time, even putting this work together has raised new interesting questions, suggestions for modification etc... ICD is like a living organism and as such it evolves, hopefully improving all the time. Because of the systems' novelty, many of its components are still in various test phases. For these reasons, the present ICD edition has been called "Preliminary".

The book is subdivided in eight chapters corresponding to the main activities in the project, which is not the same as the subdivision in tasks of the ICD project.

Chapter 1 treats questions related to the systems architecture: dataschemas, designlanguages, system standards. There are contributions of Delft University of Technology on the hierarchical multiview dataschema with version control in a multiuser environment and on a very powerfull procedural layoutdesign language called LDMC which is embedded in the C programming language. Furthermore, there is the definition of the ICD network database standard, the design rule compiler of ICS, and the "Eindhoven schematics editor" which is a powerful schematics editor to be used in conjunction with the mixed mode PWL simulator.

Chapter 2 discusses the low-level verification traject, which was entirely developed at Delft: going from hierarchical DRC, over hierarchical layout-to-circuit extraction to
electrical circuit verification - in this case switchlevel-timing. This part contains major
originalities: one can say that each program is unique in the way it operates. Recent
publications have given brief descriptions of the main principles. The papers in this book
allow for a more leisure acquaintance with the novel ideas.

Chapter 3 presents some novel discrete synthesis tools. There is the very fast new
placement and routing system for macrocells developed in Delft, and which has already
been installed in many places. Its interactive mode has proven very popular and it is a
spectacular sight to view a complete complex routing in seconds on the graphics display.
MoDG is the newly developed cell generator of ICS which also proves to be a very popular
tool in the system. Although sophisticated synthesis systems are presented in later
chapter, these direct tools give immediate support to the main problems faced by the
designer.

Chapter 4 is devoted entirely to the mixed mode Piecewise Linear Simulator. We claim to
possess the only truly general mixed mode simulation system presently available. Its
properties are also truly spectacular: the same simulator will give you either a detailed
circuit simulation with precise transistor models, or a rough timing simulation, or a
combination of both, with a speed that is very much a function of the complexity of the
circuit. It has been developed over the years at Eindhoven University of Technology with
a very concentrated efforts in the last two years. The Eindhoven group can be credited
with a truly unique piece of work!

Chapter 5 consists of a number of papers describing the ASTRA system for symbolic
design. The main goal here is the top-down generation of a design - especially for signal
processing applications - which is "correct by construction", supported by interactive tools
which generate all the relevant information (floorplans, circuits and layouts)
simultaneously, thereby avoiding the necessity to use costly verification tools. The
ASTRA system has been duly tried out on new designs where its power has been
demonstrated. It was developed at British Telecom, and forms one of the main global
synthesis tools of the project.

Another, high level synthesis tool was developed by Twente University of Technology and
is called MoDL. It is described in Chapter 6. It is based on the fundamental idea that an
isomorphism can be defined between the algebra defining the behaviour of a system, and
the algebra describing its geometrical synthesis. This functional way of looking at the
design situation leads to a new approach to top-down design and a very user-friendly
support of his silicon-compilation problem. It was originally written in LISP, but for
purposes of portability, a fast C-version has been developed, which turns out to be a
LISP-like system in its own right.

Chapter 7 is entirely devoted to the Gate-Array system developed by the Technical
University of Eindhoven. Here also, we present very original software. One of the main
requirements that the system had to satisfy was its "openness". This means that the
design-rules and properties of any commercial gate-array should be capable of support by
the system without redevelopment. Since todays' gatearrays often have multiple layers of
interconnect with rules dependent on the relation between the layers (due e.g. to vias), a
new method had to be conceived to interface rules and routing.

Last but not least, there was a major effort by PCS, documented in chapter 8, to define
what would now be the best solution for the designation of the future. It should have
the following components: a good solution for the problem of fast multiwindowed
graphics (high or low resolution), and an appealing users' interface preferably build
around the concept of "objects", derived from SMALLTALK, and also successfully
implemented in the Macintosh.

What will the future bring? The Commission of the EEC has agreed to a continuation of
the project under the Esprit program. New goals have been defined, and we expect to
achieve an order of magnitude improvement on what we now have. The NELSIS program
also will run for two more years, supporting both the activities under ICD and additional
activities especially in the Silicon Compilation area. If things go as planned, we shall be
able to present interested users with the definitive ICD book two or three years from now
containing what we hope to be the ultimate in the art of VLSI design techniques and
methodology.
A commercial version of some parts of the system is already available from ICS, and has been successfully demonstrated at the CE show in London in the fall of 1985. All partners have local versions which may differ from each other (they are experimental) but which all enjoy SCCS support. We have agreed to a free distribution to universities and non-profit organisations (except for duplication costs). Interested parties may contact the ICD manager - P. van Es - at ICS for information.

Aside from various programs and products, the benefits of the present project are in the field of manpower development and education. We have adopted a model whereby most of the conception of the system is done by young Ph.D. students, who not only generate new ideas, but also are very eager to introduce new and more powerful software development techniques. They are the motor behind the innovations process. The longterm mechanism whereby new technologies are introduced in our countries, which leads to new product development, and ultimately to new companies and an increase in economic activity goes via the top - Ph.D. level - of our educational institutions. I believe that this component has largely been neglected in the past in Europe leading over the many years to an acute shortage of capable people at the top of our enterprises (aside from weakening and degrading the institutions to mere massive student factories with no scientific nor technological content). This undesirable situation has a tendency to perpetuate itself, due to a climate which is unfavourable to universities and the sometimes appalling decision making process in our companies. I hope that the present project, in its own local ways, will have contributed to the alleviation of the problem by providing our communities with a collection of about hundred top level engineers experienced and creative in one of the major contemporary technologies.

P. Dewilde
Editor.
# Table of Contents

1. Systems Architecture, Design Languages and Base System Components

1.1 Data Management for Hierarchical and Multiview VLSI Design


   1.1.1 Introduction
   1.1.2 General Database Concepts
   1.1.3 The Design Environment
   1.1.4 The Version Concept
   1.1.5 The VLSI Dataschema
   1.1.6 Implementation
   1.1.7 Concluding Remarks
   1.1.8 References

1.2 Procedural Layout Generation and Silicon Assembly in C

   N. van der Meijs

   1.2.1 Introduction
   1.2.2 Overview of the ldmc System
   1.2.3 The ldmc Library
   1.2.4 An Illustrative Example
   1.2.5 Discussion
   1.2.6 References

1.3 Proposal for a Network Description Format in the ICD-System

   A.C. de Graaf and G. Janssen

   1.3.1 Introduction
   1.3.2 Fundamentals of Circuit Description
   1.3.3 Data-Base Aspects
   1.3.4 General File Format
   1.3.5 Interface File Format
1.3.6 Network Call File Format
1.3.7 Network Wire File Format
1.3.8 Network Behaviour File Format
1.3.9 Implementation

1.4 The Eindhoven Schematic Editor
   A. Lodder, M. van Stiphout and J.T.J. van Eijndhoven

1.5 Designer-Foundry Interface
   F.D. Smedes, P.B. Duin, W. Schouten, Chr. van Veenendaal, and L. Spaanenburg
   1.5.1 Introduction
   1.5.2 The Receiving End
   1.5.3 Lambda Based Designrules
   1.5.4 The Role of Fabricator Information
   1.5.5 The Giving End
      Appendix 1
      Appendix 2

2. Low Level Verification

2.1 An Hierarchical and Technology Independent Design Rule Checker
   T.G.R. van Leuken and J. Liedorp
   2.1.1 Introduction
   2.1.2 Augmented Instancing of a Cell
   2.1.3 Line Segment Conversion, the Statruler
   2.1.4 Design Rule Checking
   2.1.5 Results
   2.1.6 Conclusions
   2.1.7 References
   2.1.8 Appendix A: Implementation of Technology
   2.1.9 Appendix B: The Program nbool
   2.1.10 Appendix C: The Program dimcheck
2.1.11 Appendix D: The Program dubcheck

2.2 A Technology Independent Hierarchical Layout to Circuit Extractor Using a Finite State Approach

F. de Lange and J. Annevelink

2.2.1 Introduction
2.2.2 Principle of Extract Operation
2.2.3 Technology Description
2.2.4 The Element Recognition Algorithm
2.2.5 Construction of the Network Data Structure
2.2.6 Implementation Results
2.2.7 Conclusions and Suggestions

Appendix A
Appendix B
Appendix C

2.3 SLS: A Switch-Level Timing Simulator

A.J. van Genderen and A.C. de Graaf

2.3.1 Introduction
2.3.2 The Principle of the Simulation
2.3.3 Stable Voltage Determination
2.3.4 Dynamical Behavior Determination

2.4 SLS: Users Manual

A.J. van Genderen and A.C. de Graaf

2.4.1 Introduction
2.4.2 Description of the SLS-Package
2.4.3 The Description of Networks
2.4.4 Simulation and Simulation Commands

3. Discrete Synthesis Tools

3.1 Automatic Routing in a General Cell Environment

H. Cai and P. Dewilde
3.1.1 Introduction
3.1.2 System Overview and its Properties with Respect to Others
3.1.3 The Construction of Channels
3.1.4 Global Routing
3.1.5 Power Supply Net Routing
3.1.6 Preparation of the Channel Routing
3.1.7 Channel Routing
3.1.8 Relative Position of Blocks
3.1.9 Concluding Remarks

3.2 Parametrised Building Blocks
G. Kleissen, M. Roelofs, Chr. van Veenendaal and L. Spaanenburg
3.2.1 Introduction
3.2.2 Silicon Compilation
3.2.3 Silicon Assembly: Generating Layout from a Floorplan
3.2.4 MoDG a Module Assembler
3.2.5 Discussion
Appendix 1 - Manual MoDg
Appendix 2 - MoDg Syntax
Appendix 3 - An Example of a MMU Floorplan
Appendix 4 - Floorplan Description of PLA

4. The Piecewise Linear Simulator
4.1 The Piecewise Linear Simulator
J.T.J. van Eijndhoven
4.1.1 Introduction
4.1.2 Piecewise Linear Systems
4.1.3 The Linear Complementarity Problem
4.1.4 Hierarchical Analysis
4.1.5 Implementation Aspects
4.1.6 Model Examples
4.1.7 Conclusion
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.8 Menus</td>
<td>5.15</td>
</tr>
<tr>
<td>5.2.9 Conclusions</td>
<td>5.16</td>
</tr>
<tr>
<td>5.3 The Floor Plan Editor</td>
<td>5.18</td>
</tr>
<tr>
<td>D.A. Orrey and N.H. Weiner</td>
<td></td>
</tr>
<tr>
<td>5.3.1 Introduction</td>
<td>5.18</td>
</tr>
<tr>
<td>5.3.2 The Primary Editing Functions</td>
<td>5.20</td>
</tr>
<tr>
<td>5.3.3 The Visible World</td>
<td>5.25</td>
</tr>
<tr>
<td>5.3.4 Inserting Data</td>
<td>5.27</td>
</tr>
<tr>
<td>5.3.5 Moving Around the Hierarchy</td>
<td>5.28</td>
</tr>
<tr>
<td>5.3.6 Placement of a Module</td>
<td>5.28</td>
</tr>
<tr>
<td>5.3.7 Selecting and Deleting Data</td>
<td>5.28</td>
</tr>
<tr>
<td>5.3.8 Other Edit Commands</td>
<td>5.29</td>
</tr>
<tr>
<td>5.3.9 Loading, Writing and Quitting</td>
<td>5.29</td>
</tr>
<tr>
<td>5.3.10 Additional Features</td>
<td>5.29</td>
</tr>
<tr>
<td>5.3.11 The Data Structure</td>
<td>5.30</td>
</tr>
<tr>
<td>5.3.12 Starting a New Floor Plan Description</td>
<td>5.33</td>
</tr>
<tr>
<td>5.3.13 The Editor Tally List</td>
<td>5.35</td>
</tr>
<tr>
<td>5.3.14 Topological Position of the Cursor</td>
<td>5.35</td>
</tr>
<tr>
<td>5.3.15 Conclusions</td>
<td>5.37</td>
</tr>
<tr>
<td>5.4 The Symbolic Leaf Cell Editor</td>
<td>5.38</td>
</tr>
<tr>
<td>J.W. Cheshire and D.A. Orrey</td>
<td></td>
</tr>
<tr>
<td>5.4.1 Introduction</td>
<td>5.38</td>
</tr>
<tr>
<td>5.4.2 User Interface</td>
<td>5.39</td>
</tr>
<tr>
<td>5.4.3 The Data Structure</td>
<td>5.41</td>
</tr>
<tr>
<td>5.4.4 Inserting Data</td>
<td>5.41</td>
</tr>
<tr>
<td>5.4.5 Selecting and Deleting Data</td>
<td>5.45</td>
</tr>
<tr>
<td>5.4.6 Other Edit Commands</td>
<td>5.46</td>
</tr>
<tr>
<td>5.4.7 Undoing Edits</td>
<td>5.47</td>
</tr>
<tr>
<td>5.4.8 Duplicating Data</td>
<td>5.47</td>
</tr>
<tr>
<td>5.4.9 Reading and Writing Data</td>
<td>5.48</td>
</tr>
<tr>
<td>5.4.10 Additional Features</td>
<td>5.50</td>
</tr>
<tr>
<td>5.4.11 Conclusions</td>
<td></td>
</tr>
</tbody>
</table>

xii
5.5 The Interactive Leaf Cell Checker

R. Weeks

5.5.1 Introduction
5.5.2 Facilities
5.5.3 The Construct Checker
5.5.4 The Well Checker
5.5.5 The Spanner
5.5.6 The Circuit Checker
5.5.7 Error Reporting
5.5.8 Other Applications of the Circuit Checker
5.5.9 Internal Operation
5.5.10 Conclusions

Appendix I - Fragments of a Typical Error Profile File
Appendix II - Fragments of a Typical Dangling Error File
Appendix III - A Typical Error Message File

5.6 The Leaf Cell Spacer

R.T. Scarfe

5.6.1 Introduction
5.6.2 Circuit Conversion
5.6.3 Circuit Constraints
5.6.4 Imposing Constraints
5.6.5 Enforcing Constraints
5.6.6 Stretch Lines
5.6.7 Significant Edge Generation
5.6.8 Conclusions

5.7 The Chip Assembler

P.R. Benyon

5.7.1 Introduction
5.7.2 Input Considerations
5.7.3 Process Considerations
5.7.4 Output Considerations
5.7.5 Conclusions

xiii
5.8 The Automatic Route to Simulate in the ASTRA System
   R. Weeks

5.8.1 Introduction
5.8.2 Constituent Parts
5.8.3 The Leaf Cell Circuit Extractor
5.8.4 The Leaf Cell Capacitance Extractor
5.8.5 The Structure Extractor
5.8.6 The Translation Phase Proper
5.8.7 Automatic Control of Translation
5.8.8 Conclusions
   Appendix I - An Example of Leaf Cell Circuit Extraction
   Appendix II - An Example of Leaf Cell Capacitance Extraction
   Appendix III - An Example of Structure Extraction
   Appendix IV - An Example of Hitest Output
   Appendix V - An Example of Spice Output

5.9 The Mask Data Generator
   P.R. Benyon and R.T. Scarfe

5.9.1 Introduction
5.9.2 Methodology
5.9.3 Input Considerations
5.9.4 Process Considerations
5.9.5 Output Considerations
5.9.6 Conclusions
   Appendix I - Informal Definition of MGD

5.10 The Logical Naming Strategy
   R.T. Scarfe

5.10.1 Introduction
5.10.2 The Standard Strategy for Data Linkage and Storage
5.10.3 The Logical Naming Strategy
5.10.4 Conclusions
5.11 STAR - a VLSI Architecture for Signal Processing

D.J. Myers and P.A. Ivey

5.11.1 Introduction
5.11.2 Architecture
5.11.3 Design of the STAR
5.11.4 Implementation
5.11.5 Conclusions

6. Functional Design

Definition of the Syntax and Semantics of the Modelling and Design Language, MoDL.


6.1 Semantic Definition of the Modelling and Design Language MoDL

6.1.1 General Introduction
6.1.2 Introduction to MoDL
6.1.3 Variables, Execution and Communication
6.1.4 Functions
6.1.5 Models
6.1.6 Semantics of Communication
6.1.7 Semantics of Synchronous Design
6.1.8 Locality of Interconnect
6.1.9 Layout View
6.1.10 Behavior of Arithmetic Functions
6.1.11 Automatic Transformations for Interconnect
6.1.12 Automatic Transformations for Communication

Appendix 1-A: Syntax Diagrams of MoDL.
6.2.4 Extraction of Logical Values
6.2.5 Operations on Names
6.2.6 Vector Operations
6.2.7 Arithmetic Operations
6.2.8 IO Functions
6.2.9 Program Execution Primitives

6.3 The Parallel MoDL Hardware Description Language
6.3.1 The Hardware Description Language
6.3.2 Model Definition and Syntax
6.3.3 Execution of Parallel Algorithms
6.3.4 Support Functions
6.3.5 Additional Primitive Functions
6.3.6 Layout Construction by Abutment
6.3.7 Timing Analysis

6.4 The MoDL to C Compiler
6.4.1 The MoDL-to-C Compiler
6.4.2 Calling the Compiler
6.4.3 Compiler Directives

6.5 An Alphabetical Cross Reference List
6.5.1 Index for the Parts II, III and IV
6.6 MoDL Case Study: A Single Chip Digital Signal Processor
6.6.1 Introduction
6.6.2 The Processor's Architecture
6.6.3 Some Submodels
6.6.4 The Presentation of Results
6.6.5 Remarks and Conclusions

7. Gate Array Design
7.1 The Architecture of an Open Design System for Gate Arrays

J.A.G. Jess
7.4.1 Introduction
7.4.2 Hierarchical Wire Routing
7.4.3 2*N Wiring Problem, Linear Programming Approach
7.4.4 2*N Wiring Problem, Dynamic Programming Approach
7.4.5 Grid Definition
7.4.6 Evaluation and Recommendations
7.4.7 References

7.5 Local Routing
A.G.J. Slenter
7.5.1 Introduction
7.5.2 Routing in Sub-Areas
7.5.3 Problem Definition
7.5.4 The Routing Landscape
7.5.5 The Routing Primitives
7.5.6 Choice of the Routing Algorithm
7.5.7 Introduction
7.5.8 The Original Lee-Router
7.5.9 Backtracing Techniques
7.5.10 Net Ordering
7.5.11 Multi-Terminal Nets
7.5.12 Reducing the Size of the Search
7.5.13 Conclusions
7.5.14 References

8. Workstations
8.1 The CADMUS 9230 ICD Graphic Workstation 1
   D. Larsson and Dr. P. Schinner et al.
8.1.1 Abstracts
8.1.2 Communication Requirements
8.1.3 Description of Hardware Components
8.1.4 Description of Software Components
8.1.5 Communication Procedure
8.1.6 Interactions between Software Components
8.1.7 Portability and Compatibility
8.1.8 Error Detection and Recovery
8.1.9 Conclusions, Evaluation, Notes

8.2 Notes on the PCS Color Graphic Controller - CGC-4
   Dr. K. Lueddeke and S. Pope

8.3 The PCS Graphics Software World 1985
   R.H. Cole
8.3.0 What do we need?
8.3.1 Our Proposal
8.3.2 Basic Graphics Library
8.3.3 PCSMac Libraries
8.3.4 Meta Graphics Libraries
8.3.5 BGL, PCSMac, and Meta Graphics Implementation for B&W and Colour
8.3.6 Implementation steps
LIST OF PLATES

Chapter 1.4 - Plate 1
An edit of a filter circuit, with a view into one instance, showing the opamp circuit.

Chapter 1.4 - Plate 2
Zooming in automatically activates more details, such as instance names and terminal names. The context window (left bottom) shows the position of the window, relative to the instances. The grid is optionally shown.

Chapter 1.4 - Plate 3
An edit in the representation area of an opamp circuit. Besides the symbol itself, formal parameters can be edited.

Chapter 1.4 - Plate 4
A mixed analog/digital circuit is edited. Blue denotes digital, green denotes analog. These two types cannot be connected together directly, however, components can define relations between the two types of terminals.

Chapter 1.4 - Plate 5
The use of busses, bus splitting, and interconnection by abutment. Furthermore the verify function is shown, picking out the errors in the circuit. Here the errors are: a loop in a wire, an open end of a wire, and an unknown formal parameter in the expression of an actual parameter.

Chapter 2.2 - Plate 1
Layout view of the random counter network used as an example for the hierarchical extractor.

Chapter 3.1 - Plate 1
Routing result example: chip 1

Chapter 5 - Plate 1
ASTRA System: Four views of one cell; stick diagram, mask data, log diagram, floor plan placement.

Chapter 5 - Plate 2
ASTRA System: Floor plan and plot to show where stretching occurred during chip assembly.

Chapter 5 - Plate 3
ASTRA System: Stick diagram of cell showing on-line error checks.

Chapter 5 - Plate 4
ASTRA System: Spaced cell showing constraint information.

Chapter 5 - Plate 5

XX
Two chips designed using the ASTRA System:
(a) STAR  (b) METEORITE

Chapter 7 - Plate 1
Gate Array Placement.

Chapter 7 - Plate 2
Example of rout in CMOS array with one metal layer, poly underpaths and fixed vias.

Chapter 7 - Plate 3
Example of rout in 12L array with two metal layers and programmable vias.
LIST OF AUTHORS

Annevelink, J. 2.51
Benyon, P.R. 5.73; 5.110
Bu, J.C. 7.8
Cai, H. 3.1
Cheshire, J.W. 5.38
Cole, R.H. 8.22
Dewilde, P. 1.1; 3.1
Duin, P.B. 1.69
Eindhoven, J.T.J. van 1.61; 4.1
Genderen, A.J. van 2.93; 2.115
Gerer, S. 6.1
Graaf, A.C. de 1.49; 2.93; 2.115
Graaf, S. de 1.1
Herrmann, O. 6.1
Hoeven, A. v.d. 1.1
Ivey, P.A. 5.120
Janssen, G.L.M. 1.49; 4.60
Jess, J.A.G. 7.1
Jongen, R. 7.28
Kleissen, G. 3.25
Lange, F. de 2.51
Larsson, D. 8.1
Leuken, T.G.R.M. v. 1.1; 2.1
Liedorp, J. 2.1
Lodder, A. 1.61
Luchtmeijer, R. 6.1
Lueddecke, K. 8.16
Meijs, N. v.d. 1.1; 1.30
Mulder, R. 6.1
Myers, D.J. 5.120
Nusteling, J. 1.1
Nuijten, P.A.C.M. 7.55
Orrey, D.A. 5.18; 5.38
Pope, S. 8.16
Revett, M.C. 5.1
Roe, B.J. 5.11
Roelofs, M. 3.25
Scarfe, R.T. 5.67; 5.110; 5.117
Schinner, P. 8.1
Schouten, W. 1.69
Slentier, A.G.J. 7.131
Smedes, P.D. 1.69
Smit, J. 6.1
Spanenburg, L. 1.69; 3.25; 6.1
Stiphout, M. van 1.61
Veenendaal, Chr. van 1.69; 3.25
Vogel, T. 1.1
Weeks, R. 5.52; 5.88
Weiner, N.H. 5.18
Wolf, P. v.d. 1.1

xxii
Chapter 1

Systems Architecture, Design Languages and Base Systems Components
DATAMANAGEMENT FOR HIERARCHICAL AND MULTIVIEW VLSI DESIGN*

P. Dewilde, S. de Graaf, A. van der Hoeven,
T.G.R.M. van Leuken, N. van der Meijs, J. Nusteling,
T. Vogel, P. van der Wolf

Delft University of Technology
Department of Electrical Engineering
P.O. Box 5031, 2600 GA Delft
The Netherlands

ABSTRACT

This paper describes the principles underlaying a multilevel and multiview database for VLSI design. The database must provide for the following functionality: (1) fast multiuser access to data at any point in the hierarchical and multiview matrix, (2) version and lock control, (3) project and library handling. Moreover, independent evolution of datamodels and schema should be possible, together with portability to a diversity of systems. We present an original solution to the difficult problem of version control in a hierarchical environment, a conceptual dataschema incorporating all the desired features, and a methodology for assuring independence between tools and database. The principles proposed have been implemented in the ICD Release 3 of Delft University of Technology, which is a prototype that is now being tested.

* This work has been supported in part by the Commission of the EEC, under the Microelectronics Regulation 3744/81 program, ICD project MR09, as part of Task I, Systems Architecture.
1. Introduction

In order to manage the dramatic increase in complexity new design tools are being constructed which exploit the inherent regularity and hierarchy of VLSI designs. The objective of the present paper is to provide an environment in which such tools can find a logical place. The design system desired will contain an integrated set of design tools as well as a repository for design data: the design database.

It is now generally recognized, that a design database is not the same as a classical database [4]. The objects which must be stored in a design database differ fundamentally from objects in a classical database. They consist of "aggregations of design data" or to put it simpler, of files of data, instead of the single records in classical database systems. Furthermore, design transactions exhibit different concurrency and durability requirements than transactions in conventional database systems, while the notion of version is not even explicitly supported in these systems.

The present paper is the result of two years of experimentation with the problems of reconciling the conflicting demands that are put on a database for VLSI design. We have started out with a study of classical databases, have looked at the work of Katz [4] whose approach does not yet start out from an explicit datamodel, next we have looked at different possible types of datamodels - classical hierarchical, network or relational [3] and more modern semantic [9],[11] - and have studied possible dataschemas (see the next section on terminology). At the same time many tools in the hierarchical-multiview context were being developed by ourselves and our partners. Very soon it became clear that a divorce between datamanagement and tools had to be defined, so that each could develop independently without being hindered by the evolution of the others. The databasework sketched led to the definition of three ICD releases at Delft.

In Release 1, an implicit Network model was used as datamodel, and where in the dataschema the view equivalences were supposed to be attributes of a cell. The interaction between database and tools was direct (without interference of a database manager) with the consequence that the location of information had to be hard-programmed in the application tools.

This undesirable dependence was resolved in the definition of Release 2 [13], where a layer of transaction functions was defined allowing for independent development of design tools and applications. Release 2 was using the same dataschema based on the network model as Release 1. The path, however was cleared for the development of a database that would incorporate version control and a multiuser environment, and yet support the multiview hierarchical structure.

Release 3 uses the same data-independent mechanisms as Release 2, yet implements a dataschema that is a reflection of the information environment of the VLSI designer who uses hierarchies and abstractions to define his design. At first, the schema for Release 3 followed the relational datamodel. Further insights,
and especially the study of the semantic model lead to a reinterpretation of the structure of the information.

In this paper we present all the ideas that have been incorporated in Release 3 and use the semantic datamodel as mathematical basis. It is structured as follows: in section 2 we make a brief review of general database concepts, in section 3 we give a narrative of the design situation and the way it has to be incorporated in a dataschema, in section 4 we show our original solution for the version control problem in our environment, in section 5 we give the dataschema in the semantic terminology and in section 6 we present implementational details. The concluding section is devoted to a discussion of what has been achieved so far and what remains to be done. Although Release 3 shows some inconsistencies in details, we believe that we have succeeded in solving the problem of integrating multiview, hierarchy, multiuser control, version control in one system in a consistent way.

2. General Database Concepts

2.1 Introduction

Before we go into details about which datamodel is suited and which schema we use to implement our design system, it is useful to discuss some major concepts of databases in general.

A database is nothing more than a computer-based record-keeping system, that is, a system whose overall purpose is to record and maintain information.

The main reason why we need a database is that a database provides a unified method to control the data. This has certain advantages over the opposite, when each application (tool) manages its own data-files and cumbersome programs have to be developed to translate information between the different data representations belonging to the different data-files. (4) See figure 1.

The first property of a unified representation of the data is, that the redundancy can be reduced, and when it is not possible to eliminate it totally, at least it will be controlled.

The point of redundancy removal leads directly to another important property of database systems: inconsistencies in the data can be avoided.

When every fact about an organization is recorded only once, then contradictions in the data cannot occur, and when a well-known amount of redundancy is allowed, then updates in the data can be propagated automatically to all entries where that piece of data is recorded. A centralized control can thus guarantee an internal consistency of data in the database.

A third aspect of database systems is that the data can be shared. In fact all users of the database can have access to all data, even concurrently, although in practice there will be well-defined security restrictions to protect "sensitive" data.
2.2 General Database Architecture

A possible way of looking at a database system is shown in figure 2.

The database is divided in three layers [3].

First we have the internal level. This is the level closest to the physical storage of the data. (How the data is actually stored, Files, Discs etc.).

The next higher level is the conceptual level. This level is defined as a "community" view of the data. In fact it is the logical organization of the data: which links (dependencies) exist between several "units" of data, each describing an "object" from the "real" world. This level includes all user views.

Finally we have the external level. This level is the one closest to the users. The external level defines the users view of the data. Each user will see the data at the conceptual level through a possibly different "window".

The division of the database architecture in levels allows for data-independence. This means that we can change our storage structure of data on the internal level and/or our access methods to that data without having to change all our applications. This in contradiction to non-database systems, where each application has build-in a certain amount of "data-handling" capabilities, which have to be adapted every time the data-structure and/or access methods are changed. The storage and handling of data is thus "visible" to the application. The only thing we have to do
in our case, is to change our mapping from the conceptual level to the internal level in such a way, that the "conceptual ordering" of data remains invariant.

2.3 Classical Data Models

The conceptual and external levels will be expressed by means of a "dataschema" which is derived from a "datamodel". This terminology is well documented in the literature [10],[11] and has now become standard in database work. A "datamodel" makes the (mathematical) concepts explicit which will be used to describe the data and its relations. Classical datamodels are: hierarchical, network and relational [3]. We have experienced (an account of our trials goes beyond the scope of the present paper) that none of these is powerful enough to express the information content needed here.

A better model which has come up in the early eighties, and which has a firmer mathematical foundation, is the "semantic" or "object" model. In this paper we adopt this semantic viewpoint. In this model there are two central abstraction mechanisms which we shall use: aggregation (i.e. type or template calculus) and generalization (existential calculus, object of type A "is an" object of type B). Once the datamodel is decided on, the dataschema must be build which is the faithful reflection of the information environment at hand - in our case hierarchical, multiview design in a multiuser environment and with
versioncontrol. This is the most difficult part and will be covered in sections 3-5.

3. The Design Environment

Our aim is to define a dataschema which will be an accurate reflection of the organization needed to master the complexity problems in VLSI design. Aside from the inherent complexity of the chip itself, we must handle the complexity due to external factors as the existence of various technologies and the complexity due to the design process itself: the creation of new versions.

A comprehensive design philosophy has not yet been developed and most probably never will be. For instance, normally a design process does not proceed strictly top-down or bottom-up, but will be an alternating process between refining subsystems into more detailed subsystems and composing primitives to form more useful building blocks. So it is very important to maintain all possible flexibility within the system and not to support a specific design philosophy explicitly.

On the other hand some design approaches have generally been accepted, such as the hierarchical composition of a design and the representation of certain parts of the design at different levels of abstraction. These approaches have to be supported by the system allowing the user to exploit their properties and thus managing the complexity. The solution presented here will be a compromise between generality and the detailed modeling necessary to represent and order the design data.

3.1 Regularity and Hierarchy

The most important way to handle the inherent complexity of VLSI circuit design is to structure the design process and restrict it to purely hierarchical design. The description of a VLSI design can be seen as a collection of hierarchically related objects, each one describing some part of the overall design.

The primary description of such an object basically contains three parts. The first part describes a piece of design using appropriate primitives. For example rectangles can be used to describe mask geometries (layout of the chip). The second part of the description contains references to other entities which are included as components. The third part describes the interface of the entity. We will call those objects 'cells' from now on.

Closely related with this hierarchical decomposition of a design is the possibility to describe regular structures in a very efficient way. A single cell may be instantiated several times as a linear or two dimensional array of cells, but is described only once. Examples of those regular arrays of cells are found in memory circuits and gate-arrays.

The datamodel must be able to reflect this hierarchical and regular structuring of data.
3.2 Abstraction

A second important method to handle VLSI complexity is to support several levels of abstraction in the description of a design. For example, a system designer will not be interested at all in layout, but sees the VLSI circuit as a network of logic gates. For each level of abstraction the design is described in certain primitives appropriate to that level. In the sequel of this paper we will refer to these abstractions as "views".

Each view describes the circuit to some extent, omitting details which are irrelevant to that specific level of the design. For example, the symbolic view describes the topology of connections and devices on the two-dimensional surface of the silicon, while omitting all geometrical constraints of the actual layout (design rules), whereas the circuit view describes the design as a network of transistors and even topological information is lost. Some other views can be defined in addition. Possible views and their primitives are given in table 1.

<table>
<thead>
<tr>
<th>VIEW</th>
<th>LEVEL OF ABSTRACTION</th>
<th>PRIMITIVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM</td>
<td>FUNCTIONAL</td>
<td>ALU ROM PLA etc.</td>
</tr>
<tr>
<td>LOGIC</td>
<td>DIGITAL</td>
<td>GATES SWITCHES</td>
</tr>
<tr>
<td>CIRCUIT</td>
<td>ELECTRICAL</td>
<td>DEVICES</td>
</tr>
<tr>
<td>SYMBOLIC</td>
<td>TOPOLOGICAL</td>
<td>EDGES ELEMENTS</td>
</tr>
<tr>
<td>LAYOUT</td>
<td>GEOMETRICAL</td>
<td>RECTANGLES etc.</td>
</tr>
</tbody>
</table>

Table 1. "Some views with their primitives"

At the highest level of abstraction, the system view describes the functional units of the circuit such as PLA's, ALU's, Registers (building blocks), and gives a behavioral description of the circuit.

At the next lower level, the logic view describes the circuit in terms of gates. Logical functioning of the chip and timing can be simulated at this level. The circuit view describes the electrical properties of the chip. This description includes a list of nodes and elements such as transistors, resistors, capacitors (all of which are relations between a set of nodes) and can be applied for electrical simulations.

Next, a symbolic view is defined, which contains topological information about the circuit: the ordering of connections and devices on the two-dimensional surface of the silicon. And finally, the layout view contains in addition geometrical information, that is, describes the mask-geometries, used for actual chip fabrication.
The different views of a design are not independent. Each lower level of abstraction describes the chip in more detail. In particular, the layout view of a chip contains all geometric information and is the most detailed level of description. It is possible to derive other views from this layout description. One example is the extraction of the electrical network from the mask geometries. Special tools for this purpose are already available. A database designer always tends to minimize the redundancy in data stored in his database, so it seems unlogical to support all views of a certain VLSI circuit when there exists the possibility to derive one from another. However, the computational overhead this involves each time a designer requests some high level view of his design can make this approach very unattractive. It is better to store most data, once it is computed, in the database, so that it will be readily available on request.

In the example below, the different views of a very simple piece of VLSI design (inverter) are shown.

Fig. 3: "Different views of the cell 'inverter'"

3.3 View and Hierarchy

We now come to an important issue, the interplay between view and the hierarchical decomposition of the design. It seems rather logical to consider the same hierarchical decomposition among all views of a design and describe every cell in several views. However, the constraints involving this decision would make the design system very inflexible.

At the one hand it is not appropriate for certain pieces of a design to be described in all views. A very low level cell (from a hierarchical point of view) may not have a system view description.
At the other hand, a designer must be able to experiment with the behaviour of a system, without having to design it first. It is in most cases desirable to start a design with the functional specifications and simulate it at a high level in order to optimize its architectural properties.

Another point is, that at some levels of abstraction certain aspects of a design must be explicitly stated in the description, while they are completely omitted or at least implicitly present in the description of higher level views of that design. For example, the interconnection pattern between different units of a VLSI chip (busses, power supply, wires etc.) will be present at the layout level as separate cells, while at the functional level such interconnection information is implicitly stated (common variables) and does not call for an individual unit. The functional decomposition of a VLSI design need not be the same as its physical (layout) decomposition. This leads in general to nonisomorphic representations: different decompositions per view. See figure 4.

![Different Decompositions among Views]

In general, a layout hierarchy will be "deeper" than a system (functional) hierarchy. At a higher level, the decompositions will usually be "almost" the same. So, in addition to the hierarchical decomposition presented in the previous section, we have to support several, possibly different hierarchical decompositions, one for each view. At the same time there may exist correspondences between the components of the different hierarchies: e.g. they all may refer to the same piece of "hardware". The dataschema will have to represent several hierarchies with "equivalence" links between them. See figure 5.
3.4 Basic Entities of the Database

At this point we can identify the basic entities the database must handle. They are the coded descriptions of pieces of "hardware" expressed in primitives belonging to a certain view. We shall refer to them as "objects". Typically these will be implemented as files or even clusters of files concentrated in directories. This in contrast to classical database systems, where the entities are names (strings) or numerical values. An object is handled by the database as an atomic unit.

Typically, the information contained in an object consists of three parts. [4]
See figure 6.

First we have representation data, which is the actual description of the object expressed in primitives of the view. The representation data forms the "body" of the object. This data will not be present in case the object is composed of other objects as a pure partitioning. The other extreme is, when only the representation data is present. This is the case when the object is a "leaf"-object [5], and has no dependents (sons). But mostly we have a situation in between: an object composed of other objects with some "glue" to connect them.

Next we have hierarchical data. This is data describing the composition of the object from other objects. The hierarchical data of a particular design-object typically includes a list of "sons", referring to the immediate dependents of the object which will always be of the same view. Added to this son reference will be some information (transformation, repetition) unique to the son-object's instance. This information will be used to instantiate the son object when the complete composition of the
father object is needed for simulation or display. We tend to a very flexible approach with respect to this instantiation information. This way it might be possible to handle a parametrized layout, a collection of alternatives, or even a generator the same way as an object: the right instance can always be derived using the instantiation information.

A third kind of data in connection with the description of an object is the interface data. An hierarchical decomposition of an object makes only sense when you don't have to bother about the internal structure of the components. The only thing the composite "sees" is the interface of the component-object and not its contents. Such an interface description will be different for each view, but typically it includes a specification of so-called terminals that are the I/O connections of the object and a type code, which could specify the terminal in more detail (input/output).

How the object is implemented is merely a matter of taste. It could be a single (structured) file which contains all information or a collection of separate data files describing the different aspects of the object. The method we use in our implementation is to cluster a number of "type" files which contain the actual data (ascii or binary) in a single "object" directory. See chapter 6 on implementation.

An important issue is always the scope of the database: which information concerning an object is actually part of the DB and which not. For the purpose of simplicity and transparancy, we take the conservative view that the DB should be as small as possible, yet include all the features that it is supposed to
support. E.g. we do not consider a "box" or a "wire" to be a part of the database. These will belong to larger objects "boxfile", "wirelist" which are part of the database.

4. The Version Concept

As is pointed out by others [4],[6], design data is a too precious resource to let it simply be overwritten on each design transaction. This would also consequently override any verification effort which was made before. A better approach is to create a new version of an object (under certain conditions) at a design transaction. This approach also offers the user a backup facility. Older versions remain available until they are explicitly removed by means of a database command. This way the evolution of a design process is retained to some extent and older versions can be used to try alternative design approaches.

Because updates are not done in place recovery is automatically supported.

Updates (changes) are propagated in a controlled way through the hierarchy of objects. A user will normally work on a cell for some time and only when it has reached some definite stage it will be 'installed'. So only cells which have past some rudimentary checks are added to the consistent hierarchy. On its turn this promotes a strictly hierarchical design approach.

4.1 The Concept and its Motivation

In principle, it is possible to support whole trees of versions per cell, in which the different branches allow the concurrent development of alternative design approaches.

This results in a highly complicated version mechanism, due to the hierarchical relationships between the cells. The implementation of such a mechanism is very difficult and, more importantly, the resulting system would be hard to understand by the users. Our aim is to implement a relatively simple mechanism which is easy to use and meets the basic needs of the designer. A satisfying solution can be obtained by restricting the mechanism mentioned above to a linear 'evolutionary' chain of versions, accompanied by a set of commands to manipulate the versions explicitly and thereby providing the user with all the features he needs.

At every moment the data within a project of the database has to form a hierarchically consistent set. Without any redundancy this
is always the case. However, some redundancy in the data can be very valuable for certain kinds of tools (performance). The demand to keep all data consistent at any moment would result in an unnecessary inflexibility. At every transaction the consistency of the whole project should be achieved, which would result in an unnecessary performance degradation of the application programs.

The version mechanism offers the possibility to distinguish between hierarchically consistent and hierarchically inconsistent data within the database. This also offers the possibility to postpone the consistency check until the object has reached some definite state after a number of edits and verifications. This enlarges the flexibility of the system and the performance of the applications.

To prevent that during every transaction a specific version has to be specified, we need a mechanism which allows unambiguous identification of a version. There are two ways to refer to a cell:

- the cell in question is being handled at the moment. Such a cell is the 'root' of a tree of cells which represents the part of the design on which the tool acts. ('work' reference).

- the cell is instantiated within another cell at the moment. It is a cell somewhere in the tree mentioned above. ('browse' reference).

Each of these 'ways' has its own demands with respect to concurrent use of information within the project. These two ways of referring and the difference between consistent and inconsistent data can be very well combined with the demand of unambiguous reference to a version of a cell.

A 'work' reference will always be related to the last version of the (evolutionary) chain. This is the one which is updated and checked by the tools, but isn't necessarily a member of the consistent tree of versions of the different cells. The 'browse' reference will always be related to the version in the chain which is part of the consistent tree within the project at that specific moment.

We will use the following terminology: The version which is part of the consistent tree will be called the actual version. This can be the last version of the chain, but if the cell is updated after the consistency check another version will be the 'last of the chain'. This one will be called the working version.

We can define the following set of rules which apply to the versions and their evolution:

- A 'work' reference will be related to the 'working' version if present. Elsewhere the 'actual' version will be taken.

- After any update the version will be stored as the 'working' version. If this 'working' isn't present it will be created.

- A 'browse' reference will always be related to the 'actual' version, being part of the consistent tree.
The whole problem is reduced to a scope-principle. By allowing the existence of 'working' versions which are not necessarily upward consistent, the concept offers the possibility to work on cells without unnecessary consistency checks, while a small number of scope rules supports unambiguous identification.

One detail: when editing a specific 'working' version it will show the consistent 'actual' versions of its children during an expansion. This property is based on the next rule: Within a true hierarchy a change at a higher node will never influence the lower nodes. (A cell will only instantiate its subcells but never change their description) So the children don't contain information about their father. This way downward consistency is always maintained when acting on a certain node.

One problem remains: what to do with a new cell? The data will be placed in the first and only version of that cell. This version will directly have the 'actual' status. This is allowed because no other cells can already refer to this new cell; it is always upward consistent. It also agrees with the rule that every cell has its unique consistent version.

A version chain always has the following appearance:

```
( )b ... ( )b ( )a ( )w
```

The 'actual' version is always present. One 'working' and/or one or more 'backups' may be present.

The 'working' can (after it might have passed a number of edits and verifications) become part of the 'actual consistent tree' by means of a special 'install' command. The status of this version will now become 'actual' and the upward consistency will be achieved by updating all redundant data in the higher 'actual' versions up to the root of the tree. The previous 'actual' will now become a 'backup'.

```
( )a                        ( )a
\   \                      \   \a = 'actual'
\   \                        \   \w = 'working'
\   \                        \   \b = 'backup'
( )a ( )w                   ( )b ( )a
```

The downward consistency of the 'workings' also has to be preserved. So all 'actual' and 'working' versions which directly or indirectly refer to the installed cell will be made downward consistent. Because upward consistency of the workings isn't preserved we don't have to propagate their consistency. The consistency of the 'actualls' will be propagated in the 'actual'
tree as mentioned above.

1) The older 'actuals' (now backups) of a cell are always part of a consistent tree. To preserve this consistency a copy has to be made of the higher 'actual' versions before the consistency check.

2) We sacrifice the consistency of the old 'actual'. We don't have to copy the higher versions, but a special 'restore' command will be needed to bring an inconsistent backup 'back to life'.

Possibility 1 leads to 'bunches' of versions, especially at the higher levels. Furthermore the resulting picture which is reflected to the user won't be clear at all. Possibility two results in one new version per updated cell, which is far more logical. It is clear that possibility 2 is preferable.

Old backups may not be confused with 'working' versions whose downward consistency is always preserved. A special command will be needed to make the 'backups' usable again. Such a 'restore' command will bring the backup to the end of the chain. Another command will allow the user isolate a backup and transport it to a new cell. This way a user is still able to try alternative design
approaches although the version mechanism doesn't support it explicitly. Furthermore a 'rm_backup' command will allow the user to remove useless backups. It will also be possible to set a limit to the maximum number of backups per cell.

To allow the existence of more than one consistent 'set' within a project we can introduce the 'released' status. A number of versions can obtain this special status by a 'release' command: the configuration of a set of 'actual' versions can be 'frozen' and won't be affected by a consistency check at a later time. The versions belonging to a released set cannot be removed without affecting the release status of versions in a number of other cells. Commands can be made which allow the user to restore a configuration of these older versions at a later moment. The 'released' is not yet implemented but this can be done easily whenever necessary.

4.2 Summarizing

At a certain moment a cell may contain the next versions:

- a 'working' version
- an 'actual' version
- a number of 'backup' versions

Only the 'actual' versions are part of the consistent set of versions and can be 'browsed'.

Within a project of the database we can have:

- an 'actual' set of consistent versions. This set always spans the complete hierarchy within a project.
- a set of inconsistent 'backups' of cells.
- a set of upward inconsistent 'workings' of cells.

It may be clear that this version concept leads to a flexible multi-user support. From a certain node we always see a 'frozen actual tree' of consistent versions, even if someone else is working on a cell in the subtree. This means we don't have to lock whole subtrees (whose composition we might even change during our reservation by adding new subcells) but can handle the concurrency with a simple form of 'node locking'.

The complete version concept can be combined extremely well with the relational approach we have chosen for this database. According to this approach a project is a 'pool of objects'. The relations can cluster a number of objects to be the versions of a cell. The consistent configurations of versions (objects) can also be described this way.
5. The VLSI Dataschema

The schema to which the design situation described in the previous paragraphs leads will now be given. It must incorporate "hierarchy", "views", "design-equivalences" (resulting from the action of tools), "locks" (for the multiuser environment). To be correct, the schema must satisfy the "principle of equivalence", which states that any object must be completely characterized by its attributes. An examination of the dataschema, which is given in fig. 7, will reveal its correctness in this sense.

Fig. 7. The dataschema (for explanation, see text).

It specifies, topdown, the attributes of the relevant objects, represented as rectangles. Each object is characterized by a unique list of attributes. Some of these are other objects, and some are of leaf type. The latter may be either very small (e.g. a number) or very large (e.g. a file of boxes).

The schema contains one, essential, specialisation indicated by a slanted arrow: the specialisation in views. A design-object can be either a layout, a circuit or a system, or even any other type considered useful (we take a dim view of incorporating "mixed" types in the schema: they can easily be constructed hierarchically using the present schema). Each specialisation is characterized by a unique set of attributes which is not indicated in the schema to avoid congestion. A design object of the layout-type is, beside the attributes inherited from the generalized design object, characterized by a "boxfile", a "terminalfile" and a "modelcallfile" (and maybe some others), while an object of type circuit would be characterized by "netlist", "subnetwork", "pins".

All design objects share the property that they have version-attributes and that they are characterized by a "model"
identification, which is the common property of the versions in a chain.

Aside from the design-object layer, the schema contains three more layers: First there is the "model" layer, whose attributes are the lock-information. It is at this point that the multidesigner environment is established. How locks work in our system is explained in the next section. Furthermore, there are the Hierarchy and Equivalence layers whose type contains, among other things, two design-objects. For example, hierarchy establishes a relation between a "father" and a "son" design-object, while equivalence establishes a relation between the input and the output of an application function, both presumed to be of the type "design-object".

A detailed formal description of the schema is of course possible, but goes beyond the purpose of the present paper and is omitted. This schema, which is essentially a "designers' view, does not represent system elements like the structure of the project hierarchy (which will be discussed in chapter 6), nor does it show the place of the technology information which is attached to relevant projects. At the present point this type of information, although important, is not fundamentally difficult to handle, and is also omitted. For implementation details, we refer to the ICD manuals [12],[13].

6. Implementation

As mentioned earlier, we have been experimenting with different prototypes which contain various implementations of the above schema. The most complete prototype so far is "Release 3" whose principles will now be explained. In section 7 we shall make a criticism of it and indicate how we may evolve to the future.

6.1 Structure.

The general structure of the implementation of the Release 3 database is presented in figure 8. Basically, it consists of two parts for each project. The first part contains all design information. We use the UNIX file system as a storage component, in which all design objects are stored. The design objects, which themselves are files of raw design data, are the basic entities of the database. Each design object describes a certain part of a design in terms of primitives appropriate for the view of that object. It is implemented as a directory which contains a number of data files, describing the object. For example, a layout object will contain a 'box' file, a 'term' file, a 'me' file etc. Also, derived information could be added to an object like other data representations to support certain tools.

All objects of the same view are clustered in a single directory, the view directory, reflecting the view specialization, but this is merely an implementation matter. The chosen storage structure, based on the UNIX file system, is more or less arbitrary and contains no conceptual information. In fact, we can see all objects as a large 'pool' of design data. At this moment, the
The database supports only three kinds of views: the layout, circuit and system views, but others can be added when needed.

The second part incorporates a simple implementation of a relational database on top of the first part. The complete conceptual structure, relating the individual objects of the 'pool' mentioned above, is documented here. In a number of 'index' objects the tables of the relational database are stored. These 'index' objects administrate the different relationships between the 'aggregates' of data in the hierarchical data file system. The index objects take a central position within each project. Each time a transaction within the project completes, the index objects are updated. In this way the activities of several designers working on the same project are administrated and possible conflicts can be avoided. The index objects are implemented as fixed format ascii files, with a hash access structure on top of it.

In figure 8 a main project is presented with a number of sub-projects, which in turn also can have sub-projects. For one of the projects a detailed picture is given. See figure 8.

**Fig. 8: "Basic structure of the database"**
6.2 Kernel relations.

A set of standard relations is used to describe the different relationships between the objects in the database. These relations are automatically updated on every design transaction. The user can browse these relations for certain information. The following relations are implemented.

- **Hierarchy.**
  In this table the father-son relations of the cells in the project are administrated. This table is used mainly to detect recursive calls among the cells. The actual references and instantiation information are stored at each object in a special file, the 'cell-call' file. See figure 9.

<table>
<thead>
<tr>
<th>father</th>
<th>son</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM</td>
<td>PLA</td>
</tr>
<tr>
<td>FSM</td>
<td>REC</td>
</tr>
<tr>
<td>PLA</td>
<td>and_plane</td>
</tr>
<tr>
<td>PLA</td>
<td>or_plane</td>
</tr>
</tbody>
</table>

Fig 9: "Example of table containing father-son tuples"

The hierarchy-table implements the "hierarchy" objects of the data schema.

- **Design Objects.**
  The design-objects, their attributes and specialisations are implemented as a collection of tables, one for each view. An example of such a table (slightly simplified) is:

```
<table>
<thead>
<tr>
<th>l_obj</th>
<th>c_eqv</th>
<th>s_eqv</th>
<th>model</th>
<th>vers</th>
<th>vers_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt1</td>
<td>ct13</td>
<td>s5</td>
<td>basis</td>
<td>0.1</td>
<td>actual</td>
</tr>
<tr>
<td>lt2</td>
<td>ct2</td>
<td></td>
<td>shcel</td>
<td>0.1</td>
<td>backup</td>
</tr>
<tr>
<td>lt3</td>
<td>ct14</td>
<td></td>
<td>blink</td>
<td>0.1</td>
<td>actual</td>
</tr>
<tr>
<td>lt4</td>
<td>ct1</td>
<td></td>
<td>basis</td>
<td>0.2</td>
<td>working</td>
</tr>
<tr>
<td>lt5</td>
<td></td>
<td></td>
<td>shcel</td>
<td>0.2</td>
<td>actual</td>
</tr>
</tbody>
</table>
```

Fig. 10: (Simplified) Example of the Design_Object Table.

These tables are the central repository of the design information according to the given schema. As explained earlier, the attributes of the design_object specialisation are implemented as directories and files, and are not a part of the tables. This is a (slight) inconsistency because "design_objects" and their specialisations in "layout_object", "circuit_object" etc... should have different tables, with a unique reference between an object and its specialisation. As it is done now we end up with redundancy between the design_object table and the view directories. This redundancy is well documented and can be
tightly controlled by the transaction functions.

- Models.
The model table contains the locking information which will be discussed in the next section. It allows for multiuser access to the design information so that several designers can work at the same time with the same data.

6.3 Locking.

In order to allow the concurrent access of data in the database by several users, some kind of locking mechanism must be implemented. Basically, we can distinguish two kinds of locking within the database.

6.3.1 Index_lock.

This kind of lock assures the mutually exclusive access to the index information in a project of the database. The index data describes all relationships between the aggregates of design data (objects) in the database and determines the 'physical' location where the data is stored. Because all index information of a whole project is stored centrally in a number of files, each transaction between the database and an application program has to access these files. Several users may try to access an index file for update, while still others are reading it. This must of course be forbidden, and a lock is set on the entire project allowing only one application to obtain its index information. The lock is removed when the update/read activity is finished. The locking of the index files typically is of short duration. In fact it is an implementation matter, which is invisible for all users of the database. The only effect on the application programs is a (shortly) delayed access to the requested data.

6.3.2 Transaction_lock.

Apart from the index lock, a transaction lock is introduced, which must prevent the access of several users to the same cell. It assures that only one designer can change a cell at a time. No updates can be lost. However, several designers may be working on different cells within the same project. The transaction lock 'reserves' a cell for exclusive use for a certain period of time. This may be several hours.

For each cell only one transaction can be in progress at a time. Each transaction starts with a 'consistent' database and leaves it in a possible new 'consistent' state. Because of the transaction lock, a transaction can be viewed as an atomic action on a cell. A transaction must commit or must be rejected totally. Incomplete transactions must be avoided at any price. In reality the database passes through a number of 'inconsistent' states during a transaction. The transaction lock prevents access for other users during this period.

In order to be able to recover from a crash, there must be enough information in the database in each of the 'inconsistent' states either to complete the in-progress transaction (commit) or to rewind it to the last 'consistent' state reached before the crash.
The locking mechanism provides locking in a multiple process environment, where 'child' processes inherit the access rights of the 'parent' process. Every application has a 'process locking code' which determines which locked cell can be accessed. An application can lock cells using his PLC. When child processes are created, they are given this PLC, and they will have the right to access locked cells if their PLC matches the code of this lock. So an application will have access to all cells which were locked by himself or one of his parents.

This mechanism supports that a DRC or extraction can be started out of a layout editor which had already locked the specified cell. However, it is assumed that the parent waits for the child(ren) to finish, so that no more than one process at a time will actually access the data.

6.4 Architecture

A fundamental difference between a commercial DBMS, designed for business and administrative applications, and a design DBMS is that in a commercial database the interactions between the 'user' and the DBMS are very frequent and the amounts of data which are exchanged are relatively small. This in contradiction with design databases, where a single transaction can span several hours and the amounts of data involved are large.

Basically, we can divide the users of a design environment into two groups. First, the 'keyboard' user. This is the designer behind his terminal, consulting the database for small portions of data at a relatively high rate (browse). Secondly, we have the application programs (tools) which act directly on large amounts of design data and which can hold the data for relatively long periods. Both kind of users need an interface to the database. For the 'keyboard' user, a relatively simple but complete command language would be sufficient.

The application software needs a more direct interaction with the data in the database. This applies to both non-interactive programs such as design rule checkers and interactive programs such as layout editors. For this purpose, a set of "low level" I/O functions are developed which allows the application software to act more or less directly on the data in the database. At the same time, these functions provide the minimum amount of protection, needed to assure the internal consistency of the database. Also, these functions provide a way to control the concurrent use of data.
6.5 User Interface

One of the most important components in a design system is the interface with the user. It forms the "face" of the system to the outside world. Several aspects of such an interface must be taken into account when designing it.

A major aspect is its 'friendliness' to the user. A designer does not want to use difficult and/or complex procedures in order to state his requests to the system. A system with a complex behaviour, irrespectively how beautiful or powerful it may be, will never be used.

A relative simple command language will mostly satisfy the needs of the designer. However, some basic elements must be present in the language. The designer must be able to manipulate elements, start processes and specify source and destination of data used by them.

In addition to this, some important database information could be presented to the user in a graphical way. This way of presentation will greatly enhance the communication between the system and the user.

6.5.1 The CAD-Shell

We propose an approach in which a command interpreter, the "CAD-Shell" takes the central position in the user interface. This "CAD-Shell" parses the command entered by the designer and activates one or more processes. (See the figure below).
Such a "CAD_Shell" can be very helpful in administering the tasks (processes) the user has started and inform him when such a task is completed. (Compare UNIX C-shell).

The commands the CAD-Shell will accept can be single commands, possibly with one or more arguments as well as command structures. Command structures can be used to define source and destination of data which is used among the processes invoked by the shell. We can think about the usage of "redirect" and "pipe" principles or a more procedural way using parentheses.

As an extension we can add a "menu"-option. A beginning user can learn in a fast way the capabilities of the system, while advanced users will not be irritated by the process of stepwise refinement in the specification of a request. An advanced user wants to see a "prompt" and state his requests to the system in a direct manner using the expressive capabilities of a powerful command language.

The CAD-Shell will, in analogy with the UNIX Shell, offer the designer an abstract model of the environment he is using. This will typically be a tree structure of nodes (projects/libraries), where each node may contain an arbitrary number of elements (cells). The node defines the environment in which a designer is working. All processes the user invokes are "connected" to the node where they were started. The designer can move through the tree or specify elements at nodes he is not presently attached to by means of "paths". These paths follow a ../../.. pattern. They may be absolute paths (starting at the root node) or relative paths (starting at the current node).

6.5.2 Command_language

As in UNIX we can define for each process a default input and output. In combination with the "pipe" and "redirect" mechanisms this will offer the user a powerful tool. For example, a command could look like:
Where B, C and D are processes and A and E specify the source and destination of the data involved. Because processes could have more than one argument we also can use a more procedural syntax using parentheses:

(display (sls (extract A) , B))

Where A is a cell whose layout is to be extracted, resulting in a circuit. Together with the signal file B this circuit is simulated (switch level) and the results are displayed on a graphics device. The example shows us how we can get the logical response of a given layout in just entering one command line. The destination could also be a plotter, or could be stored somewhere as a file in the database. The signal file B could be the result of a special 'signal editor' and can be multiple used whenever convenient.

6.5.3 Projects

As mentioned above the project structure proposed for the prototype design system has many analogies with the unix directory structure. Each user is attached to a home project. A user may define new projects and can remove old ones (if they are empty). In this way whole 'trees' of projects may develop. The project concept creates an 'environment' for the user in which he can create and manipulate his cells. The cells in such a project form a separate design hierarchy. Within the design hierarchy, 'calls' can be made to cells in other projects (libraries) by specifying explicitly a path to the appropriate project. When no path is specified, the current project is searched for the called cell.

6.5.4 Libraries

An important facility within a design system is the presence of one or more libraries. Such libraries contain standard cells which will extensively be used by the designer or by the tools he invokes. (module generators etc.) The structure of a library is in essence the same as that of a project. Only the access/update permissions will differ. A designer can use a library cell simply by keying the cell name with the path to the corresponding library. This path can be either a relative path or an absolute one. In the same way he can use cells in other projects of himself or of other users, although this introduces some additional difficulties. A cell in a library is 'static' in the sense that it will not be updated or removed. So it is safe to refer to such a cell without the danger of 'dangling' references. Models within other projects are exposed to changes or removal. Reference to such cells must be administrated very carefully or even be forbidden totally.

Although the way of referring to library cells is very simple (just name the lib!) the implementation details involved with this facility are quite difficult. A cell can refer to other cells, which are expected without further specification in the same environment (project) as the calling cell. When a cell explicitly

1.25
calls a cell in another environment (library) all descendants of that cell must be searched in the new environment recursively unless again a new environment is specified. This 'dynamically' switching of the context in which cells must be searched while scanning a design hierarchy is not a simple problem. However, it is purely an implementation matter and must be shielded completely from the user.

An example of a typical situation is given in the figure below, where each triangle represents a design hierarchy.

![Project Hierarchy](image)

6.6 Application Interface.

The application interface contains a number of functions which offer the applications a well defined way to communicate with the database. Within these functions the following features are realized.

- **Data independence**: The structure (format, type) of the data within the files is shielded from the applications. An abstract transfer mechanism is implemented using standard C-structures.

- **Structure Independence**: The composition of the database, with its files and directories, is totally invisible. On request the functions deliver the right filepointers to the specified design files.

- **Version Mechanism**: New versions of a cell are created automatically. The version concept assures unambiguous reference to the right version of a cell.

- **Locking**: A requested cell locked to support concurrent working of several designers within the same project.

- **Recovery**: The internal file-handling is set up in such a way that recovery will always be possible. A recovery tool (which still has to be programmed) will be able to bring the database in a consistent state after a crash. This can be achieved by either rewind or commit the in-progress transaction.
6.6.1 Procedure

What happens when an application interacts with the database is described in the following steps.

First the application program reserves the data it is going to use/update. The data is locked for other users. Next, a path to the data is given by the DBMS to the application program. The application program can then act on the data. What mostly will happen at this stage, is that the application program reads all the data in a private datastructure, acts on it and writes it back to the database at the end of the session. Finally, the data which was reserved by the application program must be released again and some consistency and integrity constraints can be checked.

In order to support this kind of interaction with the database and to minimize communication overhead, a set of "kernel" functions is supplied by the DBMS which allows the application software to handle the data in a relatively direct manner. Furthermore, the application is shielded from the internal structure of the database. We will call these functions the 'application interface'.

6.6.2 Records

An abstract transfer mechanism has been implemented to assure data independence. This mechanism is based on a set of standard C-structures, which are listed in the programmers manual [12],[13]. The user (application) solely uses these functions to read and write records. This way the actual storage structure is shielded from the applications (binary or ascii) and can be changed without affecting these applications.

At this moment the contents of the database are stored as ascii files. Especially in distributed workstation environments this might improve transportability significantly. All structure members are separated by a space and every record ends with an 'end-of-line'.

6.6.3 Further Information. Extensive further information on the Releases can be found in the internal publications of the Network Theory division in Delft. A complete description of the Database management functions is to be found in [13], for Release 2. These functions are valid for Release 3 as well whose description is in [12].

7. Concluding Remarks.

The present paper has described the status of the database architecture work as it stands at the end of a two years' period of experimentation and prototype development. Overviewing the work done we arrive at the following conclusions:

- The problem of deriving an appropriate schema incorporating such diverse items as hierarchy, views, version control,
concurrency control and design equivalences has been solved. In view of the existing literature, we believe that this is a major achievement.

- The implementation given is a natural extension of what a traditional user of a workstation with a powerful operating system like UNIX is used to. It achieves independence of datamanagement and tooldevelopment, and therefore warrants for natural growth and evolution.

- Although the datatransaction functions are now ready to be called a "system standard", they have only partially be implemented in the partners' software due to the fact that the results of the research presented here have not been available for a long time, allowing software designers to adapt. However, we can say that very substantial parts of the system have already been converted to the new standard. For sure, the ideas presented require more testing, but with the testresults available so far we are justified to call the Release 3 database, the present ICD database.

- Release 3 is still based on the oldfashioned relational model. The text has hopefully clarified the relation between the semantic database and a table implementation. In a faithful implementation of the schema of fig. 7, one would expect one table per type or class of object.

Future releases will be developed as follows. In the first place we wish to introduce far more extended object oriented programming of datatypes as was done originally in Smalltalk [14]. The best candidate for that is the use of the programming language EC [15], in which very powerful, set oriented schemas together with objectspecific datamanipulation is possible. This would allow to add an artificial intelligence layer on top of the existing structures. The possibility of doing rule-based calculus on data gleaned in the database in a transparent manner would greatly enhance the possibilities of the present system. To achieve that will be one of the major goals for further development. Databases for design in the future will also be greatly enhanced if use can be made of distribution of data over workstations. This feature also will be added.
3. References

ABSTRACT

Ldmc is the C programming language with embedded layout description statements. A preprocessor as a front-end for the C-compiler recognizes all embedded layout statements and converts each of them to a corresponding C function call. The parameters of this function are extracted from the layout description statement. When such a function is called, it outputs the corresponding layout statement with the computed values of the actual parameter expressions inserted in it.

Furthermore, these functions build and maintain an internal data structure containing useful information about the layout produced. This information is used by other high-level layout assembly functions in the ldmc library. The embedded language proves to be a powerful tool for implementing automatic layout generator programs. It has been used to define several generators, including generators for PLA's and RAM's.

1. Introduction

The cells constituting a VLSI chip layout can roughly be subdivided into two groups: (1) basic cells, and (2) macro cells. Basic cells are at the lower levels in the hierarchical tree, but not necessarily leaf cells. They mainly consist of primitive layout features (i.e. boxes). Examples of these cells are flip-flop cells, PLA in- and output cells, full adder cells, etc. Macro cells are mainly compositions of basic cells and perhaps other macro cells, constituting some more or less high-level
A placement and routing program is usually invoked at the macro cells at the top-level to complete the chip layout. Besides references to other cells, they usually also contain some "glue" primitives to connect the various instances of their constituting subcells. Very often, their structure is more or less regular, for instance a one- or two-dimensional array. Examples of macro cells are register files, RAMs, PLA's, multiplexers, counters, multipliers, etc.

In (custom) VLSI design, such basic cells and subsystems are used very often, so they must be available under the finger-tips of the designer. This will drastically speed up the design process, making his job more cost effective. Two approaches are used here: cell libraries for the basic cells, and algorithmic layout generator programs for the macro cells.

The basic cells in the library may be made more universally applicable when they are defined symbolically or algorithmically [1-5], as opposed to geometrically. This way, both the electrical properties (for example, transistor aspect ratio) as well as the geometrical properties (for example, terminal pitch) may be adapted to the environment in which the cell will be used.

When the basic cells are defined symbolically, a compactor takes care of the detailed designer specifications and will produce a geometrical layout, subject to a specific set of design rules. This approach suffers from two disadvantages: (1) the resulting layout is often non-optimal with respect to area criteria, and (2) the kind of manipulations that can be performed on the cell is determined by the compaction program instead of by the designer [6].

Algorithmic or procedural layout of basic cells does not suffer from these disadvantages, but it is somewhat less flexible. For example, design rule independence is difficult to achieve. Also, algorithmic layout is somewhat harder to specify. However, if one wishes to design the most heavily used or most critical cells as optimally as possible and geometric layout is too rigid, algorithmic layout may be preferred above symbolic layout. We therefore believe that a good design environment must support all three approaches to basic cell design.

However, while algorithmic layout has some drawbacks for the design of (irregular) basic cells, it seems to be the best approach to define parameterized macro cells. In particular, the regular structure of most macro cells lends itself naturally to be expressed in an algorithm. The algorithm defines how to assemble the macro cell from the basic cells. It accepts some parameters expressing the designer's specifications, (for example, the wordsize and the number of words in a register file, or the number of inputs, outputs and minterms, and the personality matrix of a PLA) and when executed, it will produce macro cells tailored to his needs.

We now see that both approaches (i.e. cell libraries and layout generators) are complementary to one another. The layout generators beneficially make use of the basic cells in the cell library. They execute an algorithm receiving as input the
designer's specifications and that assembles the basic cells.

Moreover, if these algorithms are implemented properly a generator will produce a correct result for many different (but self-consistent) sets of basic cells. By "consistent" we mean here that the cells fit together geometrically (and electrically) to produce the macro cell. The above observation is very important: it follows that the generators themselves can largely be design-rule independent. To adapt to other design rules, only the set of basic cells must be replaced by another (consistent) set. Of course, this is particularly easy when the basic cells are defined symbolically or algorithmically. But even when the cells are defined geometrically this is perhaps not too expensive using powerful graphical edit facilities.

To support the implementation of layout generators, as well as algorithmic definition of basic cells, we have (1) embedded our geometrical layout language in the C programming language (7) and (2), augmented this system with a software library containing some powerful layout assembly functions. In effect, this system provides us with a powerful tool for implementing layout generators.

The rest of this paper is organized as follows. In the next section we will give an overview of the system, followed by a description of the silicon assembly functions in the ldmc library. An illustrative example showing how these functions may be used to implement a simple PLA generator is presented in section 4. Finally, a discussion is presented in section 5.

2. Overview of the ldmc system.

Generators written in the embedded language, called ldmc), output layout descriptions in the ldm language. An ldm layout consists of boxes, instances of other cells, and terminals; terminals are named boxes defining the interface of a cell. Ldm is not parameterized. For a complete description of ldm, see [8]. However, in ldmc we have available all the expressiveness of the C-language, its control structures such as iteration and selection, its procedure mechanism, as well as its interface to the UNIX operating system.

All ldm statements have a corresponding C function, in a precompiled form available in the same library as the assembly functions. A preprocessor operating as a front-end for the C-compiler converts all embedded ldm statements to calls for the corresponding C functions. The resulting C-code is then sent to the standard UNIX C compiler, and the resulting object code is linked with the object code needed from the ldmc library (and perhaps other libraries) into an executable program. When this program is executed, it produces an ldm layout description, which can then be entered into our design database. A pictorial description of this process is given in fig. 1.

The main advantage for having a preprocessor translating the embedded ldm statements into C as compared with just writing the C program directly, is the simple interface with other layout design
tools. For example, it is now very easy to take some ldm
description of a certain layout, created either graphically with a
layout editor or with a text editor, and edit it to parameterize
it. Fixed numbers can be replaced by variable names, some control
structures can be added and so on, to remove unwanted constraints.
Then, only some declarations must be added to obtain a
parameterized layout description.

Another advantage is that optional fields in the ldm description
remain optional. The preprocessor fills in the appropriate
identity- or null-values of the omitted parameters of the ldm
statement. As a result, the ldmc program is more readable than
the corresponding C program.

2.1 Example

To give the reader a feeling of the basic features of ldmc, we
will now give an example of a simple ldmc program. This program,
when compiled and run, will produce a cell named "cell2", being an
array of the cell "cell1". The number of cells in the array is
parameterized. These parameters must be specified on the command
line when invoking the program. However, the size of the subcell
is not specified in the program, nor on the command line.
Instead, the repetition distance is computed by the program.
Therefore, the program works for each particular implementation of
"cell1".
main (argc, argv)
int argc;
char *argv[];
{
    int nx, ny, xp, yp;
    nx = atoi (argv[1]); /* x-repetition count */
    ny = atoi (argv[2]); /* y-repetition count */

    readldm ("cell1.ldm");
    xp = pitch ("X", "cell1"); /* x-rep. dist. */
    yp = pitch ("y", "cell1"); /* y-rep. dist. */

    /* begin layout */
    ms "cell2"
    mc "cell1" cx xp nx cy yp ny
    me
}

In this program, the line

    readldm ("cell1.ldm");

is an invocation of a function in the ldmc library. This function
opens the file named in its argument list (which must be a file
containing a ldm layout) and updates the ldmc data structure using
the layout description in this file.

Pitch is another function in the ldmc library. The lines
containing

    xp = pitch ("X", "cell1"); /* x-rep. dist. */
    yp = pitch ("y", "cell1"); /* y-rep. dist. */

assign to the variables xp and yp the pitch of cell1 in the x- and
y-direction respectively. The pitch function inspects the ldmc
data structure and obtains its return value from the positions of
the terminals of cell1. The cell cell1 must be defined in the
file cell1.ldm, otherwise pitch complains about the fact that
cell1 is not defined.

The lines

    ms "cell2"
    mc "cell1" cx xp nx cy yp ny
    me

are embedded ldm statements. The first is the ldm model-start
statement. This statement is the first statement of a cell
definition. The name of the cell is a parameter of the statement.

The second statement is a model-call statement. It is used to
instantiate some cell ("call a subcell") in the cell that is
currently being defined. The parameter "cell1" is the name of the
subcell. The attributes cx xp nx and cy yp ny of the mc-statement
mean that the instantiation must be repeated nx times in the x-
direction and ny times in the y-direction, resulting in a total
number of (nx+1)*(ny+1) instances of the cell. The repetition
distance is $x_p$ and $y_p$ respectively. $C_x$ and $C_y$ are ldm keywords, rather than parameters.

Finally, $me$ is the model-end statement, ending a cell definition.

If this program is in the file `demo1.ldmc` we may compile it with the command:

```
ldmcc demo1.ldmc
```

and then run it with appropriate arguments:

```
a.out 5 6
```

The following output is produced:

```
:model cell1
:dimensions xl=0 xr=24 yb=0 yt=24
ms cell1
  term np 0 4 0 4 left
  term np 20 24 0 4 right
  term np 10 14 0 4 bottom
  term np 10 14 20 24 top
me
:model cell2
:dimensions xl=0 xr=144 yb=0 yt=168
ms cell2
  mc cell1 cx 24 5 cy 24 6
me
```

This output is a ldm layout description, defining the cells `cell1` and `cell2`. The definition of `cell1` is just copied from the file `cell1.ldm`, but the definition of `cell2` is new. The repetition distance in x- and y-direction (i.e. 24 in both cases) as well as the repetition number in the x- and y-direction (i.e. 5 and 6) are filled in.

Although this example does not show all aspects and possibilities of the ldmc language, it should have given the reader a feeling of the basic features of ldmc.

3. The ldmc library.

In this section we will discuss the functions in the ldmc library in some detail. To do this, we will group the functions in the ldmc library in several classes, and dedicate a subsection to each of these classes. Where we give examples, they are not chosen to be the most efficient solution to the example problem, but only for the sake of illustration. A working knowledge of the C-programming language [7] or some other modern programming language like Pascal is sometimes assumed. Furthermore, the discussion will be somewhat informal and perhaps incomplete. An interested reader is referred to [9], containing the users guide and the reference manual.
3.1 Primitive ldm functions and ldmc data structure.

These functions have a one-to-one correspondence with the embedded ldm statements, and their calls are generated by the preprocessor. When they are executed, their result can roughly be described as three-fold.

First, they output the corresponding ldm statement, with the computed values of the actual parameter expressions inserted in it. Secondly, they provide a check for certain semantic errors. For example, the box function complains about illegal layer codes, negative repetition parameters, etc. These error messages are presented in a distinctive form in the ldm output stream.

Finally, and most importantly, these functions build and maintain an internal data structure. This data structure contains for every cell some information about its dimensions, its terminals, and the instances of its subcells. It consists of three kinds of nodes:

- **MODEL** nodes - mainly containing data about the size of a cell and references to its INST and TERM nodes.
- **INST** nodes - mainly containing data about the subcells of a cell.
- **TERM** nodes - mainly containing data about the terminals of a cell.

Each node is implemented as a C structure (record).

The data structure has two merits: It makes superfluous much of the tedious bookkeeping tasks normally found in a layout generator program, and this data structure makes it possible to augment the ldmc library with high-level layout composition functions. The ldmc functions supporting the bookkeeping are described in the next two subsections, followed by the description of the other functions in the ldmc library.

3.2 Low level data structure access.

The functions in this class can be used to retrieve some information that is present in the ldmc data structure. For example, to retrieve a MODEL node, the function `getmodel` can be applied. This function returns a pointer to the MODEL node of the cell named in the argument of getmodel, and using this pointer the size of the cell may be found.

For example, suppose we want to write a tally-block generator, see Mead and Conway [10], that assembles basic tally cells into a tally block. To do this, we must know the size of the tally cell. Using the function `getmodel`, we can set the variables `xp` and `yp` to the width and height of the tally cell as follows:
MODEL *mp;
...
mp = getmodel ("tallycell");
xp = mp - > xr - mp - > xl;
yp = mp - > yt - mp - > yb;
...

Of course, the tally cell must be 'known' in the ldmc data structure for this to work. If we assume that the tally cell is a predefined (library) cell, we can enter the data of the tally cell in the internal data structure with the function dbread(). This function accepts as arguments the name of the database (library), and a list of cell names. The list must be terminated by the special argument 'NUL'. If we assume that the tally cell is in the library "/usr1/nick/dbase", the complete program may look like this:

```c
#include <stdio.h>
#include <stdldmc.h>
mktally (name, Nins)
char *name;
int Nins;
{
    int xp, yp, i;
    MODEL *mp;
    dbread ("/usr1/nick/dbase", "tallycell", NULL);
    mp = getmodel ("tallycell");
    xp = mp - > xr - mp - > xl;
    yp = mp - > yt - mp - > yb;
    for (i = 0; i < Nins; i++)
        mc "tallycell" t i*xp 0 cy yp i+1
}
```

The #include line is necessary if we want to access the ldmc data structure.

If Nins is 4, the resulting layout will be structured like this:

```
+-------+
|       |
|       |
|       |
|       |
+-------+
```

In general, the correct repetition distance is not equal to the size of the cell; its terminals need not touch the bounding box.
Thus, we might want to determine the pitch of a cell from the position of its terminals. To do this, we can use the mgetterm function.

For example, suppose we want to determine the distance between the terminals of the tallycell, and use this as the measure of the cell pitch. The following piece of code is then suitable.

```c
TERMVEC *tv1, *tv2;
int xp, yp;

tv1 = mgetterm ("%1", 0, "tallycell");
tv2 = mgetterm ("%r", 0, "tallycell");
xp = tv2[0] -> xr - tv1[0] -> xl;
tv1 = mgetterm ("%b", 0, "tallycell");
tv2 = mgetterm ("%t", 0, "tallycell");
yp = tv2[0] -> yt - tv1[0] -> yb;
```

The function mgetterm returns a pointer to a vector of TERM nodes of the tallycell. Which TERM nodes are in this vector is specified by the first two arguments. The lines beginning with `xp` and `yp` set these variables to contain the distance between the outer sides of the terminals denoted by the first elements of `tv1` and `tv2`. The check that these terminals are actually opposite to each other, is neglected.

In general, the function mgetterm can return information about a several terminals of the cell specified in its third argument. The first argument can specify the name of the terminal, e.g.

```c
mgetterm ("l_vdd", ...);
```

or can be some form of a "wildcard" specification, e.g.

```c
mgetterm ("%lr", ...);
```

denoting all terminals at the left or right side, or

```c
mgetterm ("%rp", ...);
```

denoting all terminals at the right side in polysilicon.

The second argument of mgetterm specifies the index member of the TERM nodes, and is only of importance for terminals with a non-zero repetition count. For example, for the following term statement,

```c
term np 0 4 0 4 1_in cy dy ny
```

`ny + 1` TERM nodes are created, with their index member running from 0 to `ny` inclusive.

The return value of mgetterm points to a vector of pointers to TERM nodes, the last member of this vector is the NULL pointer. A pictorial representation of this is as follows:
The ldmc library also contains a function to retrieve the information about the subcells of a cell.

### 3.3 High level data structure access.

These functions also return information that is present in the data structure, but do some interpretation of it first. They combine information found at different places, and extract just the information that is wanted. A good example of a function in this class is the `pitch` function. This function determines the pitch of a cell, it accepts two string parameters; a `key` and a `cell name`. If `key` is "x", the x-pitch is determined from the terminals of the cell, and if the cell doesn't have opposing terminals on both its left and right side, its size is used. Similarly for the y-direction. In addition, a 't' or an 's' can be appended to `key`, respectively specifying that the terminals must be present or should be ignored.

In the preceding example, we could have used the pitch function as follows:

```c
xp = pitch("x","tallycell");
yp = pitch("y","tallycell");
```

Another very useful function in this group of library functions is the `igetterm` function, which calculates the "instantiated coordinates" of a terminal. With these coordinates we mean the coordinates obtained by applying the transformation of a cell to the coordinates of the terminals of the cell.

### 3.4 Assembly functions.

An example of an assembly function in the ldmc library is the `array` function. This function generates the ldm primitives defining a new cell to be an array of some other cell. The formal parameters of the `array` function are the name of the new cell to be made, the name of the cell that is instantiated in the array, its orientation, and a repetition count in the X- and Y-direction. The repetition distance is not a parameter of this function, but the pitch function is called by `array` internally.

Also, the array will have terminals around its perimeter there where the subcell has terminals. This is done by calling the `tup` function. This function will place terminals in a cell over
selected terminals of instances of its subcells, and is also callable from the application directly. See the mkplax example in section 4 for an application of the array function.

Another function is abut, generating the ldm statements so that a cell is abutted to a specified side of an instance of another cell, with the terminals of both cells connected. Box statements are generated that connect pairs of terminals of both cells that are too far apart.

The abut function accepts 4 string arguments, one being optional that we will not discuss here. The second parameter names the instance of the cell to which the new cell named by the third parameter will be abutted. The first parameter is a key controlling the abutment, its value can be:

"l" - abut the new cell to the left side of instance
"r" - abut the new cell to the right side of instance
"b" - abut the new cell to the bottom side of instance
"t" - abut the new cell to the top side of instance

Optionally, key may contain an o, specifying that the terminals of the instance and the new cell must overlap as far as possible.

The abutment is such that terminals of the same layer of both subcells touch each other. However, it is not required that the terminals of both subcells pitch-match perfectly. For example, the width of two corresponding terminals need not be equal:

![Diagram](image)

Also, the abut function will generate connecting wires if two corresponding terminals do not touch, and the subcells can't be placed closer to each other, because some other terminals already touch each other:

![Diagram](image)

Furthermore, not all terminals of one cell need to be connected to a terminal of the other cell:
Several terminals of one cell may be connected to one terminal of the other cell:

Combining these aspects, we may say that any terminal in some specific layer of one cell can be connected to zero or more terminals in the same layer of the other cell. `abut` will add connecting wires to connect terminals that do not touch, but do 'see each other'. What `abut` always will prevent is to place the new cell such that terminals that are not properly aligned only partly touch each other, or such that terminals in different layers touch each other. Thus, the following two cells can't be abutted, and `abut` will issue an error message.

If the overlap option is specified, (i.e. if `key` contains an `o`) terminals will be overlapped as far as possible:
For an example of the usage of the `abut` function, we refer to section 4.

3.5 Connection and routing functions.

The functions in this class make it possible to automatically generate many of the "glue" boxes that connect the terminals of the different instances of the cells making up a larger assembly, without a precise knowledge of the exact locations of the terminals. Instead, the designer specifies how the terminals are to be connected. The coordinates of these terminals are then automatically computed, taking into account the transformation of the instances. The terminals to be connected are in principle specified by their name, but the same abbreviations as for `mgetterm` exist.

Often, the `connect` function is powerful enough to perform the desired routing. This function can generate either I-shape (straight), L-shape (with one bend) or U-shape (with two bends) connections. Some examples of the usage of `connect` can be found in the PLA generator example.

The connection paths of L- and U-shape connections are uniquely determined because terminals are associated with a side, as specified by the first character of the terminal name. A connection wire always leaves the terminal in a direction corresponding to its side. For example, a left terminal `l` may be
connected to a bottom terminal b by two boxes; they form a wire that runs horizontally (to the left) from l and then upwards to b.

The separation s between the 'bars' of two U-connections adapts automatically to the minimum separation allowed by the design rules. This is also the default minimum length l of a 'leg' of an U-connection. However, if it is more appropriate to specify the value of upos in the above picture or if the default separation is not OK, one can use the function uconnect.

Except for what is specified above, the connection functions are further very naive about design rules. They assume that a straight channel exists from the terminal to the bounding box of the cell, in which the connection wire does not violate design rules. The connection wire is always in the same layer as the terminals it connects, it is currently not possible to automatically generate via's.

Another function is the river function, executing a simple multilayer river routing algorithm.

3.6 Ldmc kernel functions.

The ldmc kernel functions are the workhorses of the system. They miss an application directed 'cosmetic' interface when compared to the functions mentioned earlier, but allow more freedom in adjusting some options directing the result of their invocations. They may therefore be used when the standard functions just don't fit, either by calling them directly or by defining a new function in terms of the kernel functions.

As an example, the abut function doesn't have the possibility to specify a repetition of the cell that is abutted, but one parameter of the corresponding ll_abut kernel function is a pointer to an ll_abut_opts options structure. Among the members of this structure are 4 integers that specify the repetition, but normally they are zero. They can however be used to define a new abut function in terms of the ll_abut function as follows:

```c
#include "ll_abut.h"
/*
* rabut - abut with repetition
*/
INST * rabut (key, instance, cell, inst_name, nx, ny) char *key, *instance, *cell, *inst_name; int nx, ny;
{

    /* this is declared in file "ll_abut.h" */
    static struct ll_abut_opts opts = DEFAULT_ABUT_OPTS;

    opts.nx = nx;
    opts.ny = ny;
    if (nx > 0) opts.dx = pitch ("x", cell);
    if (ny > 0) opts.dy = pitch ("y", cell);

    return (ll_abut (key, instance, cell, inst_name, &opts));
}
```

1.43
3.7 Miscellaneous functions.

The ldmc library contains several other functions not mentioned above. For example, there are functions to read a database or an ldm file, to change the current output stream, etc.

4. An illustrative example.

In fig. 2, an example is given of a simple PLA generator implemented with the ldmc system.

This PLA generator constructs a PLA from some basic cells, which can be found in a database. They are read into the data structure with the call for \texttt{dbread}. Then comes the ldm primitive for the begin of a cell, whose name is a parameter of the \texttt{mkpla} function. Next, the andplane and orplane are built by the corresponding functions. Then, an array of input cells, output cells, pull-up cells for and- and or-plane, connect cells and ground cells are made with the calls for the \texttt{array} function. The pull-up array for the orplane is made with the same leaf cells as the pull-up array for the andplane, but they are mirrored about the y-axis and then rotated 90 degrees counter-clockwise.

Then, the subcells are placed, beginning with the andplane. An array of ground-cells is abutted to the bottom of the andplane with the \texttt{abut} function. The other cells are placed similarly. The invocations of \texttt{connect} make the wiring around the pla, here resulting in four L-shape connections and one U-shape connection. Finally, the ldm primitive \texttt{me} means "end of cell".

The \texttt{andplane} function is shown in fig. 3. The code for the orplane function is not shown, since this is similar to the code for the andplane function. The andplane function has the name of the cell and the number of inputs and minterms, as well as an array describing the personality of the plane as parameters. First, the personality of the plane is read from the array \texttt{prog}, and the switch statement subsequently assigns to the variable \texttt{cell} the name of a leaf cell depending on the character found in \texttt{prog}. This variable is then used in the ldm \texttt{me} primitive. This statement means: "Place the cell denoted by the variable \texttt{cell} translated over a distance $x*dx$ in the x-direction and over a distance $y*dy$ in the y-direction." $x$ and $y$ are the loop variables of the two nested for-loops, and $dx$ and $dy$ are set to $x$- and $y$-pitch of the cell in the declaration part of the function.

The topology of the PLA and its connections is shown in fig. 4. Of course, this is only a skeleton of a very simple PLA generator, but more features are added easily. For example, in a production version of a PLA generator you must provide the code to insert extra groundlines in the andplane and the orplane when the number of minterms or the number of outputs becomes too high, etc. We have omitted these details in favor of the conciseness of the example.
```c
#include <stdio.h>
#include <stdlib.h>

char *modlist[] = {
};

mkpla (name, nins, nterms, nouts, dbase, prog)
char *name, *dbase;
int nins, nterms, nouts;
char *prog[];
{
dbread (dbase, modlist);
/*
* begin layout, make building blocks first
*/
ms (name)    /* begin layout */
andplane ("andplane", nins, nterms, stdin);
orplane ("orplane", nouts, nterms, stdin);
array ("inputs", "plain", 0, 0, nins - 1, 0);
array ("outputs", "plaout", 'x', 0, (nouts/2)-1, 0);
array ("andplups", "plaplp", 0, 0, 0, nterms-1);
array ("orplups", "plaplp", 'y', 90, nouts-1, 0);
array ("connect", "plaaoc", 0, 0, 0, (nterms/2-1));
array ("ground", "plagnd", 0, 0, nins - 1, 0);
/*
* actually place building blocks
* place andplane first, rest grows like a crystal
*/
mc "andplane"
abut ("b", "andplane", "ground", "")
abut ("b", "ground", "inputs", "")
abut ("l", "andplane", "andplups", "")
abut ("r", "andplane", "connect", "")
abut ("r", "connect", "orplane", "")
abut ("b", "orplane", "outputs", ")
abut ("t", "orplane", "orplups", ")
/*
* make connections
*/
connect ("l_vdd", 0, "andplups", "l_vdd", 0, "orplups")
connect ("l_vdd", 0, "inputs", "b_vdd", 0, "andplups")
connect ("r_vss", 0, "inputs", "b_vss", 0, "connect")
connect ("l_vss", 0, "outputs", "b_vss", 0, "connect")
connect ("r_vdd", 0, "orplups", "r_vdd", 0, "outputs")
me /* end of layout */
}

fig. 2 - PLA generator function.

5. Discussion.
The system as described is implemented (the library source consists of about 6000 lines of C-code) and proves to be a useful tool for implementing layout generators. It raises the level of abstraction on which the generators describe the layout they produce, thereby minimizing the effort to implement them. At the

1.45
andplane (name, nins, nterms, prog)
char *name;
int *nins, nterms;
char *prog[];
{
    int x, dx = pitch("x", "placl0");
    int y, dy = pitch("y", "placl0");
    char tabs[BUFSIZ], *cell;
    ms (name) /* begin andplane */
    for (y = 0; y < nterms; y++) /* for all minterms (rows) */
        for (x = 0; x < nins; x++) /* for all columns */
            switch(prog[y][x]){
                case '0': cell = "placr"; break;
                case '1': cell = "plact"; break;
                default : cell = "plaln"; break;
            }
    mc (cell) t x*dx y*dy
    if (x==0 && y==0) /* if lower-left cell */
        tup("%", 0, cell, "&", dx, nins-1, dy, nterms-1);
}
/* end of andplane */

fig. 3 - PLA generator andplane function.

Fig. 4 - Topology and connections of generated PLA.

same time, their design-rule independency as well as their
modularity (a software engineering criterion) is maximized. The
emphasis is on powerful high-level functions, but supplemented
with lower-level functions that are useful in cases where the
generalities to which the former functions are tuned, aren't
applicable. It is currently being used to implement several
layout generators, under which a RAM generator and a sequencer
generator.

The most notable disadvantages of the ldmc system are the somewhat
larger execution time of the generators and somewhat larger
executables. This is the price to be paid for some functional
overhead that is necessary to cover as many generalities as possible.

5.1 Comparable systems.

Lubrick [2] is a pascal implementation of a silicon assembler. The output of a lubrick program is a (textual) Lucie layout description. Its functions also include composition and connection functions, but lack the balance between high-level and low-level functions characterizing ldmc. Terminals (pins in lubrick terminology) are also oriented. The data structure underlying lubrick is not supported by access functions. Since lubrick runs under the Multics operating system, where link-editing is done dynamically, lubrick programs occupy only little disk space.

The ic [3] language, like ldmc, is also an hybrid language. Circuit descriptions in the ic language are embedded in C programs, and a preprocessor converts ic to C by placing the embedded ic statements in standard C printf statements. When the resulting C programs are compiled and executed, they output ic text.

Generators written in ic need not calculate the exact location of every element in the layout, since the ic interpreter can resolve geometric constraints. The ic language is not supported by a library of silicon assembly functions, and no data structure describing the layout is maintained. Therefore, generators written in ic must perhaps do some explicit bookkeeping.

5.2 Future extensions.

Some extensions to the ldmc system might perhaps be useful, the major ones now thought of being:

1. to support terminals that have a multiple orientation (i.e. with a straight channel in more than one direction). This may, however, have repercussions on the connect function.

2. a combination of abutment and routing as in tiles [11] allowing instances of cells with non-pitch matching terminals really to be placed as close as possible, taking into account all design rules and the complete boundary of the cells.

3. the implementation of more general channel routing procedures, allowing connections to change layer, etc.

A solution to above points might implicate unacceptable sizes of the executables, because the object code implementing the added functionality is duplicated in every layout generator. It is then perhaps appropriate to put all ldmc functionality in a single program with a command interpreter on top of it that invokes the different functions. Any time a generator is executed, the ldmc interpreter is also started. A pipe is set up between these processes through which the generator sends commands and, on request of the generator, the interpreter sends back information about the layout that is produced.

1.47
6. References.


At the previous general ICD meeting it was decided that a Working Group would study and prepare a standard for the representation of electronic network structure related data in the ICD-system. Members from ICS/ICN, TH-Delft and TH-Eindhoven participated in this Working Group. Remarks and suggestions were received from British Telecom and TH-Twente. Intermediate results and meeting notes were sent to all interested parties.

The current state of work on the proposal and some remarks on its implementation will be presented.

1. INTRODUCTION

This report tries to establish a common format for describing the structure of an electronic circuit. Traditionally this description is called a net-list. Mind though that the proposed format allows the net-list to be specified in a hierarchical way. The descriptions in this format reside in the general database and are fully controlled by its manager. With this format it should be possible for various applications programs to exchange their data in a convenient way. Of course it can not be expected that this format allows a description generated for a particular simulation program to be processed by an arbitrary other simulator without intervention. We hope that this format at least helps in minimising these problems. We strongly feel that the format is sufficiently general to be used by most application programs in our project that are related to network description, net list extraction or generation, now and in the future.

The disadvantage of the indicated generalness is that application programs still need to have their own evaluation programs for the data stored in the format. In other words: the interpretation of the items in the format is left to the application, the format merely specifies where and how certain data must be stored. Hints and suggestions about how to represent the data are given but not enforced.

This paper was produced as part of Task I of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
We primarily consider descriptions at the circuit and logic level of design. Aspects of lower levels (geometry) should be contained in another view of the database.

This report starts with a section discussing the objects of circuit description. Section 3 states some database aspects of the format. Section 4 defines the general file format and in section 5 and following the details of the format are presented.

2. FUNDAMENTALS OF CIRCUIT DESCRIPTION

2.1 Types of description

A circuit has both structural and behavioural aspects, together they form the complete circuit description.

Structure description is intuitively more simple and better understood. Also it can be said to be pretty application independent. On the contrary behavioural description is highly application dependent and therefore it will be much harder to find generally useful description formalisms.

The network description format focuses on structure description. A place for putting in the behaviour description is foreseen but its format will not be defined due to the diversity of application programs such as simulators.

2.2 A structural circuit model

Traditionally an electronic circuit is thought to be composed of a number of elementary components such as resistors, transistors, capacitors, inductors and current and voltages sources. The typical behaviour of a circuit is completely determined by the behaviour of the constituting circuit elements and the structure of the network, i.e. the interconnection of the various elements. In an abstract, mathematical way we can describe a circuit by the triple:

\[ C = (T, I, W) \]

where

- \( T \) is the set of terminal ports of the circuit \( C \),
- \( I \) is a set of circuit elements, and
- \( W \) is a wiring specification.

A second glance at this abstract model leads to a further refinement. Note that \( I \) generally contains many elements of the same type, perhaps only differing in the particular value assigned to the element: a resistor of 100kΩ and 2 of
10kOhm are all of type resistor but with different resistance value. Transistors commonly have many parameters, but all may be classified by a few types, e.g. bipolar transistors and MOS-transistors.

Let us assume that any type of circuit element is characterised by a number of fixed properties (the number of terminal ports) and some properties that are variable (the parameter values). This is not a restrictive assumption because we could introduce parameters that allow the so-called fixed properties to be varied.

Obviously the set I consists of determined elements, which will be called instances of a type. To complete the circuit description we add a component-type list S to it, so C becomes the quadruple:

\[ C = (T, S, I, W) \]

Each element of S serves as a template for deriving instances I. In more detail the elements of S are described by the following categories:

- A set of terminal ports,
- A set of parameters,
- A behaviour description with the parameters as static, i.e. time-independent and the signal values at the ports as dynamic, i.e. time-dependent, variables.

Each element of I specifies:

- The element of S of which this element is an instance. This implicitly defines the terminals of the instance. They are simply inherited.
- The values for parameters of the element of S.

So far nothing has been said of the constituent W. Intuitively W must specify which ports are connected and in what way. The ports that are to be considered are the ports in T and of course the ports of all instances I. It will be shown that interconnection may be specified in various ways in a circuit description language. Here we concentrate on two particular forms that may be called fundamental.

Clearly in W we must be able to express:

1. A port is not connected to any other port at all. We then say it is left open or unused.
2. Two identified ports say p1 and p2 are connected.

To achieve the above requirements we introduce the concept of a wire and state that W is a set of wires.

We define a proper wire to realise a connection between two (no less, no more) distinct ports.

This definition does not imply any physical meaning like a copper wire between two solder pens, or a track on a PCB or an aluminium path on an IC.

Some degenerate cases must be considered, like:

1. A wire is identified but no ports are assigned to it. We will call this a loose wire.

2. A wire is connected to just one port. This will be referred to as a dangling wire.

3. The two ends of a wire are connected to the same port.

Of course the non-proper wires should not appear in a correct circuit description. The reason they are mentioned here is for sake of completeness. The tool for processing a circuit description may regard non-proper wiring as design errors and act accordingly, e.g. by disregarding the wire and/or by reporting back to the user. In the rest of this section we understand all wires to be proper.

It is important to notice that the ports involved in the wires must all be uniquely identifiable. Here we are not interested in how this is achieved in a typical description language, so for convenience we adopt the convention that a port "p" belonging to the terminal set of the instance "i" is denoted by "i.p". In fact, in practice we often encounter this very same qualified identification scheme.

Another view on connection leads us to the concept of nets. Informally a net can be defined as a collection of wires that pairwise share a common port. More formally we use the notion of equivalence relation to define a net. When we regard wiring as a functional mapping from two ports to an unordered pair of ports, then this mapping is an equivalence relation and precisely those wires that pairwise share a port all belong to the same equivalence class. There are as many nets as there are equivalence classes.

Whether one defines interconnection by means of wires or by nets heavily depends on the application. Somehow information is lost when moving from a wire...
specification towards a net list. An example will make this clear. Suppose in some application one associates with each wire a resistance value, it then becomes impossible to define an equivalent property for a net. One may argue that when a wire has resistance (or some other property) it is likely to be replaced by a lumped model, thus again regarding the wire as ideal which also allows the net oriented approach. Still much of these decisions depend on the level of design one is concerned with. Surely at the lay-out level connection wires do have properties that cannot be incorporated in a net characteristic, think about sizes and type of connection (metal, polysilicon). Whereas at a logic level wires will even be abstractions from the electrical level.

In the format we allow both types of wire specification. Basically the format part that describes connection is wire oriented: each wire is described by two terminal ports. A name may be assigned to a wire; by assigning equal names to different wires a net can be specified.

2.3 Hierarchy

In the previous section we studied the basic elements of circuit description. We modelled a circuit as the quadruple (T, S, I, W), with T containing the circuit’s external ports, S the type of components used, I the list of instances of S and W an interconnection specification. To these we now add the concept of hierarchy.

Today any design of considerable complexity will exploit hierarchy. The main reasons for this may be summarised by:

- allow the design to be partitioned into a number of manageable pieces
- allow for many levels of abstraction
- allow a toolmaker to make efficient use of computer resources

In circuit description hierarchy means allowing each type of circuit element to be a complete circuit on its own. Such a circuit type may be instanced just like the primitive components. Thus the quadruple (T, S, I, W) defines a circuit type, not an instance! It is quite natural within this context to also let a circuit have its own set of parameters, which leads to following extended definition of circuit description:
C = (T, P, S, I, W) 
where T, I, and W are as in (T, S, I, W) 
but 
P is a set of parameters of the circuit type C, 
S is a set of circuit types, either primitive 
component types or compound component types. 
Clearly this definition is recursive in nature: compound elements of S are 
themselves defined by a quintuple.

For the lowest level circuit types, which we will call leaves, we have, 
leaf-C = (T, P, S, I, W) 
with I = W = 0 and S replaced by 
a behavioural description B, 
leaf-C = (T, P, B) 
The highest level of the hierarchy may be defined as a closed system often 
called the root, which is denoted by the triple (S, I, W). A root system does 
not have an interface therefore the sets T and P are discarded.

With the above definitions and considerations in mind we now proceed with the 
definition of a format for network description.

3. DATA-BASE ASPECTS

Each circuit template (or type) description forms one item for the data-base. 
We propose to implement it as a UNIX directory. So all templates are directly 
controlled by the data-base manager and all relations between the templates 
must be supported by the data-base. Typically the data-base manager controls 
the hierarchy imposed on the objects via instance descriptions.

Each directory describing a network template will contain four files:

1. The network interface file.
2. The network call file.
3. The network wire file.
4. The network behaviour file. This file is optional for structured 
templates, i.e. templates with a call and wire file.

The contents of these files is handled as a whole. A data-base application has 
access to routines to store and retrieve the contents of one of the above 
files. For this, a data-base interface need be defined. The interface will be 
described by a C structure type. This report does not specify the data-base 
routines and interface implied by the format.
4. GENERAL FILE FORMAT

Each file defined by the network description format is a sequential ASCII-file containing only printable characters in the range from ASCII code 32 until and including 126 and some control characters that are meta-characters defined by this format.

A user wishing to include control characters in the file must define his own rule for escaping them, preferably he uses an approach consistent with UNIX and C.

A file consists of one or more items called records. These records are variable length character lines, separated by newlines in the C sense. The newline is a meta-character of the format and can not be used otherwise.

\[
<\text{format-file}> ::= <\text{record}> \ ( <\text{record-separator}> <\text{record}> ) \ .
\]

\[
<\text{record-separator}> ::= <\text{NEWLINE}> .
\]

One such logical record can be spread out on a number of physical lines mainly for readability or to comply with line-length constraints by escaping the newline meta-character with a backslash. To include a backslash in a field it must be written twice.

Each record consists of a fixed number of fields. The number of fields is defined for each file of the format. The purpose, contents and intended interpretation of a field depends on its position in the record and the file it appears in. Fields are separated by one or more white space meta-characters. A white space character is either a blank or a tab. To allow white space to appear in a field, a field may be quoted by enclosing it in a pair of double quote characters. Double quotes occurring in a quoted field must be preceded with a backslash character.

\[
<\text{record}> ::= <\text{field}> \ ( <\text{field-separator}> <\text{field}> ) .
\]

\[
<\text{field-separator}> ::= <\text{white-space}> .
\]

\[
<\text{white-space}> ::= <\text{BLANK}> | <\text{TAB}> .
\]

\[
<\text{field}> ::= <\text{normal-field}> | <\text{quoted-field}> .
\]

\[
<\text{normal-field}> ::= <\text{sequence-of-l-or-more-characters}> .
\]

\[
<\text{quoted-field}> ::= <\text{DOUBLE-QUOTE}> <\text{normal-field}> <\text{DOUBLE-QUOTE}> .
\]

\[
<\text{empty-field}> ::= <\text{DOUBLE-QUOTE}> <\text{DOUBLE-QUOTE}> .
\]

Two consecutive double quote characters indicate that a field contains nothing or is left blank: an empty field.
Of course normal fields may always be quoted. Note that even then field separators are required.

4.1 Contents of fields

Fields are intended to contain a single item of information. We visualise the following sorts:

- names, to identify some object, typically introduced by a user of an application program
- keywords, typically defined by the application program
- expressions, to denote some value, typically introduced by a user
- manifest constants, denoting some value
- functions, to denote some action, highly application dependent

The representation of most of these items is not enforced by the format. We strongly suggest that implementors of applications defining a particular representation mimic the UNIX/C conventions. Please report chosen representations to all other interest groups.

The following list gives some suggestions:

- names: sequence of letters and digits, starting with a letter. An extra character for readability is for instance the underscore.
- keywords: small lowercase names of letters only.
- expressions: follow the C syntax for expressions.
- constants: represent numbers, characters and character strings as in C.
- functions: no suggestions.

5. INTERFACE FILE FORMAT

The interface file describes the arguments of the network. Within the argument list we explicitly distinguish two classes of arguments, namely terminal ports and parameters, because the first are physically present in an instanced network whereas the second are not. Each of the parameters or ports is described by one record. The format of the record is:

name class ctype default constructor attribute
Description of the fields:

name: The name of the formal argument. All the names of the same class must be unique.

class: We distinguish two classes of formals, parameters and ports denoted by the keywords "parameter" resp. "terminal".

ctype: Each class of formals can have different types.

- the type of a parameter is denoted by one of the keywords "int", "float", "double", "char" or "bool". These are all unstructured types with their intended meaning borrowed from the languages C and Pascal. Moreover a subrange type may be specified by two constant values of the types indicated separated by a comma.

\[ <\text{subrange}> ::= [ <\text{constant}> ] ",", [ <\text{constant}> ] \]

Not specifying the constant to the left of the comma, implies the minimum value defined by the type. Likewise, leaving out the right constant implies the maximum value of the type. Parameter typing allows for checking the actual parameter value when the template is instanced.

- types of ports may be dependent on the particular simulator used. We suggest to use keywords, e.g. "electr", "bool", "logic".

default: It is convenient to have default values for arguments. For ports this can be interpreted as tying the port to a particular signal source, e.g. logic gate inputs that are left not connected assume a logical high value. For parameters the purpose should be clear; here we demand the value to be of the corresponding type.

constructor: this field specifies the structure of the formal argument by a function script. Here we think for instance of a vector constructor, stating that a port consists of many identical subports that can be indexed.

attribute: this field is introduced for specifying properties of terminal ports other than the type of signal values that pass through it. An attribute could be a list of keywords. Examples of useful
attribute keywords are: "input", "output", "inout", "notused". At the moment we think that attributes for parameters are not meaningful.

6. NETWORK CALL FILE FORMAT

This file specifies the sub networks constituting the template circuit. A sub network is called an instance. Components of an instance are its name, a possible constructor, a reference to the template it is derived of, actual parameter values and a guard.

The network call file contains records with the following fields:

- **instance**
- **constructor**
- **object**
- **parameters**
- **guard**

A record is subdivided into the fields:

- **instance**: The name of the instance. All instance names in the file must be unique.

- **constructor**: The constructor is a function script that specifies a structure of a particular network instance. For instance an application could define this field as "array[1..p]", meaning that an array of "p" instances of the object are identified by the instance name. "p" being a parameter specified in the interface file. The constructor has a corresponding selector in the wire file.

- **object**: This field specifies the name of the network template called.

- **parameters**: Every instance of a network template, say "T", may assign certain values to the parameters of "T". These actual parameter values could be specified by means of expressions, containing constants and parameters as operands. We recommend to use a named association for formal and actual parameters.

- **guard**: The guard field is introduced in order to specify, whether the network call has to be skipped or not. Typically this will be specified by a boolean expression, possibly with formal parameters as the non-constant operands.
7. NETWORK WIRE FILE FORMAT

The wire file gives the specification of the interconnection of the ports of the instances and of ports of the template being described. Ports of instances are identified with a so-called qualifier. A qualifier consists of the instance name with its selector and the port name with its selector. One wire record is meant to specify one point-to-point (read port-to-port) connection. It is allowed to introduce an alias for a port by only specifying one half of the connection. The alias or net-name may also appear instead of a port in other records.

A wire record contains the fields:

net instance selector port selector
instance selector port selector
constructor guard

explanation of the format:

net: The name for the connection. When more than one record has the same name, this can be interpreted as a net. Nets get a certain structure as described by the constructor.

instance: The name of the instance to which the port belongs. May be empty for ports of the template.

port: The name of the port.

selector: The selector for the instance respectively for the port or net. In general it will be a function script.

constructor: This field specifies the sort of construction between two ports. It also determines the structure of "net".

guard: Here again we need a guard field to be able to specify the presence of the wire.

8. NETWORK BEHAVIOUR FILE FORMAT

Due to the variety of application programs with different capabilities no format for behaviour description can be defined. The contents of the behaviour file must be completely defined by the application.
9. IMPLEMENTATION

In order to interface with the database, a complete and versatile library of C routines is implemented. Basic functions are open_model to access all files of one template, several read and write functions, and queries to find all models in a library. The routines return structures and lists of structures, in accordance with the file definition, however independent of the specific format of the file. Messages are provided for errors in the file format, memory allocation errors and so on.
THE EINDHOVEN SCHEMATIC EDITOR

A. Lodder
M. van Stiphout
J.T.J van Eijndhoven

ICS
Eindhoven University of Technology

ABSTRACT

This paper presents an overview on the schematic editor. The purpose of the program is defining networks, by interactively drawing a hierarchically structured network. The output is in accordance with the ICD network standard, interfacing with several simulation programs, and layout generation tools. In order to cope with the hierarchy, for each module (or template) a contents is defined, as well as a representation. The contents section consists of instances of other modules, connections, and expressions for actual parameters of the instances. The representation consists of a symbol, terminals and a formal parameter declaration. Furthermore the schematic editor is capable of handling busses in a very flexible way, handling mixed analog and digital circuits, and features a multilevel undo command.

This paper has been produced in the context of TASK 4 of the project MR-09 which was supported by the Commission of the EEC, under the 3744/81 Microelectronics program. Partial support by the Ministry of Economics Affairs of the Netherlands under the NELSIS program is hereby also acknowledged.
1. Introduction.

The Eindhoven SCHematic Editor ESCHER is an interactive graphical editor suitable for entering or changing information in the ICD network database. Other programs like simulators and automatic placers may use this information as an input. A few pictures illustrating some of ESCHER's capabilities, can be found in the colour section of this book.

2. Templates.

A basic unit of information is the model or template consisting of an inner side, the circuit, and an outer side, the representation. The circuit is build from instances and terminals and the connections between these and the terminals of the instances. The circuit is completed by assigning a value to the actual parameters of the instances. Terminal and wires have their own type, electrical or signal, used by the Piecewise Linear Simulator. A Terminal has attributes like input, output, inout or unused and there is the choice between pull up, pull down, tristate or other. Terminals and instances have their own unique name. Wires may be combined together in busses, each wire having his own number. Low and high bound of the wire is user selectable. Terminals also have their own width, in this case the range is from 0 to the high bound - 1. If necessary a connection between a terminal and a wire is accompanied by an offset which is to be added to the terminal number to get the number of the wire with which it is connected.

The representation consists of the symbol, the terminals of the template, the formal parameters and the surrounding box which determines the size of an instance of the template. Symbol elements are straight lines, circles or arcs.

3. Hierarchy.

Although the editor is meant to deal with a hierarchical description of a network the editor will not work hierarchical. Models are changed one on a time. By selecting a certain model as root a complete network is defined. This is done by programs reading the network database, not by ESCHER.


To end an edit of a template and start the edit of another one a change template command is provided. Change of the the current template involves an automatic save of the graphical information of the template on disk and will establish if possible a complete network description of the template in the

1.62
network database. The dump of the graphical information will be used when re-editing the model. Fast change of changing the current template is provided by selecting an instance of which the template will become object of the editor. With every change the name of the old template is saved on a template stack. It is possible to walk back in this stack. With these features it is possible to design top down and bottom up as well.

5. Manipulations.

Editing is manipulating instances, actual parameters of these instances, wires and terminals of the circuit and symbol elements, terminals, surrounding box and formal parameters of the representation. The manipulations are add, delete, change, move, copy, rotate and mirror. Not every manipulation goes with every kind of object. In detail:

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>delete</th>
<th>change</th>
<th>move</th>
<th>copy</th>
<th>rotate</th>
<th>mirror</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instances</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Wires</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Terminals</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual parameters</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formal parameters</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>surrounding box</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbol elements</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Add and copy of instances and terminals will automatically generate an unique name for it. Change of terminals and instances will only involve the name of it.


Alphanumeric input is provided by means of a line editor. A default value may be edited using control characters to perform special functions like: toggle insert/overwrite, insert and delete character, delete rest of or complete line and undo last input.
7. Abutment.

Instances can be abutted. Properly matching of abutted terminals is checked. This check concerns terminal type, width and attributes, so combination of 2 inputs or outputs or the combination of pull up and pull down will not occur. When abutted the manipulations move, copy, rotate, and mirror will be done on the set of abutted instances as if it was a unit.

8. Grid.

To fasten input and improve the quality of a drawing all input is converted to points on a grid. The grid may be visible or invisible.

9. Window handler.

ESCHER provides fast and flexible window management functions including zoom in and zoom out with a factor of 0.7 and a function which will adjust the window in such a way the total circuit or representation will be visible. The screen contains a window context area in which all instances are drawn as solid boxes and the current window as a solid box in a different colour. Selecting a coordinate in this context area will cause a window centre function. All these input will not interrupt the currently active edit command. The locator echo position is adjusted as well and clipped against the new window if necessary. Another window function is window box which needs 2 coordinates as the corners of a box which will be the new window.

10. Split.

A split function is provided to split the circuit or representation by moving all the instances, terminals, wires and symbol elements over a certain distance to the left, right, up or down so giving room for more wires and instances. Wires across this split will be automatically lengthened.

11. Parameters.

A template has formal parameters which has attributes as kind: real, integer or boolean, range and default value. The user may give these parameters an actual value if the template is called, each instance his own actual value. The actual parameter value may be a complex expression consisting of real, integer and boolean values and variables, the last being the names of formal parameters of the template. The expression may contain the names of common used mathematical functions like \( \exp() \), \( \log() \) and \( \sin() \). The editor will assign the
default value of the formal if an new instance is made.

12. Verify.

ESCHER has the possibilities to determine the consistency of the network information. This verify will involve dangling end and short cuts of wires, syntax checking of formal parameters and values of actual parameters and it will compare the names and attributes of the template terminals as part of the circuit and the representation.

13. View.

The circuit of a template will show only the outside of the template of an instance. With the view command it is possible to give a fast overview of its inside. No editing of this circuit will be allowed. While editing the representation of a template the view command will give a view of the circuit of the template.


Every command is kept on the undo stack which has a user defined depth. By pressing undo the command on top of the stack will be undone. The undo stack is cleared if some interaction with the disk implying a save operation has taken place.

15. Command parsing.

A command consists of the selection of the command in a menu and giving 1, 2 or 3 coordinates as parameters. Of course this number is command dependent. After executing the command most commands will stay active so only the coordinates have to be entered. Manipulation commands work on an active item. While editing a circuit these items are instance, wire and terminal. A representation is build from the items terminal, line, circle, and arc. Items stay active until an other item is selected. Commands needing no arguments like zoom, window center, selecting an item and toggle grid may be combined with the normal command input without interrupting it. Selecting a command instead of a coordinate will result in a reject of the current input and the new command will become active.
16. **Info.**

With the info command information of instances is obtained. This information consists of its name, the name and attributes of terminals and the values of the actual parameters. While editing the representation it will show the formal parameters and the names and attributes of the terminals.

17. **Libraries.**

Templates are determined by their name and an UNIX pathname. An entry of an instance in the network database will also involve the pathname of the original template. **ESCHER** has mechanisms to supply a flexible and consistent use of these pathnames. First of all when entering a model name of a template which will become editor object the directories in the user supplied library path environment variable are searched for this name. If it not found then a new template is created in the first directory of the library path. If the name of a template is given which will be the origin of an instance the same library path is searched. It is possible to replace an instance of a model by an instance of another model. This is done by an **ESCHER** program option. In the same way it is possible to replace all the models from a library by the models of another library with the same name. Replacement can be hard or soft: Hard replacement will cause a deletion of the old instance if the new one doesn’t fit. Soft replacement will cause to flag an error and a reject of the edit of the template. Hard replacement is also done if the template has instances of which the instance time is older than the last modification time of the representation of the instance. If the new one doesn’t fit the old one will be removed. Fitting is rather intelligent: All changes including adding and deleting terminals or formal parameters, moving terminals or changing their names, enlarging the size of the representation will cause a replacement if the new instance will fit in the circuit. Actual parameters will be deleted if a formal was deleted or a new one will be added if a new formal was formed.

18. **Errors.**

A line on the screen is dedicated to show error messages. An error will stay on screen for at least 2 seconds. If within this time a new error occurs the editor will pause for a short time. Errors will not be shown longer than 10 seconds. After this period it is replaced by the name of the current template.
19. **Menu's.**

The commands are gathered in 2 menu's, 1 visible at a time. The first menu contains the editing commands like add and delete, the other has the commands to do disk access as change template and save. Always visible is the status menu to change terminal attributes and wire ranges. Selecting a status command will not interrupt the current command input.

20. **Save.**

The user may save the graphical information of the template on disk. This can be done with the same pathname and model name it is called or with a new name and pathname. If the network information is consistent also the network database is updated with the new information. If it is not the current network files of the model are cleared thus providing always consistent and actual network information.

21. **Plot.**

A drawing of the circuit or representation of the template can be made by ESCHER. It is in fact a call to a total independent program so drawings may also be made without the use of the editor. Plots may be spooled. When called from ESCHER error messages will appear on the screen.

22. **Options.**

The user may control certain options by means of UNIX flags. A brief description:

- Autosave at certain time intervals
- Depth of undo stack.
- Interchange of meaning of mirror-x and mirror-y commands.
- Allowance of drawing wires across instances.
- Appearance of line edit menu when necessary.
- Default terminal names.
- No old name as default when editing names.
- Options for the plot program.
- Hard and/or soft replacement of certain models.


The programs understands the use of several specific UNIX environment variables. For example one is used to give the name of the graphical device, another contains the library path. So the user can configure the program to his needs.

24. Portability.

*ESCHER* is written in C and has a LINT approval on portability. It uses the standard UNIX memory allocation functions. So it will run on every UNIX system. The use of the line editor which only uses functions from the graphics routines also improves portability. The network database is designed to be portable to other systems. A transfer of graphical information will only work if destination is a machine which has pointers implemented as 32 bit words.
Designer-foundry interface.

F.D. Smedes, P. B. Duin, W. Schouten, Chr. van Veenendaal,
ICD, Enschede (The Netherlands).

and

L. Spaanenburg.
Twente University of Technology,
Enschede (The Netherlands).

Abstract. The designer foundry interface is described. Outlined are both the introduction of process data into a design system as well as the verification of these data with respect to completeness and consistency. From the influence of process data in a CAD system it is suggested, how acceptance procedures between designer and foundry can be constituted to ensure a smooth and secure transition of designs.

1. Introduction.

The ICD system is to be used by a large number of designers at many locations, in such a way that their designs can be produced at a variety of silicon foundries. In the first instance ICD wishes to interface with silicon foundries, who are willing to process chips for outside customers. It is believed that others will follow in the course of time.

A basic requirement for the ICD system is that a clear interface between silicon foundry and design house is defined. This interface is of importance at:

- the specification of the fabrication data by the foundry, which serves for
- the construction of a fabricator database at the CAD house, which allows the CAD system
- the production of design data, that facilitates the foundry to fabricate the specified circuit.

In other words there is not a single foundry interface, but rather a two-some: a receiving and a giving end. At the receiving end the CAD system is supplied with process

This paper was produced as part of Task II of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
dependent data, which needs to be validated to be precise and complete. These data are used inside the CAD system, but the operations must be validated to ensure that in no way design data can be corrupted.

2. The receiving end.

Of course there are standard processes. Dreadfully enough some processes are more standard than others. So it is best not to count on standardisation in this respect.

One way out of standardisation is to interface to a reasonably small number of foundries, that have a large share of the ASIC market. A strong link between fabrication and design technology combines the fast ageing of the former with the large development time of the latter. This is equally not agreeable with a sense of commercial soundness.

What can be expected from a foundry? Processing is laying down geometrical patterns with behavioural implications. Hence the foundry can be expected to tell which geometrical dimensions lead to the designed type of devices but moreover which geometrical dimensions lead to reliable working devices. This information is usually brought out in a mixture of prose and illustration.

2.1 Designrules.

Designrules are rules about minimal mask geometries for designing reliable integrated circuits with a reasonable yield. These rules result from the capabilities of the fabrication technology and electrical properties of the materials. The complexity of a set of designrules is a matter of philosophy, the number of special cases one is willing to explain, and the complexity of the process. It can vary from the two-page lambda rules of Mead and Conway to sets with over 50 pages of documentation. A set of designrules must take into account the characteristics of the photolithography process, etching capabilities, alignment tolerances and electrical constraints.

First the minimum geometry is determined by the resolution of the photolithographic process and by the tolerances of small photoresist patterns. A problem is the non-planarity of the surface. When the number of layers increase, more "hills" and "valleys" arise. When photoresist above the metal layer is exposed in the unmasked area, part of the masked area may be laterally exposed by reflections. So, in areas with many steps, reflection rules have to be applied and the metal spacing value may be smaller or larger than for planar regions.
Secondly, the etching-techniques contribute to the uncertainty in linewidths. Traditional wet etching is isotropic and also material is removed laterally. To improve dimensional control anisotropic (dry) etching-techniques are developed, like plasma etching and reactive ion etching, in which the etchants prefer to etch vertically. A complication for all etching processes is that the etch rate is not uniform across the wafer. Because contact cut dimensions are critical to the layout density and difficult to control, the process is often optimized for minimum size contacts. Thirdly the alignment tolerance is reflected in overlap- and multilayer spacing rules. The alignment sequence is important because the registration of one mask to another depends on the number of alignment steps between them. Finally, some levels are limited in minimum dimensions by electrical constraints. For example, the separation of diffused lines is also governed by the junction punch through condition.

2.2 Description and checking of layout design rules.

Almost all DRC’s use a Design Rule Description File as input. The user translates design rules documentation into statements of the language. It is often not possible to describe all rules because the language is limited in expressing all the rules. The limitations of the language are dependent on the limitations of the DRC. For instance, in general an edge-based DRC can handle more complex rules than an area-based DRC, so the language belonging to the former can express more rules than the one which belongs to the latter. But it is also possible to include more check-operators in the language than the DRC possesses. These extra operators must be translated into combinations of operations the DRC can handle.

These extra operators are meant to provide a user-friendly and easy way to describe the rules. In the case of Schouten ([20]) the language and compiler have to interface with a hierarchical, area-based, DRC ([37]). This checker uses the line-scan technique and checks width and spacing of orthogonal polygons. Rules which involve more than one layer, are translated to spacing- or width checks on layers which are boolean combinations of mask layers.

First of all the design rules are entered in Foundry Interface Specification Format. This specification is compiled and stored for future reference. The two problems (a) foundry verification and (b) design rule entry verification can be solved in a single step. A process template is given in SUPREM-like format and from this template the set of all possible device configurations is generated. These potential devices are grouped according to the impact of process variations. Then it is checked
whether all design rules are valid under the following rules:
- all generated devices are described by the given design rules (if not, then the given set of design rules is incomplete);
- all generated devices are invariant to the given set of design rules, i.e. a device is only allowed to degenerate, not to change behaviour (if not, then the given set of design rules is potentially in fault);
- the geometrical ordering of design rules in the given set is isomorphous with the geometrical ordering of design rules as follows from the device template (if not, then the given set of design rules is inconsistent).

This procedure serves two purposes: it double-checks the manually entered set of design rules and it makes sure that no chip (even no interface validation chip) is designed without a proper insurance.

2.3 State of art layout design rules generation.

Several researchers observe a trend toward more complex and sophisticated structures in advanced VLSI processes ([2], [26]). Think about two metal layers, two poly layers, or incorporating bipolar devices in MOS processes ([4]). All this increased complexity results in more mask levels and therefore more layout rules. Also the scaling of processes towards submicron features introduces more layout rules because of new physical effects and process effects which are not negligible anymore as in less advanced processes. The many layout rules require an enormous number of calculations which is a reason to develop computer aided rules generation tools. Until now a few papers about layout rules generation are published and almost all researchers are from semiconductor industries because they are meant to be used by process engineers.

Bayless et al ([2]) developed a computer program that calculates the layout rules based on electrically measured data on critical dimensions and misalignment tolerances. This data is automatically measured from test structures with the two-terminal or four-terminal method. The relation between e.g. drawn width and electrically measured width on the wafer is:

\[ \text{drawn width} = \text{electrical width} + \text{delta width}, \]

which includes mask, lithographic and etch effects. In general, delta width consists of a bias and a tolerance term: \( \text{delta width} = \text{bias} +/\text{tolerance} \) which means in statistical terms: \( \text{width} = W \)
\[ \text{delta } W = \text{mean } \text{delta } W \pm (\text{factor } \times \text{sigma } \text{delta } W). \]

The layout rule for e.g. width (W) and space (S) of single level features is:

drawn W or S = the larger of:
electrical W or S + delta W or S
or
minimum photoresist W or S

For each level delta W and delta S are measured from wafers with test structures and each wafer has a number of fixed measurement points ranging from center to edge positions. The individual distributions resemble gaussian distributions. The overall distribution must contain the extreme ends, e.g. 3 sigma of all individual ones. The same procedure is applied for misalignment measurements.

The final layout rules can be made "aggressive" or conservative by choosing a certain factor*sigma. For example the layout rule "poly field overlap" in which the overall distributions of the involved levels (active area, poly) are chosen rectangular, is the following:

\[ \text{poly-field-overlap} = [\text{poly bias + poly tolerance}] \times 1/2 - [\text{active bias - active tolerance}] \times 1/2 + \text{misalignment tolerance} \]

A similar approach is pursued by Sugino ([26], Layout Rule Generator) but nothing is specified about the acquisition of dimensional data or misalignment data. The program (LRG) uses almost the same sort of input information. Globally this information is:

1. mask and alignment sequence.
2. dimensional changes of mask, lithography, etch and other processes (mean and standard deviation).
3. alignment tolerances.
4. minimum dimensions (width, space) of mask, lithography, etch and other processes.
5. electrical constraints like source drain spacing, metal step coverage, etc.
6. rule confidence limits

Groves et al ([34]) present a system for automated generation and evaluation of layout rules from a set of specially designed test structures. The system consists of a test hardware configuration and interactive software which provides a variety of displays and statistical analysis of the results. Test structures are designed with reference to the layout rules to be generated or tested, e.g. width-and-space test structures or the "P-field implant overlap of active area" test structure. These structures have varying...
widths and spacings, or overlaps in the latter case. By measuring certain electrical parameters, the minimal values for the corresponding rules are determined. Beck ([33]) developed a computer program which aids the process engineer in calculating design rules and determining the optimal alignment strategy. In fact it is meant to eliminate engineering judgement. According to Beck ([33]) the separation of edges of 2 layers is determined by:

- limits or physical margins, i.e. how close one circuit element may approach another without circuit jeopardy (circuit element - layer feature). Minimal physical margins are determined with test patterns and device physics computer models (SPICE, SAMPLE, SUPREM, etc.) by trial and error, to achieve greatest circuit speeds and packing densities.
- overall processing tolerances. The overall factory processing tolerances must take into account the uncontrollable individual tolerances of registration accuracy, critical dimension control and reading errors plus the controllable factors such as alignment mark polarities, orientation of verniers, or alignment sequence.

The input information for DESRULE is:
- registration errors between each alignment pair.
- critical dimension excursions of each layer.
- list of specific design rules and which layers they involve.
- physical margin, for creation of a rule, or, layout rule value, for analysis of a rule.

A circuit layout rule is calculated by adding the physical margin to the overall factory tolerance. It can be concluded from the above mentioned methods that they are not applicable for a design house because the latter has not the means to fabricate and measure test structures. Furthermore no effort is done to fully automate the generation of rules with respect to completeness. That has still to be done by a human expert.

2.4 The first step towards layout rules.

Design rules can be classified into two categories. First the internal design rules are considered. They deal with rules about overlap and extension. Overlap is defined as follows: when pattern 1 has 3 or 4 edges inside pattern 2, it is said that pattern 2 overlaps pattern 1. The definition of extension is: when pattern 1 has 1 or 2 edges outside pattern 2, it is said that pattern 1 extends beyond pattern 2. The reason for overlap or extension is to exclude the
possible misalignment of different masklevels. The problem encountered is: does pattern i overlap or extend pattern j? The gate pattern, for example, extends beyond the active area window, and the metal pattern overlaps the contact window. Another one is, given that the contactmask is aligned to the polygatemask: does an active area window overlap a contactwindow?

A next category is separation. Two patterns must be separated if it cannot be allowed, that the two patterns become one, or that by the closeness of the two patterns a parasitic device becomes active. Examples are: two metal patterns carrying different signals cannot be allowed to become physically merged. Two diffused regions cannot be so close, that the connected lateral bipolar transistor becomes active. It is clear that geometric relationships between patterns of different masklevels are determined by their meaning or function. A solution to this problem is the idea of masktypes. One can divide processes in groups or classes, e.g. the polygate MOS class, or the polygate bulk CMOS class. For each class exists a group of masktypes which also overlap each other. And in general those masktypes have fixed geometric relationships.

A set of masktypes is composed for the two classes above and are listed next:

- IN : Interconnect, e.g. metal interconnect.
- CC : Contact Cut; all contactcuts between conductors.
- GI : Gate and Interconnect.
- BC : Buried Contact; contact between GI and substrate.
- CI : Channel Implant; depletion, threshold adjust, anti-punch-through.
- SD : Source/Drain diffusion or implant (CMOS).
- WL : Well (CMOS).
- AA : Active Area; isolation of active areas.
- FI : Field Implant
- CA : Capacity; e.g. poly II is topplate of capacitor with poly I.

Because of their fixed relationships it is possible to construct a table which determines the kind of relation. For example, a matrix for the polygate MOS class is shown:

1.75
Table 1. Design rule interrelation matrix for an exemplary polygate MOS process.

<table>
<thead>
<tr>
<th></th>
<th>AA</th>
<th>BC</th>
<th>CI</th>
<th>GI</th>
<th>CC</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td></td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CI</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GI</td>
<td>e</td>
<td></td>
<td></td>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>o</td>
</tr>
</tbody>
</table>

e = extension ; o = overlap

Lambda based designrules.

In order to ease the designer to get his job done, Mead and Conway introduced the concept of lambda based designrules. In the meantime other authors have given adjustments or extensions of these rules, e.g. Lyon (NMOS), Sequin (NMOS, inverted bulk CMOS) or Griswold (CMOS). This concept could be a good start for automatic DG algorithms, so some research is done on the method of deriving the rules. Lambda is thought of as the resolution of a lithographic step. After Lyon, it is also a bound on the width deviation of a feature from its ideal as-drawn size, so the maximal edge shift of a feature is .5 lambda. From this it follows that the minimal feature size is 2 lambda. Finally, lambda is the worst case deviation of its ideal position by misalignment. So worst case, features of different mask levels can deviate as much as 2 lambda from their ideal relationship.

Most rules are derived from a pair of metarules, stated as follows:

- Fatality metarule: Relative movements of feature edges by amounts less than 2 lambda should not be obviously fatal.
- Degradation metarule: Relative movements of feature edges by amounts less than 1 lambda should not cause significant degradation in performance.

Sequin assumes that the worst case misalignment between any 2 levels is lambda and has different metarules. His metarule that governs the overlaps and separations of features of different masks, says:

- Features that should not touch must be separated by 2 lambda if touching is fatal, and by 1 lambda if touching is not fatal.

He derives a number of rules of which some are not obvious.
With these metarules it is relatively easy for a human expert to derive the rules, if you don't consider the exceptions. He can handle concepts like fatality or degradation, but how do you implement these in a computer program? An example:
- metal partly covers a contact cut, and,
- poly partly covers an active area.

The first case is degradation in performance because the contact resistance is increased. The second case is fatal because the drain and source are short circuited. It is not pattern recognition but knowledge about electronics, connected with patterns, that distinguishes both cases. This problem can be circumvented by building a set of rules, expressions in mask type patterns, which describe the fatal pattern combinations. When confronted with the question the rule base is entered and the patterns compared. A disadvantage of this approach is that you have to analyse a process completely, i.e., find all the fatal pattern combinations, to compose the rule base.

Some examples of rules, expressed in language, are:

"enhancement channel in series with depletion channel is enhancement."
"parallel in series is depletion."
"parallel in series bur. contact is channel."
"parallel is buried contact."

A possible procedure of deriving internal design rules is the following:

```
default separation between edges is lambda;
IF relative movement of 2 lambda is fatal
THEN make separation 2 lambda;
```

The second group of design rules are the spacing rules between devices (pattern combinations). First you have the single layer spacings between conductors, and between contact cuts which are 2 lambda, sometimes 3 lambda. The multilayer spacings between conductors are governed by the vertical separation of the layers and the misalignment. For example, there is only a spacing rule for between the AA-type and GI-type conductors because they can form a channel when crossing each other.

Spacing between devices in general can be divided in 3 kinds of spacing rules:
1. singlelayer spacing, e.g. IN - IN type spacing.
2. multilayer spacing between conductors, e.g. AA - GI separation.
3. separation of conductors to the types CC, BC and CI (NMOS).

Note that masktypes like CC, BC and CI cannot exist on its own; CC connects conductors, BC connects GI and AA, CI converts a transistor. Suppose the spacings of the types 1 and 2 are known, then e.g. the spacing between an enhancement - and a depletion transistor reduces to the problem of the separation between enhancement transistor and CI-type pattern.

3.1 Refinement of designrules.

For a new process designrules are derived by a combination of calculation and experience. The calculation of some designrules can be done by using statistical data of the foundry, i.e. the alignment tolerance and the line-width tolerances. The alignment sequence has important consequences in the development of designrules, because the number of alignment steps \( n \) between 2 masklevels determines the registration. Worst case the registration is \( n \) times the alignment tolerance, but in practice the tolerances are statistical and usually a square root relationship is used. So the registration \( R \) is \( R_0 \times (n)^{0.5} \), with \( R_0 \) the alignment tolerance.

From the alignment data of a process an alignmenttree can be constructed. In the next figure an alignmenttree of a typical NMOS process is shown:

![Alignment tree of a typical NMOS process.](image)

From this tree an alignment table with the number of alignment steps can be derived.
Table 2. Alignment table for a typical NMOS process.

<table>
<thead>
<tr>
<th></th>
<th>aa</th>
<th>de</th>
<th>bc</th>
<th>po</th>
<th>co</th>
<th>me</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>de</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>bc</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>po</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>co</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>me</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As an example of designrule derivation the rules associated with metal to poly contacts are calculated, for a 2μm process. The characteristics are as follows: (see Glasser, Dobberpuhl, page 193)

Table 3. Characteristics of the metal to poly contact.

<table>
<thead>
<tr>
<th></th>
<th>level</th>
<th>linewidth tol.</th>
<th>aligned to</th>
</tr>
</thead>
<tbody>
<tr>
<td>poly</td>
<td>-/+ .25 μm/edge</td>
<td>active area</td>
<td></td>
</tr>
<tr>
<td>metal</td>
<td>-/+ .40 μm/edge</td>
<td>contact</td>
<td></td>
</tr>
<tr>
<td>contact</td>
<td>-0 + .6 μm/edge</td>
<td>active area</td>
<td></td>
</tr>
</tbody>
</table>

Because of a possible step coverage problem the contactcut edge must not overlap the poly. Furthermore, because of a specified maximum contact resistance, the minimum contact area between poly and metal is supposed to be 3 μm sq. The contact is aligned to the active area, so n=2 and Rpc=R0 \( \sqrt{2} = 1.06 \) μm.

Fig. 2 example of metal – poly contact calculation.

In the worst case the poly is small and the contact is large, so the worst case bloats, shrinks and translations
have to be added. The minimum overlap is 1.91 μ which is rounded to 2 μ. Metal is aligned to the contact. Assume that the worst case translation is in both x and y direction, then the overlap area must be $\sqrt{3} \times \sqrt{3}$. So the metal must extend $\sqrt{3} - 1 + 0.75 + 0.4 = 1.88$ past the center of the contact, rounded to 2 μ. Then the metal overlap is 1 μ.

4. The role of fabricator information.

Integrated circuit design is in essence the structured and repeated application of design steps such as data entry, design transformation and verification. From the initial behavioural specification one works his way down through structure and geometry, simultaneously refining the design and adding data. While this road is travelled by each designer for every design, the other lane is taken only once per process. The fabrication data are entered at the lowest level of abstraction and fed to higher levels. This process is similar to the design process, but works always bottom-up. For instance the geometrical design rules are supplied

---

Fig. 3 Two-way traffic on the design road.
first. Then the device data are given. Ensuing the logic delays become available and finally an entire cell library is characterised. At each level of abstraction, technology specifications are entered serving as restrictions on the next higher level of abstraction. This two-way traffic is depicted in fig.3. This figure shows that a partial design is entered and transformed. The result is verified and analysed not only considering the initial partial design, but also with respect to technology information.

It is not uncommon, that technology information at the higher levels of design abstraction is supplied on an ad-hoc basis. For instance process parameters for the circuit simulation are supplied by the foundry without any correspondence to design rules. On the contrary they originate from on-wafer measurements using either testchips and/or process control modules. Hence such data can be corrupted.

Even if all technology information is present, there is no guarantee that this information is really used by the system. In an open system architecture such as SPIRIT this might be even more the case. Hence there is a real need for acceptance procedures for every SPIRIT configuration.

5. The giving end.

Giving a design to a foundry is largely governed by the question of getting a design accepted by a foundry. Such acceptance procedures are based on test data collected on each chip. The design must be such that it conforms to a set of rules which guarantees the correspondence between the features on the chip, their description, and the test procedures at the electrical as well as at the logic level. The difficult issue arises because of the large size of the chip and the expected or possible low yield. The custom chip producers are entitled to require:

- Conformity of the geometry with the logic and functional description of the chip. This conformity can be certified by an automatic verification program, accepted by the manufacturer.
- Adherence to a set of rules with respect to dimensions of devices, power distribution and density, modular floorplans, decoupling of subsystems on the chip by applying buffers. This constitutes a document accepted by both the foundry and the CAD house.
- Circuit simulation (of specific quality) of critical parts of the chip, based on device models supplied by the foundry and counterchecked by similar software at the foundry.
Fig. 4 Layout of a device parameter chip.
Bibliography.

17. J. Middelhoek, "IC – Technologie", lecturebook for students, Technological University of Twente.
24. L. Spaanenburg, "Digital MOS Integrated Circuits", Notes from the graduate course, Technological University of Twente.
Foundry Interface file syntax.

1. The use of the syntax.

The syntax has been written in accordance to the standard notation proposed by Niklaus Wirth. Production rules use equals (character '"') to relate identifiers to expressions, vertical bars (character '|'') to indicate a choice of expressions, and double quotes (character '"'') to indicate terminal characters. Curly brackets (characters '{' and '}') indicate repetition any number of times including zero. Square brackets (characters ' [' and ' ]') indicate optional factors (i.e. zero or one repetition). Parentheses (characters '(' and ')') are used for grouping. Rules are terminated by a period (character '.').

The syntax reflects that lower and uppercase letters may be used, all blocks and statements must be closed with a semicolon (character ';'). Just before or right after a semicolon comments may be written. There are several types of comments:
- /* type I */
- (* type II *)
- # type III #
- co type IV co
- CO type V CO

Comments of the same type may not be nested, comments of different types may be nested. An example:
- /* this is /* wrong */ */
- /* this is # correct # */

Also wrong are not correct balanced comment pairs:
- (* this is # also *) wrong #

Every statement may be separated by blanks, tabs or newlines. (seps in the syntax)


For the time being, the FI-file consists of three parts.
- The Header_information block,
- The Process_information block,
- The Design_rules block.

Extending the FI-file with e.g. the electrical interface, a new block has to be added to the syntax.

The syntax:

\[
\text{fi-file } = \{\text{comment}\_\text{statement}\} \\
\text{header}\_\text{information}\;\text{semi}
\]
3. The first part: Header_information block.

The first part of the FI-file is the header_information block. The header information block should start with the keyword HEADER_INFORMATION_BLOCK, which may be abbreviated to H_I_B, followed by a semicolon. (All statements and keywords must be followed by a semicolon!) The following are zero or more header_information statements. The header_information block is terminated with the keyword END, followed by a semicolon.

The syntax:

```
header_information = header_info_block_symbol semi
                      {{header_info_statement semi}}
                      end_symbol.

header_info_statement = {{ext_string {sep}}}
                        colon
                        {{(sep) ext_string {sep}}}).

ext_string = {{string | \" | \" | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | " | “}.
```
Appendix 1-3

/* any printable ascii but space, 
d_quote tab, newline, colon or 
semicolon */

An example of a header information block:

```
HEADER_INFORMATION_BLOCK ;  (* this may also be: h_i_b *)

    foundry name   :   AMI;
    internal code  :   5002.C;
    creation date  :   4/24/85;
    originator     :   W.R. Iter;
    modification date :   4/25/85;
    modified by    :   M. Odifier;

END;
```

The FI-compiler will check the header information block only for correct syntax, not for the contents of the header info statements.


The second part of the FI-file is the process information block. The process information block should start with the keyword PROCESS_INFORMATION_BLOCK, which may be abbreviated to P_I_B, followed by a semicolon. The following statement must be a process name line. This statement contains the name of the process (e.g. nmos) and is followed by a semicolon.

The syntax:

```
process_information    = process_info_block_symbol semi
                        process_name_line semi
                        [((nr_of_layers_line semi)
                          ((process_info_statement semi))]
                        end_symbol.

process_name_line      = process_name_symbol (sep)
                        colon (sep) ext_string.

nr_of_layers_lines     = nr_of_layers_symbol (sep)
                        colon (sep) integer.

process_info_statement = layer_name (sep) layer_nr (sep)
                        layer_code (sep) layer_color
```
Appendix 1-4

{sep} type {sep} terminal.
layer_name = ext_string sep.
layer_nr = integer sep.
layer_code = char [char] [char] [char] sep
layer_color = color_symbol.
type = solid_symbol | stipple_symbol.
terminal = yes_symbol | no_symbol.

An example of a process information block:

PROCESS_INFORMATION_BLOCK;
process_name : cmos;
diffusion 1 cd green solid yes;
depletion 4 de magenta stipple no;
minuswell 3 pm yellow stipple no;
polysilicon 5 cp red solid yes;
capacitor 7 cc white stipple no;
boron 6 cb orange stipple no;
metal_contacts 8 cm black solid no;
metal 9 me blue solid yes;
pad 10 pa aqua solid yes;
boron_reversal 11 br black stipple no;
END;

5. The third part: Design_rules block.

The third part of the FI-file is the design rules block. The design rule should start with the keyword DESIGN_RULE_BLOCK, which may be abbreviated to D_R_B, followed by a semicolon and terminates with the keyword END also followed by a semicolon. The design_rules can consist of:
- (zero or more) alias_definitions and
- (zero or more) design_rule statements and
- (zero or more) device_blocks.
A design_rule statement consists of a one_layer_rule, a multi_layer_rule or an if_statement with a design_rule statement. These rules are followed by a violation_text and a violation_number.

Alias_definitions are used to create a new (auxiliary) layer
which can be composed by means of one_layer_rules, multi_layer_rules of logical_operations. (The rules in the alias_definition are, not followed by a violation_text and violation_number).

Design_rules which are only valid at certain devices are described in the device_block. A device_block should start with the keyword DEVICE_BLOCK followed by a semicolon. The next statement must be a device_block_name line. The name of the device_block is specified here. Next statements in a device_block are
- (zero or more) alias_definitions and
- (zero or more) design_rules.

A device_block must terminate with the keyword END followed by a semicolon. The device blocks are discussed more detailed in the following sections.

The syntax of the design_rules block:

```
design_rules    = design_rules_block_symbol semi
                 | ((alias_definition |
                 | design_rule_statement |
                 | device_block) semi)
                 | end_symbol.
alias_definition = define_symbol
definition_part  = one_layer_rule
                 | multi_layer_rule
                 | logical_layer_operator
                 | (logical_layer_operator).
design_rule_statement = if_statement
                       | one_layer_rule
                       | multi_layer_rule.
if_statement     = if_symbol (sep)
                 | lrb_symbol (sep) condition (sep)
                 | rrb_symbol then_symbol (sep)
                 | design_rule_statement fi_symbol
                 | semi
                 | if_symbol (sep)
                 | lrb_symbol (sep) condition (sep)
                 | rrb_symbol then_symbol (sep)
                 | design_rule_statement else_symbol
                 | (sep) design_rule_statement
                 | fi_symbol semi.
```
condition = same_voltage_symbol (sep) layer (sep) 
  | different_voltage_symbol (sep) layer 
  | to_be_shrunk_symbol 
  | not_symbol to_be_shrunk_symbol 
  | angle_symbol (sep) 
  | relational_arguments angle_integer (sep) boolean_operator 
  | relational_arguments (sep) angle_integer (sep) layer (sep) layer.

device_block = device_block_symbol semi 
  device_name_line semi 
  { (alias_definition | 
    design_rule_statement) } 
  end_symbol.

device_name_line = device_name_symbol (sep) colon (sep) ex

There are two types of design_rule_statements checks:
- one_layer_rules: rules that check the distance between objects within one layer.
- multi_layer_rules: rules that check the distance between objects of one layer with objects of another layer.

Both types of design_rules have an almost common form:
checkoperator relational-arguments layer [layer] 
violation_text violation_number.

The checkoperator determines the kind of check that must be verified. (e.g. width, length, overlap, inset) The relational-arguments specify the checking criteria. (e.g. 'width > 200' or 'spacing >= 200 && < 350') The objects that have to be checked are situated within the specified layer(s). Violation_text and violation_number are used for reporting purposes. There is also an conditional selection statement:
- if_statement.
   The if_statement is used for certain conditions:
   - static conditions: e.g. circuits to be shrunk or not, (static means here: anteriori known),
   - dynamic conditions: e.g. crossing angle restriction, same/different voltages. (dynamic means here: determined by the layout and/or electrical specifications)

The implementation of these condition carry some difficulties along: For example if the circuit can be shrunk it must be known along with the layout. For two objects with the same voltage, it has to be specified how these must be
connected will they be at the same voltage. (a different voltage will run across the same problems.)

5.1 The layers.

A layer can be a layout layer or a result of a layer_operation, (by means of the layer_name of an alias_definition or a logical layer operation). These result layers are called auxillary layers.

The syntax:

```
layer = lrb_symbol {sep} edge_symbol {sep}
       layer {sep} rrb_symbol
       | lrb_symbol {sep} inside_symbol {sep}
       layer {sep} rrb_symbol
       | layer_code
       | layer_name
       | logical_layer_operation.
```

The layout_layers consists of the original layout-data i.e. the diffusion_layer, the metal_layer, etc. These layers are determined by the layer_codes of the layers in the design_rule_statement, e.g. nd, nm, etc. Which are defined in the process_information block of the FI-file. The result of a layer_operator will be an auxilary layer described at the alias_definition or the layer_operation itself. Alias definitions define an auxilary layer which has a new layer_name. Logical layer_operations create also an auxilary layer which has no name. (The operations forms the result layer.)

5.2 The violation_text.

A violation description should be given in the design_rule_statement for the reporting purposes, e.g. the design rule verification program of the ICD system will use the violation description in its violation report.

The violation_text is given as strings between the characters `""` (vio_text_begin_symbol) and `""` (vio_text_end_symbol).

```
vio_text = vio_text_begin_symbol {sep}
           ((ext_string {sep}))
           vio_text_end_symbol.
```

1.91
5.3 The violation_number.

For the reporting purpose, a violation number should also be given in the design_rule_s in form of a string between brackets, the characters '( ' and ')'. The violation numbers are the design rule numbers of the design rules in the manual.

violation_number = lrb_symbol sep {sep} string period string (period string) sep rrb_symbol.

5.4 Measurements.

All numbers specified in the checks are in units. Units are hundredths of a micron (micrometer).

5.5 Check operators.

The check operators can be grouped:
- if statement
- one layer checks
- multi layer checks

and which will be discussed below.

5.5.1 If statement. The if statement can be used for certain conditions like to be shrunk, same voltage, different voltages or crossing angles. (For the time being.)

if_statement = if symbol {sep}
               lrb_symbol {sep} condition {sep} rrb_symbol then_symbol {sep} design_rule_statement fi_symbol semi
               | if_symbol {sep} lrb_symbol {sep} condition {sep} rrb_symbol then_symbol {sep} design_rule_statement else_symbol {sep} design_rule_statement fi_symbol semi.

condition = same_voltage_symbol {sep} layer
            | different_voltage_symbol {sep} layer | to_be_shrunk_symbol
            | not_symbol to_be_shrunk_symbol
            | angle_symbol {sep} relational_arguments angle_integer

1.92
5.5.2 One layer rules. A one_layer_rule can consist of:
- width_rule,
- spacing_rule,
- notexists_rule,
- size_rule.

5.5.2.1 Width. The width_rule describes a rule where specified value(s) of the rectangle's width on one layer is determined. The values specified in width_rule are a relational operator followed by an integer number. The next boolean operator, relational operator and the integer number are not used in the alias_definition. (The width is the contrary of the length. For the definition of length see below)

The syntax:

width_rule = width_symbol (sep)
relational_arguments layer (sep)
[violation_text (sep)
violation_number]

relational_arguments = relational_operator integer
[boolean_operator
relational_operator]

relational_operator = less_symbol
| less_equal_symbol
| equal_symbol
| notequal_symbol
| greaterequal_symbol
| greater_symbol

boolean_operator = relational_and_symbol
| relational_or_symbol.

For example for the design rule:

1.1 Minimum diffusion width 6u 6u diffusion

the corresponding design_rule_statement should be:
width >= 600 nd
(" diffusion width violation ") (1.1);

where nd is the layer_code for the diffusion layer.

5.5.2.2 Spacing. The spacing_rule describes a rule for the separation checking of rectangle's separation on one layer.

The syntax:

```
spacing_rule = spacing_symbol {sep} 
relational_argument layer {sep} 
[violation_text {sep} 
violation_number].
```

For example for the design rule:

```

| diffusion |
---|---|---|
1.2 Minimum diffusion separation 6u 6u
  v

---|---|---|
```

the corresponding design_rule_statement should be:

```
spacing >= 600 nd
(" diffusion spacing violation ") (1.2);
```

where nd is the layer_code of the diffusion layer.

5.5.2.3 Notexist. The notexist_rule describes a rule for the checking of not occurring of objects on one layer.

The syntax:

```
notexists_rule = notexists_symbol {sep} 
layer_operation 
[violation_text {sep} 
violation_number].
```

For example the design rule:

```
1.3 capacitor cannot be contacted upon poly

not_exists (and cp (and cc cm))
```

1.94
where cp, cc and cm are the layer_code for respectively 
poly, capacitor and metal layers.

5.5.2.4 Size. The size_rule describes a rule for the size 
(i.e. width and length) checking of rectangle's on a layer.

The syntax:

```
size_rule = size_symbol {sep}
  relational_arguments [by_symbol]
  {sep} relational_arguments
  layer {sep} [violation_text
  {sep} violation_number].
```

For example for the design rule:

```
1.4 Minimum contact size 3x4u
```

the corresponding design_rule_statement should be:

```
size >= 300 by >= 400 cm
("minimum metalcontact size violation ") (1.4);
```

where cm is the layer_code of the metalcontact layer. Also 
maximum values can be given (as in all check rules:)

```
1.5 Maximum contact size 5x9u
```

the corresponding design_rule_statement should be:

```
size < 500 by < 900 cm
("maximum metalcontact size violation ") (1.5);
```

The two design rules could be written also in one:

```
size >= 300 && < 500 by >= 400 && < 900 cm
("minimum and/or maximum metalcontact size violation ") (1.4,5)
```

5.5.3 Multi_layer_rules. A Multi_layer_rule can consist 
of:
- multi_spacing_rule,
- overlap rule,
- oversize_rule,
- inset_rule.

5.5.3.1 Multi spacing rule. The multi_spacing_rule 
describes a rule for the separation checking of edge's 
separation on two different layers. The outside edges are
being checked:

```
// | //
// | //
first // | // second
layer // | // layer
// | multispacing //
// | //
// | //
// | //
```

The syntax:

```
multispacing_rule = multispacing_symbol {sep} relational_arg layer {sep} layer {sep} [violation_text {sep} violation_number]
```

For example for the design rule

```
1.6 Minimum separation between diffusion and poly 3u 3u
```

```
+--+--+--+--+---+
|  |  |  |  |  |
+v 3u +---+-+
```

The corresponding design_rule_statement should be:

```
multispacing >= 300 nd np
("diffusion to poly violation") (1.6);
```

where nd is the layer_code of the diffusion layer and np is the layer_code of the poly layer.

5.5.3.2 Overlap rule. The overlap_rule describes a rule for the separation checking of edge's overlap on two different layers. The outside edge of the first layer is being checked against the inside edge of the second layer:
The syntax:

```
first_layer overlap second_layer
```

An example of a overlap rule:

1.7 Minimum overlap of metal on poly
(when running in parallel)

```
| m | m |
|--------------------------|
| mmnnnnnnnnnnnnnnnnnnnnnnn | mmnnnnnnnnnnnnnnnnnnnnnnn |
```

The corresponding design_rule_statement should be:

```
overlap >= 200 np nm (perpendicular direction np)  
("metal poly overlap violation") (1.7);
```

where np is the layer_code for poly and nm the layer_code for metal.

5.5.3.3 Inset rule. The inset_rule describes a rule for the separation checking of edge's separation on two different layers. The inside edge of the first layer is being checked against the inside edge of the second layer:
The syntax:

\[
\text{inset\_rule} = \text{inset\_symbol} \ (\text{sep}) \ \text{relational\_arguments} \text{ layer} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ [\text{violation\_text} \ (\text{sep}) \ \text{violation\_number}]
\]

1.8 Minimum poly overlap over diffusion 8u

The corresponding inset\_rule\_statement should be:

\[
\text{inset} \geq 800 \ \text{np\ nd} \ \text{("minimum poly-diff overlap violation") (1.8)};
\]

where np is the layer\_code for poly and nd the layer\_code for diffusion.

5.5.3.4 Oversize rule. The oversize\_rule describes a rule for the separation checking of edge's overlap on two different layers in all directions. The outside edge of the first layer is being checked against the inside edge of the second layer:
The syntax:

```
oversize_rule
  = oversize_symbol (sep)
  relational_arguments layer {sep}
  layer { sep} [violation_text {sep}
  violation_number].
```

An example of a oversize rule:

1.9 Minimum diffusion overlap around contact hole 2u

```
  oversize >= 200 nd nc
  ("contact-diffusion enclosure violation") (1.9);
```

where nd is the layer_code for diffusion and nc the layer_code for contact-holes.
5.6 **Alias_definitions.**

**Alias_definitions** provide a method for defining auxiliary layers which can be specified by means of logical operations on layers (and, or, invert) and checks on layers (single-, multilayer checks). **Alias_definitions** must have a unique name and may not be redefined. (Only in device_blocks the alias_definitions may be redefined.)

For example, a transistor gate may be defined as an and operation between poly and diffusion:

\[
\text{define (and nd np) as transistor;}
\]

5.6.1 **Checks on layers.** Checks on layers are the same as the design_rules (one- and multi layer operations). Violation text and violation number may not be present.

5.6.2 **Logical operations.** There are three logical operators:
- **And_operator,**
- **Or_operator,**
- **Invert_operator.**

5.6.2.1 **And operator.** The **and_operator** creates an auxiliary layer with the symbols occurring on both layers. (The logical AND.)

The syntax:

\[
\text{and_operator} = \text{and_symbol (sep) layer (sep) layer.}
\]

The **and_operator** may be embraces by round braces. (This is not obligated.) This operator can be nested because in layer another **and_operator** may be used.

An example:

\[
\text{and nd np}
\]

or

\[
(\text{and nd np})
\]

where **nd** and **np** are the layer_codes for diffusion respectively poly.

5.6.2.2 **Or operator.** The **or_operator** creates an auxiliary layer with the symbols occurring on one of both (or on both) layers. (The logical OR.)
The syntax:

or_operator = or_symbol {sep} layer {sep} layer.

The or_operator may be embraces by round braces. (This is not obligated.) This operator can also be nested because in layer another or_operator may be used.

An example:

or nm nd or (or nm nd)

where nm and nd are the layer_codes for metal respectively diffusion.

5.6.2.3 **Invert operator.** The invert_operator creates an auxiliary layer with the inverted symbols occurring on the layers. (The logical *NOT*.)

The syntax:

invert_operator = invert_symbol {sep} layer.

The invert_operator may be embraces by round braces. (This is not obligated.) This operator can also be nested because in layer other logical_operators may be used.

An example

invert nm or (invert nm)

5.7 **Device blocks.**

Some design_rules are sometimes only valid at some device. By means of the device_blocks this can be achieved. In the device_block there can be specified:
- (zero or more) alias_definitions and
- (zero or more) design_rules.

The alias_definitions defined in this device_block are only valid inside this device_block. Alias_definitions from which the name is already defined outside the device_block can be redefined. The scope of this alias_definition is limited to this device_block.
Foundry Interface file syntax.

\[
\text{fi-file} \quad = \quad \{\text{comment_statement}\} \\
\text{header_information} \text{ semi} \ 
\text{process_information} \text{ semi} \\
\text{design_rules} \text{ semi}.
\]

\[
\text{comment_statement} \quad = \quad "\text{/\text{*}}" \text{ comment_text1 } "\text{*/}" \\
\quad | \ "\text{\text{*}*}" \text{ comment_text2 } "\text{*/}" \\
\quad | \ "\text{#}" \text{ comment_text3 } "\#" \\
\quad | \ "\text{CO}" \text{ comment_text4 } "\text{CO}" \\
\quad | \ "\text{CO}" \text{ comment_text5 } "\text{CO}".
\]

\[
\text{header_information} \quad = \quad \text{header_info_block_symbol semi} \\
\quad \{(\text{header_info_statement semi})\} \\
\quad \text{end_symbol}.
\]

\[
\text{header_info_statement} \quad = \quad \{(\text{ext_string } \{\text{sep}\})\} \\
\quad \text{colon} \\
\quad \{(\text{sep} \text{ ext_string } \{\text{sep}\})\}.
\]

\[
\text{process_information} \quad = \quad \text{process_info_block_symbol semi} \\
\text{process_name_line semi} \\
\quad \{(\text{process_info_statement semi})\} \\
\quad \text{end_symbol}.
\]

\[
\text{process_name_line} \quad = \quad \text{process_name_symbol } \{\text{sep}\} \text{ colon } \{\text{sep}\} \\
\quad \text{ext_string}.
\]

\[
\text{process_info_statement} \quad = \quad \text{layer_name } \{\text{sep}\} \text{ layer_nr } \{\text{sep}\} \\
\text{layer_code } \{\text{sep}\} \text{ layer_color } \{\text{sep}\} \\
\text{type } \{\text{sep}\} \text{ terminal}.
\]

\[
\text{layer_name} \quad = \quad \text{ext_string } \text{ sep}.
\]

\[
\text{alias_name} \quad = \quad \text{ext_string } \text{ sep}.
\]

\[
\text{layer_nr} \quad = \quad \text{integer } \text{ sep}.
\]

\[
\text{layer_code} \quad = \quad \text{char } [\text{char}] [\text{char}] [\text{char}] \text{ sep}.
\]

\[
\text{layer_color} \quad = \quad \text{color_symbol}.
\]

\[
\text{type} \quad = \quad \text{solid_symbol} \mid \text{stipple_symbol}.
\]

\[
\text{terminal} \quad = \quad \text{yes_symbol} \mid \text{no_symbol}.
\]

\[
\text{design_rules} \quad = \quad \text{design_rules_block_symbol semi} \\
\quad \{(\text{alias_definition} \mid \text{design_rule_statement} \mid \text{device_block})\} \text{ semi}
\]

1.102
end_symbol.

alias_definition = define_symbol (sep)
   definition_part (sep)
   as_symbol (sep)
   alias_name.

definition_part = one_layer_rule
   | multi_layer_rule
   | logical_layer_operator.

design_rule_statement = if_statement
   | one_layer_rule
   | multi_layer_rule.

if_statement = if_symbol (sep)
   lrb_symbol (sep) condition (sep)
   rrb_symbol (sep) then_symbol (sep)
   design_rule_statement semi
   fi_symbol semi
   | if_symbol (sep)
   lrb_symbol (sep) condition (sep)
   rrb_symbol (sep) then_symbol (sep)
   design_rule_statement semi
   else_symbol (sep)
   design_rule_statement semi
   fi_symbol semi.

condition = same_voltage_symbol (sep) layer (sep) layer
   | different_voltage_symbol (sep) layer (sep) layer
   | to_be_shrunk_symbol
   | not_symbol to_be_shrunk_symbol
   | angle_symbol (sep)
   relational_arguments (sep)
   angle_integer (sep) boolean_operator (sep) relational_arguments (sep)
   angle_integer (sep) layer (sep) layer.

one_layer_rule = width_rule
   | spacing_rule
   | not_exists_rule
   | size_rule.

multi_layer_rule = multi_spacing_rule
   | overlap_rule
   | oversize_rule
   | inset_rule.

width_rule = width_symbol (sep) relational_arguments
Appendix 2-3

\[
\begin{align*}
\text{spacing_rule} & = \text{spacing_symbol} \ (\text{sep}) \ \text{relational_arguments} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ [\text{violation_text} \ (\text{sep}) \ \text{violation_number}]. \\
\text{not_existst_rule} & = \text{not_exists_symbol} \ (\text{sep}) \ \text{layer_operation} \ (\text{sep}) \ [\text{violation_text} \ (\text{sep}) \ \text{violation_number}]. \\
\text{size_rule} & = \text{size_symbol} \ (\text{sep}) \ \text{relational_arguments} \ (\text{sep}) \ [\text{by_symbol}] \ (\text{sep}) \ \text{relational_arguments} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ [\text{violation_text} \ (\text{sep}) \ \text{violation_number}]. \\
\text{multi_spacing_rule} & = \text{multi_spacing_symbol} \ (\text{sep}) \ \text{relational_arguments} \ \text{layer} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ [\text{violation_text} \ (\text{sep}) \ \text{violation_number}]. \\
\text{overlap_rule} & = \text{overlap_symbol} \ (\text{sep}) \ \text{relational_arguments} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ [\text{directionSpecifier} \ (\text{sep})] \ [\text{violation_text} \ (\text{sep}) \ \text{violation_number}]. \\
\text{directionSpecifier} & = \text{lrb_symbol} \ (\text{sep}) \ [\text{perpendicular_symbol} \ (\text{sep}) \ \text{direction_symbol} \ (\text{sep}) \ \text{rrb_symbol}]. \\
\text{oversize_rule} & = \text{oversize_symbol} \ (\text{sep}) \ \text{relational_arguments} \ \text{layer} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ [\text{violation_text} \ (\text{sep}) \ \text{violation_number}]. \\
\text{inset_rule} & = \text{inset_symbol} \ (\text{sep}) \ \text{relational_arguments} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ \text{layer} \ (\text{sep}) \ [\text{violation_text} \ (\text{sep}) \ \text{violation_number}]. \\
\text{device_block} & = \text{device_block_symbol} \ \text{semi} \ ((\text{alias_definition} \ | \ \text{design_rule_statement} \ \text{semi}) \ \text{semi}) \ \text{end_symbol}.
\end{align*}
\]

\[
\begin{align*}
\text{device_name_line} & = \text{device_name_symbol} \ (\text{sep}) \ \text{colon} \ (\text{sep}) \ \text{ext_string}.
\end{align*}
\]

\[
\begin{align*}
\text{relational_arguments} & = \text{relational_operator} \ \text{integer} \\
& | \ \text{boolean_operator} \ \text{relational_operator} \ \text{integer}.
\end{align*}
\]

\[
\begin{align*}
\text{relational_operator} & = \text{less_symbol} \\
& | \ \text{less_equal_symbol} \\
& | \ \text{equal_symbol} \\
& | \ \text{equal_equal_symbol}.
\end{align*}
\]

1.104
boolean_operator = relational_and_symbol
| relational_or_symbol.

layer = lrb_symbol (sep) edge_symbol (sep) layer (sep) rrb_symbol
| lrb_symbol (sep) inside_symbol (sep) layer (sep) rrb_symbol
| layer_code
| alias_name
| logical_layer_operation.

logical_layer_operator = and_operator
| lrb_symbol and_operator rrb_symbol
| or_operator
| lrb_symbol or_operator rrb_symbol
| invert_operator
| lrb_symbol invert_operator rrb_symbol.

and_operator = and_symbol (sep) layer (sep) layer.

or_operator = or_symbol (sep) layer (sep) layer.

invert_operator = invert_symbol (sep) layer.

violation_text = vio_text_)begin_symbol (sep) {}
| text_string (sep))
| vio_text_end_symbol.

violation_number = lrb_symbol sep (sep) string period string (period string) (sep) sep rrb_symbol.

header_info_block_symbol = "HEADER_INFORMATION_BLOCK"
| "header_information_block"
| "H_I_B"
| "h_i_b".

end_symbol = "END" | "end".

process_info_block_symbol = "PROCESS_INFORMATION_BLOCK"
| "process_information_block"
| "P_I_B"
| "p_i_b".

design_rule_block_symbol = "DESIGN_RULES_BLOCK"
| "design_rules_block"
| "D_R_B".
device_block_symbol = "DEVICE_BLOCK" | "device_block".
define_symbol = "DEVINE" | "devine".
as_symbol = "AS" | "as".
if_symbol = "IF" | "if".
then_symbol = "THEN" | "then".
else_symbol = "ELSE" | "else".
fi_symbol = "FI" | "fi".
same_voltage_symbol = "SAME_VOLTAGE" | "same_voltage".
different_voltage_symbol = "DIFFERENT_VOLTAGE" | "different_voltage".
to_be_shrunk_symbol = "TO_BE_SHRUNK" | "to_be_shrunk".
not_symbol = "NOT" | "not".
angle_symbol = "ANGLE" | "angle".
device_name_symbol = "DEVICE_NAME" | "device_name".
process_name_symbol = "PROCESS_NAME" | "process_name".
color_symbol = "AQUA" | "BLACK" | "BLUE" | "CYAN" | "GREEN" | "MAGENTA" | "ORANGE" | "RED" | "YELLOW" | "WHITE" | "aqua" | "black" | "blue" | "cyan" | "green" | "magenta" | "orange" | "red" | "yellow" | "white".
solid_symbol = "SOLID" | "solid".
stipple_symbol = "STIPPLE" | "stipple".
yes_symbol = "YES" | "yes".
no_symbol = "NO" | "no".
width_symbol = "WIDTH" | "width".
spacing_symbol = "SPACING" | "spacing".

 Appendix 2-5

1.106
multi_spacing_symbol = "MULTI_SPACING" | "multi_spacing".
not_exists_symbol = "NOT_EXISTS" | "not_exists".
size_symbol = "SIZE" | "size".
overlap_symbol = "OVERLAP" | "overlap".
direction_symbol = "DIRECTION" | "direction".
perpendicular_symbol = "PERPENDICULAR" | "perpendicular".
and_symbol = "AND" | "and".
or_symbol = "OR" | "or".
invert_symbol = "INVERT" | "invert".
oversize_symbol = "OVERSIZE" | "oversize".
stretch_symbol = "STRETCH" | "stretch".
vio_text_begin_symbol = "(" d_quote ")."
vio_text_end_symbol = d_quote "
less_symbol = "<".
lessequal_symbol = "<=".
equal_symbol = "=".
notequal_symbol = "!=".
greaterequal_symbol = ">=".
greater_symbol = ">".
relational_and_symbol = "&&".
relational_or_symbol = "!!".
lrb_symbol = "(" /* left round bracket */
rrb_symbol = ")" /* right round bracket */
string = {{char | digit | "}
ext_string = {{string | "!" | "@" | "}

Appendix 2-7

/* any printable ascii but space, d_quote, tab, newline, colon or semicolon */

comment_text1 = (any printable ascii) without sequences of "/*" or
d_quote, tab, newline, colon or semicolon */

comment_text2 = (any printable ascii) without sequences of "/(*)" or

comment_text3 = (any printable ascii but "/")

comment_text4 = (any printable ascii) without sequences of "co".

comment_text5 = (any printable ascii) without sequences of "/CO".

angle_integer = integer value between 0 and 180.

sinteger = "-" integer | "+" integer.
            /* not used */

integer = digit {digit}.

semi = (sep) {comment_statement} (sep)
       semicolon (sep) {comment_statement} (sep).

d_quote =

period = ".".

colon = ":".

semicolon = ";".

sep = blank | newline.

blanc = tab | space.

space = " ".

tab =

newline = 0.

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
<table>
<thead>
<tr>
<th>char</th>
<th>= lowchar</th>
<th>uppchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowchar</td>
<td>= &quot;a&quot;</td>
<td>&quot;b&quot;</td>
</tr>
<tr>
<td>uppchar</td>
<td>= &quot;A&quot;</td>
<td>&quot;B&quot;</td>
</tr>
</tbody>
</table>
Chapter 2
Low Level Verification
An Hierarchical and Technology Independent Design Rule Checker

T. G. R. van Leuken
J. Liedorp
Delft University of Technology

ABSTRACT

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

This paper was produced as part of Task III of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
1. Introduction

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

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

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

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

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

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

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

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

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

Section 6 at last will give some conclusions and indicates a
direction for further research.
2. Augmented Instancing of a cell

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

![Figure 2.1: Cell interconnection.](image)

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

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

2. The implied circuit of the cells referenced may not be changed as a result of interfringing primitives.

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

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

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

- All rectangles of the top level cell. Those rectangles have a checktype zero.

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

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

The conversion of the orthogonal rectangles to line segments is done based on the stateruler scan algorithm given in fig. 3.1. We discuss the algorithm first, followed by an explanation. It is applicable in many situations where the artwork data is used as input. E.g. it can equally be used for (layout to circuit) extraction, design rule checking and the generation of derived geometries (polygons, groups, pattern generation). The method is best illustrated with the simple example of rectangle to line segment conversion. The complexity of this essential algorithm is linear in the number of edges in the layout and will strongly influence the overall complexity.

begin

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

figure 3.1: Stateruler Scan Algorithm.

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

The algorithm proceeds by making and analyzing what we call stateruler profiles. A stateruler profile is made by repeatedly selecting an event and inserting that event in the stateruler. The selection of events is based upon a selection criterion. In the case of rectangle to line segment conversion the events are the rectangles. The selection criterion is the smallest left value of the rectangle and if two rectangles have the same left value, the smallest bottom value. The procedure 'select_event' returns the event_status. The event_status will be 'NIL' if there are no more events to select.

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

![Figure 3.2](image)

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

- **START**: A start segment will have the interior of the polygon located to the right of it.
- STOP: A stop segment will have the interior of the polygon located to the left of it.

- CHANGE: A change segment indicates a change of checktype. The polygon area to the right will have the checktype of the CHANGE segment.

- START_OV: A start_ov segment indicates the start of an overlap of two areas. The overlap is situated to the right of the segment.

- STOP_OV: A stop_ov segment indicates the cease of an overlap. The overlap will be situated to the left of this segment.

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

Whenever two or more rectangles with different checktypes start to overlap we will generate a CHANGE segment. The checktype of such a segment will be made zero. At the end of the overlap another CHANGE segment is generated, restoring the checktype. In doing so we can assure that these overlap areas are checked because all areas with checktype zero will be fully checked. (see section 5).

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

- x_value: The x_value of the segment.
- occurrence: the occurrence type as described above.
- y_bottom: The y_bottom value of the segment.
- y_top: The y_top value of the segment.
- connection_type: This variable indicates if the segment has connections to the upper or the lower side or both.
- group_nbr: This variable is an integer indicating what polygon the segment belongs to.
- checktype: This variable has the meaning as stated above.
The procedure 'analyze_stateruler' returns the next position for which the stateruler should be analyzed again. It returns MAX_INTEGER if that is not necessary.

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

The program performing this conversion is the program makevin.
4. Design Rule Checking

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

- A part that for a certain rule selects the items involved.

- A part that does the actual checking upon these items.

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

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

- A part that from this data gathered selects the data needed for a check of a certain design rule.

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

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

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

2. A part that does overlap checks and gap checks concerning two (combination) masks.
So the design rule check is done in three steps:

1. First all the combination masks needed in the check are made by the program nbool. The files that have to be made must be given on a file nbool, the format of which is given in appendix A.

2. Then the checks concerning one (combination) mask are carried out by the program dimcheck. The checks that have to be carried out must be given on a file proc1.d, the format of which is given in appendix A.

3. At last the checks concerning two (combination) masks are carried out by the program dubcheck. The checks that have to be carried out must be given on a file proc1.d, the format of which is given in appendix A.

In the files mentioned one also must specify the design rules the design of the integrated circuit has to obey.

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

4.1 The program nbool

nbool is a program to generate logical combination masks, which are a combination of input masks in line segment format (vln format).

As input it uses a file with the description of the logical formulas of the masks to be made. For a description of this file see appendix A. Also the vin files mentioned in these formulas and a file with the name of the cell(s) of which the files have to be made is needed. It uses the stateruler algorithm in a similar way as described in section 3.

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

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

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

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

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

- Overlap of interconnection layers without the presence
  of a terminal in that layer.
- Overlap of layers which do not interconnect.

- Overlap of different layers belonging to different cells indicating that one has possibly created an unwanted element.

For a detailed description of nbool as it appears in the ICD release 1, see appendix B.

4.2 The program dimcheck.

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

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

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

- The x_position of the edge in the field previous to the one where the stateruier is analyzed.

- The status of the layer (PRESENT, NOT_PRESENT, CHG_TO_PRESENT or CHG_TO_NOTPRESENT).

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

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

For a detailed description of dimcheck as it appears in the ICD release 1, see appendix C.

4.3 The program dubcheck

The program dubcheck performs gapchecks between two different masks and determines if a mask is overlapped by another mask with a certain value. As its input it uses a file containing the files that have to be checked with respect to each other and the distance or overlap they have to obey. Also an integer is given for a gapcheck to indicate if groupnumbers have to be taken into account for. For a description of this file see appendix A. Furthermore the .vln files stated in the file above must be present and a file containing the cells to be checked.

It also uses the stateruler algorithm in a similar way as described in section 3. The global way the program works is:

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

The events of the algorithm here are the segments read from
the two vln files. The selection criteria again are the x_value and the y_bottom value of the segment.

In the stateruler the status (PRESENT, NOT_PRESENT, CHG_TO_PRESENT or CHG_TO_NOTPRESENT) of both masks is recorded, together with the groups and checktypes of the edges.

The analysis of the stateruler is done in three different procedures: a check for overlap errors in masks that must fully overlap in the procedure 'extr_overlap', a check for errors in masks that must overlap in one direction (x or y) in 'extr_overlap1' and a check for gap errors in the procedure 'extr_profile'. In case of an overlap test checks are carried out for start segments in the first mask (that must be overlapped) and stop segments in the second mask. The error is immediately reported. Areas in the first mask with a checktype that do not equal zero, indicating that they originate from a sub_cell are not tested. If they should give an error this will be reported when the sub_cell itself is tested.

In the case of a gap test checks are carried out for stop and start segments of both masks. Two cases are distinguished:

- If an area from mask1 has an overlap with an area of mask2 anywhere no errors will be reported concerning distances between these two areas.

- If an area from mask1 has an overlap with an area of mask2 this will not lead to a suppression of errors that occur on other places between the two areas in the masks.

Because of the fact that in the first case an event that seems to be an error, later on may turn out not to be erroneous because of an overlap, the errors with a gapcheck are not directly reported but temporarily stored. Also a structure is kept indicating what groups of the two masks have an overlap. After all the edges have been processed the errors will be written, suppressing the errors between equivalent groups if this is wanted.

For a detailed description of dubcheck as it appears in the ICD release 1, see appendix D.
5. Results

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

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

Table5.I. Checker cpu times.

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

<table>
<thead>
<tr>
<th>cell</th>
<th>linear convert (seconds)</th>
<th>linear checker (seconds)</th>
<th>hierarchical convert (seconds)</th>
<th>hierarchical checker (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod2_fb</td>
<td>9</td>
<td>26</td>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>latch</td>
<td>10</td>
<td>24</td>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>select</td>
<td>8</td>
<td>20</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>feedback</td>
<td>13</td>
<td>28</td>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td>sel_reg8</td>
<td>32</td>
<td>95</td>
<td>29</td>
<td>56</td>
</tr>
<tr>
<td>rand_cnt</td>
<td>38</td>
<td>110</td>
<td>20</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 5.II. Comparison between hierarchical and linear expanded cells.

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

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

1. The exploitation of the hierarchy.

2. The generation of combination masks necessary for the checking of intermask rules.

3. The complexity of the scan itself.

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


8. APPENDIX A: Implementation of Technology

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

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

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

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

1. Width checks.
   To implement a width check on a (combination)mask, the following steps must be taken:
   a. In the case of a combination mask, the logical formula of that mask, if not yet present, must be included in the file nbool.d. For the format of this file see later on under the sub-section on file formats.
   b. The (combination)mask to be checked must be given in the input file of the program dimcheck. In this file the minimum width of the mask must be given also. For the format of this input file see the sub-section on file formats later on in this appendix.

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

2. Gap checks between items in one (combination)mask.
   To implement a gap check on a (combination)mask, the following steps must be taken:
   a. In the case of a combination mask, the logical formula of that mask, if not yet present, must be included in the file nbool.d. For the format
of this file see later on under the sub_section on file formats.

b. The (combination)mask to be checked must be given in the input file of the program dimcheck. In this file the minimum gap between areas of the mask must be given too. The possibility also exists to specify in this input a reduced gap in case the gaplength is smaller than a certain given value. For the format of this input file see the sub_section on file formats later on in this appendix.

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

3. Gap checks between two (combination) masks.
   To implement a rule for the gapcheck between two masks, the following steps must be taken:
   
   a. In the case that one or both masks are combination masks, the logical formulas of these masks, if not yet present, must be added to the file nbool.d (for format see sub_section on file formats).

   b. The masks between which the check has to be carried out must be added to the input file of dubcheck. The minimum gap between unrelated areas of the masks must be specified here too. The possibility here also exist to specify a reduced gap if the gaplength is smaller than a certain given value. Furthermore a kind must be specified in this input. This variable may have the values 0 or 1. If kind = 1 errors originating from areas from the two masks that do have an overlap somewhere will be reported; if kind = 0 they will not. As an example consider fig. 8.1.
fig. 8.1 Influence of kind.

If in this figure kind is 1, the error indicated will be reported; if kind is zero it will not because the two areas have an overlap (for format see the sub_section on file formats).

Examples of rules that can be tested this way are e.g. in the nmos process:
the gap between an undercrossing and unrelated diffusion (kind = 0) and the gap between unrelated poly and diffusion (kind = 1).

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

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

b. The masks concerned must be given in the input file for the program dubcheck. Here also the value of the overlap must be given and for kind the value 0 must be chosen (for format see the sub_section on file formats).

An example of a rule that can be tested this way is e.g. in the nmos technology: overlap of metal over a connect_cut etc.

5. Overlap checks of (combination) masks for an overlap in one direction.
To implement a rule to test an overlap of a combination(mask) over another one in one direction, the following steps must be taken:
a. In the case that one or both masks are combination masks the logical formulas for these masks, if not yet present, must be added to the file nbool.d (for format see sub_section on file formats).

b. The masks concerned must be given in the input file for the program dubcheck. Here also the value of the overlap must be given and for kind the value 1 must be chosen (for format see the sub_section on file formats).

An example of a rule that can be tested this way is e.g. in the nmos technology: overlap of poly over an active area etc.

8.1 file formats

In this sub_section the formats of the input files for the programs will be discussed.

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

Example:

```
nd_vln np_vln ni_vln nb_vln nx_vln nc_vln
nm_vln
nb_vln&nd_vln
: filenames
nb_vln&np_vln
: 0 2.4.1
nb_vln&np_vln&nd_vln
: 10 2.1+2.2
nb_vln&np_vln&nd_vln
: 6 2.2
nc_vln&nd_vln
: 1 4.6+4.4
nc_vln&np_vln
: 13 4.1+4.3.1
nc_vln&nm_vln
: 15 4.1+4.4
nc_vln&nd_vln&nm_vln
: 5 4.6.1
nc_vln&np_vln&nm_vln
: 3 4.3.2
np_vln&!nb_vln
: 2 5.4
nd_vln&np_vln&!nx_vln&!nb_vln
: 4 4.3.2
nd_vln&np_vln&!nx_vln
: 16 2.4
nx_vln&ind_vln&nx_vln&!np_vln
: 7 5.1+5.3
np_vln&nx_vln&np_vln&!nb_vln&ind_vln
: 8 3.4
nd_vln&nx_vln&ind_vln&!nb_vln&!np_vln
: 9 1.1.2
nd_vln&np_vln&!nb_vln&!nx_vln&!ni_vln
: 11 3.3 enh trans
nd_vln&np_vln&!nb_vln&!nx_vln&ni_vln
: 19 3.3 dep trans
np_vln&nm_vln
: 20 4.6.1
nd_vln&!nx_vln
: 14 3.5
```

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

2. The file proc1.d, read by the program dimcheck, contains the names of the files to be checked and their width and gap dimensions.

Example:

```
nd_vln 4 4 10 1.1+1.2
np_vln 4 4 3 10 3.1+3.2
nc_vln 4 0 4 10 4.1+4.4
nm_vln 5 4 3 10 4.7+4.8
```

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

1. The name of the file to be checked.
2. The minimum width of elements on the file. If it is zero no check will be carried out.
3. The minimum gap between two elements on the file. If it is zero no check will be carried out.
4. The minimum gap between elements on the file for short lengths of the gap.
5. The maximum length of the gap for which the reduced gap may be applied.
6. A string indicating the design rule(s) involved.

In this example only primary vln files are used. However one may also use vln files made by nbool, so files bool_nn.
3. The file proc2.d, read by the program dubcheck, contains the names of the files to be checked and the gap and overlap dimensions.

Example:

```
bool_0  np_vln  0  2  0  0  1  2.4.1
bool_1  np_vln  0  3  0  0  1  4.6
bool_3   bool_4  0  4  0  0  1  4.3.2
bool_5  bool_20  0  4  0  0  1  4.6.1
bool_10 bool_6  2  0  0  0  0  2.2
bool_12 nd_vln  0  2  0  0  1  2.4.2
bool_13 bool_4  0  3  0  0  1  4.3.1
bool_13 np_vln  3  0  0  0  0  4.2
bool_1  nd_vln  1  0  0  0  0  4.5.1
bool_15 nm_vln  2  0  0  0  0  4.9
bool_19 ni_vln  2  0  0  0  0  2.8
nx_vln nd_vln  0  2  0  0  1  5.2
nx_vln bool_12  0  2  0  0  1  5.5
nx_vln bool_0  0  2  0  0  1  5.6
nx_vln bool_2  0  3  0  0  1  5.4
nb_vln bool_16  0  3  0  0  1  2.4
ni_vln bool_4  0  2  0  0  1  2.9
np_vln bool_14  0  2  0  0  0  3.5
```

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

1. The first file involved with the operation. In case of overlap check this is the file of whose elements have to be overlapped.

2. The second file involved with the operation. In case of overlap check this is the file whose elements have to overlap the elements of the first file.

3. The overlap the second file must have over the first file. If it is zero, no overlap check will be carried out.

4. The minimal gap between non overlapping elements of the first and second file. If it is zero no check will be carried out.

5. The minimal gap that must be maintained if the length of the gap is only small.

6. The maximum gaplength for which the reduced gap value may be applied.
7. The value of the variable kind. For overlap checks kind = 0 means a full overlap check; kind = 1 a overlap check in one direction. For gap checks kind = 0 means suppressing of errors between overlapping areas, kind = 1 means no suppression.

8. A string indicating which design rules are involved.
9. APPENDIX B: The program nbool

9.1 Introduction

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

nbool [-c] [-n]

If called without arguments or with the argument -c nbool will check for hierarchical errors; if called with argument -n it will not. If nbool has to operate on input files that themselves are boolean combination files, the last option has to be chosen, otherwise false error messages will occur, because nbool then does not know what terminal masks are involved with the boolean masks. Like the program dimcheck it is based on the concept of the stateruler.

As its input the program needs:

- The file exp_dat containing the model(s) the program has to be applied to.
- The file nbool.d containing the logical formulas the program has to perform upon its input files.
- The vln files (edge files) involved in the logical combinations.

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

The main parts of the program are:

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

- The part that analyses the stateruler and determines from that what edges have to be output. This will be described in the part about extract_profile.

- The part that adds the group_numbers to the vln files made. This will be described in the part about add_groupnumbers.

9.2 Decoding the design rules

Most of the design rules involve more than one mask. To check these rules masks must be made containing logical combinations of the masks needed for that particular check. The formulas of the masks that must be made for all the checks of a certain technology must be given on the file nbool.d, which will be described in appendix A. This section will describe the way these formulas are stored in memory to allow for an efficient way to produce all output masks wanted in one pass of the algorithm. In the program this is done in the routine mk_formstruct.

This routine starts by reading the first line of the file nbool.d which names all masks involved in the formulas to be made.

Then the routine ini_heap is entered which makes an input structure. For each file listed in the input line, and if the hierarchy must be checked also for each terminal file, a structure is set up containing:

- The name of the mask involved.

- The binary number of the mask. If the name of the mask is known to the process, the corresponding mask number is taken from it. If not it gets a number twice as high as the previous unknown mask, starting at the mask_number twice as high as the highest mask_number known in the process.

- The mask_type of the mask. If the mask is known to the process the mask_type is copied from it. So terminal masks become 1, connection masks become 2 and the others become 0. For masks unknown to the process the mask_type is set to BOOLEAN (=3).

- The pointer to the vln file

- The data of the first edge in the vln file. So x_position, y_bottom, y_top, edge_type and check_type.

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

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

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

- A number of buffers to temporarily store the edge data before it is written to the output file.

- A number curr_place indicating which buffer has been filled last. Initially this variable is set to -1 to indicate that all buffers are free.

- A pointer to a list of min_term structures, which will be explained later on

- A vulnerability mask containing all masks present in the formula. This variable is not strictly needed, but added for efficiency reasons.

- A pointer to the next form_struct, or if there is not any a NULL_pointer.

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

- The masks that must be present for a mask combination to be part of the term of that formula.

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

These variables are updated in the procedure 'update_masks' each time a '&' or '|' character is discovered. If a term of the formula is finished (a | character discovered or end of formula) these variables mask and not_mask are placed in a structure min_term and this structure is added to the list of min_term structures of the formula. This is done in the procedure 'add_minterm'. The variable vuln_mask of the form_structure there is updated too. Upon leaving the procedure 'mk_formstruct' we thus have created a structure like the one shown in fig.9.2.1.
fig.9.2.1: the formula structure.
9.3 The stateruler

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

- yb: the bottom of the field.
- yt: the top of the field.
- chk_type: the check_type of the layers in the field. If the layers have different check_types this value is set to DIFF_CT (= -2).
- p_check: A pointer to a structure in which the check_types are stored per layer. If all layers have the same check_type there is no need for such a structure and p_check is a NULL_pointer.
- p_chg_ct: A pointer to a list of structures which contain the old check_type and the mask a change of check_type occurred in. If no change of check_type occurred this is a NULL_pointer.
- mask_past: This variable bitwise contains the layers present in the field before the edges at the present x_value are installed.
- mask_fut: This variable bitwise contains the layers that are present after the insertion of the edges at the present x_value.
- ov_mask: This variable contains bitwise the layers in which an overflow of layers of different check_type has occurred.
- next: A pointer to the next stateruler field.
- prev: A pointer to the previous stateruler field.

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

After being updated a new stateruler is built for the next x_value until all edges have been read. A schema of the operations is given in fig.9.3.1.
fig. 9.3.1. Stateruler main flow.
9.4 The hierarchy check

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

- The ov_mask.
  If a bit in this mask is set, indicating that an overlap in the corresponding mask has occurred, the following is done:
  - If the mask in which the overlap occurred is a connection mask, a check is carried out to see if the corresponding terminal mask is present. If not an error message is generated, telling where the error occurred and in which mask.
  - If the layer is not a connection mask a warning massage is generated, telling where the overlap took place and in which mask.

- The check_types.
  If the variable chk_type in the stateruler field is DIFF_CT, indicating that in the field layers with different check_types are present, the following actions are taken:
  A check is made to see if the difference is caused by a check_type 0 in a connection mask with the presence of a terminal in the same layer, indicating an overlap permitted. In this case no messages are generated. If the difference is not caused by the situation described above, a warning massage is generated telling the place where different check types occur, and the checktypes of the layers present.

- Change of checktype.
  If a change of checktype in the y_direction occurs the following steps are taken:
  - If the change takes place in a connection layer a check is carried out if a terminal is present there. If not, an ERROR message is generated, stating where the error occurred and in which layer.
  - If the change takes place in another layer a WARNING is generated, stating the place of the change of checktype and the layer it occurred in.
9.5 Analysis of the state register

The analysis of the state register is carried out in the procedure 'extr_profile'. It finds the edges that have to be output in the VIn file of the corresponding formulas. In this procedure a loop is set up, which examines each state register field. If the values of mask_past and mask_fut differ, indicating that one or more layers have changed state, the procedure 'buff_edge', which does the actual work is called. The main flow of this procedure is given in fig. 9.5.1.

The way the program checks if a certain mask combination belongs to a formula is done using the structure made with mk_formstruct. The mask combination is compared to the masks and not_masks of the min_terms of the formula. If all the masks present in the variable mask of the min_term structure are also present in the mask combination to check, and the masks set in the variable not_mask do not appear in the mask combination to check, the mask combination belongs to that min_term, and hence to the formula.

According to the presence of mask_past and mask_fut in the formula the pres_flag is set. If mask_past belongs to the formula and mask_fut does not, pres_flag is set to 1, indicating a stop edge. If mask_past does not belong to the formula and mask_fut does, pres_flag is set to -1, indicating a start edge. If mask_past and mask_fut both belong to the formula, or if they both do not belong to the formula, pres_flag is set to 0, indicating that no edge has to be output.

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

- The bottom value of the newly found edge and the top value of the last buffered edge of the formula are the same and so are their x_positions. In this case the last buffered edge is updated, i.e. its top value and its connection type are updated. This is done in the procedure 'update_edge'.

- If the values mentioned above do not coincide, the edge is added to the next place in the buffer. If all buffers of the formula have been filled, the buffer is appended to the file, whose name is given in the f_name variable in the form_structure. These actions are carried out in the procedure 'add_edge'.

2.36
fig. 9.5.1. buff_edge main flow.
The generation of the group_numbers(connectivity).

The group_numbers of the edges indicate to which connected region they belong. They are generated after nbool has generated the edges. This is done on temporary files, which for efficiency reasons are in binary format. They have a boolean name, with btl added to it. For example bool_2btl. Now the files generated are read one by one and a pointer structure is set up in the same way as it is done in i.e. the program 'makevin'. The pointers are added to the file and written on a file with the addition of bt2 (e.g.bool_2bt2), and the bt1_file is removed from the system. The bt2 file then is read and the pointers are replaced by their corresponding group_numbers. Then the edges are written to the boolean file that remains in existence and is used by the programs dimcheck and dubcheck. These files (e.g bool_2) are in the known vln_format. The bt2_files are also removed from the system.
10. APPENDIX C: The program dimcheck

10.1 Introduction

After the generation of the combination masks, dimcheck is called whose task it is to check a (combination) mask for gap and width errors.

The program must be called as:

dimcheck [-z][-n]

If the program is called without options, or with the option -z, dimcheck will report errors about areas that have just one point in common (see fig.10.1.1); if called with the option -n these occasions will not be reported.

```
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXX
XXXXX
```

fig.10.1.1: Areas with one point in common.

The last option is used for checks of boolean files made by the program 'nbool'.

As its input the program needs the following files:

- The file exp_dat containing the model(s) dimcheck has to be applied to.
- The file procl.d containing the layers to check and the gaps and widths permitted.
- The vln files mentioned in the procl.d file.

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

The program may be divided into two mayor parts:

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

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

10.2 Making and updating the stateruler.

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

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

Upon initiation the stateruler consists of one field, reaching from -MAXINT to MAXINT, with lay_status NOT_PRESENT.
After initiation a loop is started in which edges are read from the vln file (procedure get_vln) and inserted into the stateruler (procedure insert_edge). The loop is continued until all edges with the same x_value have been inserted. The stateruler for that x_value then is completed, and an analysis of the stateruler then will take place (procedure extr_profile). The stateruler is updated (procedure update_srl) and new vln files are read and inserted to form the stateruler for the next x_position. This process is repeated until all edges have been read from the vln file. The process described above is carried out in the procedure main_check.

As stated above the insertion of new edges in the stateruler is done in the procedure insert_edge. In this procedure the fields of the stateruler are scanned from the current position to the topmost position to see if an overlap with the edge to insert is present. If this is the case and the bottom values of the field and the new edge do not coincide, the stateruler field is split into two and the variables are copied from the old field. As long as the top value of the edge is greater then the top value of the stateruler fields, the latter are updated. If the top value of the new edge becomes smaller as the top value of the stateruler field, again a split is carried out and the bottom field of the two newly created fields is updated. The splitting of the fields is carried out in the procedure 'split_fld', the updation of the fields is done in the procedure 'update_fld'. An example is shown in fig.10.2.1.

![fig. 10.2.1 Building the stateruler.](image-url)
In the procedure 'update_sr' the stateruler is updated after being analysed. This means:

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

10.3 The analysis of the stateruler.

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

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

- lay_status = CHG_TO_PRESENT.
  This means that a new area has started. In this case the following checks are carried out:
  - A check to see if the distance between the previous edge and the new edge is great enough (procedure 'check_xgap').
  - If in the previous stateruler field the layer is not present a check to see if previous edges are not too close to the bottom of the new edge (procedure 'check_g_circle').
- If in the next stateruler field the layer is not present a check to see if previous edges are not too close to the top of the new edge (procedure 'check_g_circle').

- If in the previous stateruler field the lay_status is not CHG_TO_PRESENT and in the next stateruler field the lay_status is not PRESENT (in which cases an error, if any, already has been reported), a check of the y_width of the edge starting in the stateruler field is carried out (procedure 'check_ywidth').

- lay_status = CHG_TO_NOTPRESENT
  This means that an area has stopped. In this case the following checks are done:

  - A check to see if the area that stopped was not too small in the x_direction (procedure 'check_xwidth').

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

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

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

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

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

2.43
If the layer is not present in the next stateruler field a gap check is done to see if the gap between the stopped area and the first area over it is not too small (procedure 'check_ygap').

- \texttt{lay\_status} = \texttt{NOT\_PRESENT}.
  This means that no change of \texttt{lay\_status} has taken place, so nothing needs to be checked.

All of the width and gap checks mentioned above are only done if the width or gap flag are set in the procedure main. This has been done because in some cases one only wants a width or only a gap check.

In the check routines check_xgap etc. gap errors will not be generated for:

- Gaps are between areas with the same groupnumber, meaning that these areas are connected,
- Gaps between areas that have the same checktype, provided that the checktype not equals zero. This situation means a gap between two areas of the same sub_model, and this is checked when the sub_model is tested.

In the check routines check_xwidth etc. width errors will not be generated if the edges have the same checktype (except if it is zero). This situation means that the area originates from a submodel and has already been tested there.
11. APPENDIX D: The program dubcheck

11.1 Introduction

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

```
dubcheck
```

As its input the program needs the following files:

- The file `exp_dat` containing the model(s) dubcheck has to be applied to.
- The file `proc2.d` containing the layers to check and the minimum overlaps and gaps.
- The `vIn` files mentioned in the `proc2.d` file.

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

The program may be divided into two mayor parts:

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

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

11.2 Making and updating the stateruler.

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

- \texttt{xstart[0]}: The \texttt{x} position of the previous edge of \texttt{mask1} in the field.
- \texttt{xstart[1]}: The \texttt{x} position of the previous edge of \texttt{mask2} in the field.
- $yb$: The bottom of the field.
- $yt$: The top of the field.

- $\text{lay\_status}[0]$: The status of mask1. This may be:
  - $\text{NOT\_PRESENT}$: This means that the layer is not present at the stateruler position.
  - $\text{CHG\_TO\_PRESENT}$: This means that the layer starts at the stateruler position.
  - $\text{CHG\_TO\_NOT\_PRESENT}$: This means that the layer stops at the stateruler position.
  - $\text{PRESENT}$: This means that the layer is present at the stateruler position.

- $\text{lay\_status}[1]$: The status of mask2.

- $\text{group}[0]$: The group of mask1 in the stateruler field.
- $\text{group}[1]$: The group of mask2 in the stateruler field.

- $\text{chk\_type}[0]$: The checktype of mask1 in the stateruler field.
- $\text{chk\_type}[1]$: The checktype of mask2 in the stateruler field.

- $\text{next}$: A pointer to the next stateruler field.
- $\text{prev}$: A pointer to the previous stateruler field.

Upon initiation, the stateruler consists of one field, reaching from $-\text{MAXINT}$ to $\text{MAXINT}$, with $\text{lay\_status} \text{ NOT\_PRESENT}$, $\text{xstart} = -\text{MAXINT}$, $\text{next}$ and $\text{prev}$ pointing to the field itself and the other variables set to zero. After the initiation a loop is started reading edges from the vIn files (procedure get_vln), selecting the one with the smallest $x\_value$ and the smallest value of $y\_bottom$ and inserting them into the stateruler (procedure insert_edge). The loop is continued until all edges with the same $x\_value$ have been inserted. The stateruler for that $x\_value$ then is completed, and an analysis of the stateruler then will take place (procedures $\text{extr\_profile}$, $\text{extr\_overlap}$ and $\text{extr\_overlap1}$). The stateruler is updated (procedure $\text{update\_sr}$) and new vIn files are read and inserted to form the stateruler for the next $x\_position$. This process is repeated until all edges have been read from the vIn file. The process described above is carried out in the procedure 2.46.
As stated above the insertion of new edges in the stateruler is done in the procedure insert_edge. This procedure is similar to the one used in the program dimcheck, with only a difference in the variables that are present in the stateruler fields.

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

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

11.3 The analysis of the stateruler.

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

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

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

- If the masks have no overlap the distance to the
  last recorded edge of mask2 in the field is
  checked.

- If lay_status[0] of the previous or next field in
  the staterule is NOT_PRESENT checks are carried
  out to see if no error exists in the areas left
  under respectively left above the edge.

lay_status[1] = CHG_TO_PRESENT.
This means that an area in mask2 is starting. In this
case the same checks are carried out with respect to
mask1, as in the previous case with respect to mask2.

- lay_status[0] = CHG_TO_NOTPRESENT
  This means that an area in mask1 has stopped. In this
case the following checks are done:

  - A check is carried out to see if mask1 and mask2
do have an overlap. An equivalence of groups then
  is set up again.

  - If no overlap occurs a check is carried out to see
    if no error occurs at the bottom of the field and
    a check is carried out to see if no error occurs
    at the top of the field.

- lay_status[1] = CHG_TO_NOTPRESENT.
  This means that an area in mask2 has stopped. In this
case the same checks are carried out with respect to
  mask1, as in the previous case with respect to mask2.

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

11.3.2 detection_of_overlap_errors_of_kind_0 In the procedure extr_overlap checks are carried out to see if all areas of the first layer are fully overlapped by a distance overlap by the areas of layer 2. According to the lay_status of the masks in the stateruler fields the following checks are carried out:

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

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

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

11.3.3 detection_of_overlap_errors_of_kind_1 In the procedure extr_overlap1 checks are carried out to see if all areas of the first layer are overlapped by a distance
overlap by the areas of layer 2 in the x_ or y_direction. According to the lay_status of the masks in the stateruler fields the following checks are carried out:

- lay_status[0] = CHG_TO_PRESENT. In this case the next checks are carried out:
  - if lay_status[1] = PRESENT, a check is carried out to see if the overlapping area started at least a distance overlap earlier.
  - if lay_status[1] = CHG_TO_PRESENT, checks are carried out to see if the overlaps over the starting area of layer1 to the top and bottom are great enough.
  - if lay_status[1] = NOT_PRESENT or CHG_TO_NOTPRESENT an error is generated.

- lay_status[1] = CHG_TO_NOTPRESENT. In this case the next checks are carried out:
  - if lay_status[0] = NOT_PRESENT, a check is carried out if the stop edge of the area to be overlapped has occurred at least a distance overlap before.
  - if in the previous stateruler field lay_status[1] = PRESENT and lay_status[0] = NOT_PRESENT, a check is carried out to see if over a distance of at least overlap under the stateruler field no area in mask1 is present.
  - if in the next stateruler field lay_status[1] = PRESENT and lay_status[0] = NOT_PRESENT, a check is carried out to see if over a distance of at least overlap over the stateruler field no area in mask1 is present.

Errors are reported immediately in this case. The check procedures are such that no errors are generated if the area of mask1 in the stateruler field has a checktype not equal zero, indicating it originates from a submodel. In this case the error will already be detected when the submodel is checked.
A TECHNOLOGY INDEPENDENT HIERARCHICAL LAYOUT TO CIRCUIT EXTRACTOR USING A FINITE STATE APPROACH

Fons de Lange
Jurgen Annevelink

Delft University of Technology
Department of Electrical Engineering

ABSTRACT

The paper describes a layout to circuit extractor program that can operate on a hierarchical description of the layout data. The extractor is based on a linear finite state scanning algorithm to recognize electrical functions in layout, and is suited to extract circuit layouts in different kinds of technology (NMOS, CMOS, IIL, BIPOLAR).

1. Introduction

The growing complexity of IC-designs makes the development of hierarchical CAD-tools necessary [1,2]. In this paper we describe the design and implementation of an hierarchical layout to circuit extractor. The program exploits the hierarchy and repetition present in the layout description of an IC. As a result the extracted network description has the same hierarchy as its corresponding layout description. To take advantage of the hierarchy in a layout description, the extractor recognizes the connections of a cell with its subcells. This allows the subcells of a cell to be extracted independent of the cell itself. Moreover, the extractor has to extract the cells used in the layout description of an IC only once. By doing so, the run time and memory requirements are not longer a function of the number of layout features of the complete IC, but only of the number of features present in the individual cells, from which the IC is build. To further increase the efficiency of the extractor, the extraction algorithms are based on a state ruler scan algorithm [3], which is linear in the number of features encountered in the cell.

The extraction program discussed here is based on the extractor presented in [4]. It is similar in that it operates on a hierarchical layout description in the form of vertical line segment files, uses a a linear state ruler scan algorithm, and is

This paper was produced as part of Task III of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
based on the detection of element boundaries. It is different with respect to the implementation of the algorithms that recognize elements, the algorithms that update the element geometries, the algorithms that determine nodes, and finally the data structure of the extracted elements.

As a result, the new extractor is capable of extracting layouts in different technologies and with varying levels of detail, provided that a suitable technology description is available. The algorithms used by the extractor are general, meaning that it is possible to extract different types of elements (capacitances, transistors, etc.) with varying level of detail. To calculate the electrical properties of these elements, the extractor determines their geometrical properties. The set of geometrical properties is appropriate to obtain the electrical modeling parameters of extracted elements, as well as to decide if extracted elements should be split up in sub elements, in order to improve the accuracy of the resulting network description. If necessary, i.e. if specified in the technology description file, the extractor can generate the actual line segment description of an element instance. This line segment description can then be used by further postprocessing programs, that can do e.g. precise resistance and transmission line calculations. The internal network data structure made by the extractor (cf. section 3), can optionally be written to a (binary) file after the extraction is completed. As a result the extractor can be combined with different post-processors, generating input files for different types of simulators. Moreover, some functions have already been implemented that operate directly on the extractor’s internal data structure and that generate input descriptions for the switch-level simulator SLS [5] and the circuit-simulator SPICE. Currently, these functions will produce a hierarchical network description that contains only transistors and capacitors. However, functions and postprocessors that are capable of accurate resistance and transmission line calculations are in development. The postprocessor programs compute the electrical parameters of the elements, based on the geometrical parameters determined by the extractor. The postprocessor programs can consult the technology description to retrieve the values of the electrical process parameters, e.g. square- and side wall capacitances. The extractor itself doesn’t use these parameters.

The contents of the paper is now as follows. In section 2 we outline the basic extract algorithm and explain how the hierarchical information in the line segment files allows us to consider a cell almost independent of its subcells. Next, in section 3 we describe the format of the technology description, as well as the datastructures required to represent the technology information, including the elements to be extracted, in the extractor. In section 4 we specify the Element Recognition Algorithm (ERA). The ERA uses the information read from the technology file, to determine the boundary edges of the elements that are extracted. It then calls a routine that maintains and updates the network data structures. The network data structures are discussed in more detail in section 5. In section 6 we give an example and some performance data. Section 7 gives some conclusions. Finally, we have three appendices. In appendix A we give an example of the extraction of a (very) simple layout to illustrate the use of the algorithms, and to show how the datastructures are actually build

2.52
Appendix B gives an example of a technology description, while appendix C contains the user manual of the extractor.

2. Principle Of Extract Operation

The extractor operates on a description of the layout as a set of sorted vertical line segment files. The line segment files are derived from the layout description by a preprocessor which converts the box-information to line segments (or edges), and assigns a unique group number to polygons [3]. Individual cells may be connected only via what we call terminals. A terminal is an ordinary rectangle (or polygon) on a layout mask which has a name. It shows a place where layout features of different cells may overlap. Terminals are also represented by line segment files as if they are ordinary polygons. Thus the line segment files represent individual process masks, but also convey electrical interconnect information.

An example of a layout pattern and its edge representation is given in figure 1.

Since we consider only paraxial layouts, the vertical edges suffice to describe the polygon. An edge is represented by a record consisting of six parameters, resp. the X or horizontal value, Yb and Yt the two vertical values, the edge_type, the check_type, and a groupnumber.

The group_number of an edge relates the edges belonging to the
same polygon. For a terminal edge file it also relates a polygon with a terminal.

The edge-types recognized by the extractor are combinations of five basic types, respectively START and STOP, ChangCheckType (CCT), StartOverlap (SAO) and StopOverlap (SOO). We speak of a START edge when the interior of the polygon bounded by the edge is situated on the right side of the edge. Conversely we speak of a STOP edge when the interior is on the left. The other edge-types, CCT, SAO and SOO, in combination with the checktype information of an edge, are used to add hierarchical information to the line segment description. An edge of type SAO or SOO is found at the boundary of an area where a polygon defined in a cell overlaps a polygon defined in one of its subcells. The checktype of an edge makes it possible to distinguish between the polygons (or parts of polygons) defined in a cell, and the polygons defined in the subcells of the cell. It also allows edges stemming from polygons defined in different subcells to be distinguished from each other. The checktype of an area within a polygon can be changed using an edge whose edge-type is CCT. Figure 1(a) shows a situation where a feature of a cell overlaps the bounding box, and a terminal, of one of its subcells. By looking at the corresponding line segment description (figure 1(b)), we see that we can recover the essential hierarchical information by taking into account the edge-type and checktype of the line segments.

From the edge files the extractor makes so-called state ruler profiles. A state ruler profile is a vertical transverse of the layout. It consists of a linked list of fields, characterized by a bottom and top value and a set of layer states. The bottom and top values of the field are non-overlapping, while the profile is sorted about the bottom value. The layer states are a set of Boolean variables, giving the past and future state of each layer. The state of a layer is TRUE when the layer is present; it is FALSE when the layer is absent. The past state of a layer is the state of the layer between the previous profile and the current profile, the future state is the state of the layer between the current profile and the next profile. We say that a field or a layer has changed state if the past and future states are not equal. The edges that are inserted in the state-ruler profiles are selected from the line segment files in such a way that the extractor only has to deal with polygons whose checktype is that of the cell to be extracted (i.e. the top-level cell). To that end the input routines of the extractor can change the edge-type of an edge from CCT to START or STOP. START or STOP edges whose checktype is not equal to that of the top-level cell are skipped completely. The overlap information is explicitly represented in the statenruler profiles, and can be used to check that the areas in which the overlap happens is correctly covered by terminals.

The state ruler profiles are scanned by the element recognition algorithm (ERA). For each field that has changed state this algorithm determines whether the field forms a boundary edge of an element. This is described in more detail in Section 4.

If a boundary edge is recognized the element recognition algorithm performs an appropriate action to update the description of the network element from which the edge is part. The element that must be updated depends on the environment of the field in the
state ruler profile. The elements are linked with the state ruler fields and are represented by element descriptors, linked together in element lists. These data structures will be explained in more detail in Section 5.

By scanning the entire layout, making successive state ruler profiles and scanning these profiles as well, the extractor determines all network elements, as well as their connectedness.

A simplified algorithm of the routine 'extract' is given in figure 2. From the algorithm we see that a state ruler profile is made by repeatedly calling select_edge and insert_edge. The procedure select_edge, will select the edge with the smallest horizontal and vertical position, possessing an appropriate edge_type, from the line segment files, while the procedure insert_edge will insert this edge at the correct vertical position in the state ruler removing overlap between adjacent fields.

```plaintext
edge_status : integer ;
edge : record of
    y_low, y_high : integer ;
group_no : integer ;
edge_type : integer ;
layer_code : STRING of char ;
end ;
edge_pos : integer ;
sr_pos : integer ; [ x_value of stateruler_profile ]

Initialize local variables ;
select_edge(edge_status, edge, edge_pos) ;
sr_pos = edge_pos ;
while (edge_status is not EOF)
loop
    do [ make a stateruler_profile ]
    loop
        insert_edge(edge, sr_pos) ;
        select_edge(edge_status, edge, edge_pos) ;
    end loop
    while (edge_status <> EOF and edge_pos == sr_pos)
recognize_elements(sr_pos) ;
sr_pos = edge_pos ;
end loop
```

Figure 2 : Algorithm of routine 'extract'.

Each profile in turn is scanned by the routine recognize_elements. This routine implements the element recognition algorithm (ERA), and calls the network element construction / update routines.
3. Technology Description

The technology description file contains all information specific for a particular technology, e.g. NMOS or CMOS. Among this information, and of particular importance for the extractor, is the specification of the layers that can be used in a process. Layers are specified by their name and their type. The type of a layer distinguishes between auxiliary layers, implantation layers, and interconnect layers. Auxiliary layers are ignored by the extractor. Interconnect layers form the conducting patterns in a chip layout, so in a chip all interconnections will always be made via such layers. If the layer is of type interconnect, an associated terminal layer must be specified for it. Given the interconnect layers, the extractor is able to determine where the nodes of an element are located. Another important part of the technology description is the specification of the elements to be extracted.

3.1 Specification of Network Elements

The basic method of specifying an element is as a collection of minterms, forming a so-called Element Condition (EC). A minterm in an element condition is a boolean expression in which the names of the variables are replaced by layernames. The value of the variable is true when, at a given point, the indicated layer is present, otherwise it is false. We require that the EC's are mutually exclusive, i.e. at any one point at most one minterm may evaluate to true. This requirement is not necessary, but can be easily fulfilled, and, as it turns out, will greatly simplify the implementation of the extract algorithms.

Elements can thus be specified by means of a set of boolean expression consisting of layernames. The presence of an element at a particular position can be easily determined by evaluating all of the minterms in a specific EC. An example of an element condition for an enhancement transistor in a standard NMOS technology could be:

```
enh_trans :
  ecl_enh_tr := poly AND diffusion AND NOT crossing AND NOT buried AND NOT implant;
```

The format is rather simple, a list of minterms preceded by the name of the element. Note that it is not necessary to define the nodes of an element, when the interconnect layers are known. This is very appropriate for MOS-technologies, and is sufficient for various other technologies (i.e., bipolar). Nodes are placed at the boundary edges of elements and element minterms, when interconnect layers that are referenced in the elements EC's touch these edges (cf. figure 3). In figure 3, the diffusion and poly layers are interconnect layers that are present at both sides of the indicated edges. The transistor is only present when both poly and diffusion are present. Therefore, in accordance with what was said about the implicit creation of nodes, nodes are created at these edges.

As can be easily seen the transistor in figure 3 will have four nodes. Since a transistor has only three nodes, the extractor has a facility that allows the nodes of an element to be merged during extraction, even if the nodes are located in different interconnect layers. The poly nodes of the transistor of figure 3 will
Figure 3: Nodes in a nmos enhancement transistor circuit layout

normally be merged during extraction.

When it is desired to merge all nodes in or between one or more interconnect layers for a certain minterm of an element condition, this can be specified by putting parentheses around the layer names in the EC. A transistor and a metal poly contact can thus be specified by:

enhancement_transistor:
encl_enh_tr := (poly) AND diffusion AND NOT crossing AND NOT buried AND NOT implant;
metal_poly_contact:
encl_mpcnt := (metal AND poly) AND contact AND NOT diffusion;

meaning that the transistor only has one node in poly, and that the metal-poly-contact has only one node which is the same in metal as well as poly. Besides the specification of elements/minterms to be recognized, also specification of ignore elements/minterms is allowed. These elements/minterms specify some layer combinations the extractor has to ignore when it recognizes them in layout. An example of a layer combination that should be ignored (in the Delft NMOS process), is:
crossing AND NOT poly AND NOT diffusion AND NOT metal

When only certain types of elements must be extracted, while others must be skipped, one is allowed to specify the unwanted elements to be "IGNORE". A "IGNORE" element is characterized by having the NULL-character as name.

From the element specification a data structure is build which makes the recognition of elements possible.

3.2 Element Recognition Data Structure

For every specified minterm in an element condition, a so-called minterm info descriptor is made. Each minterm has a name and a pointer to an element info descriptor of which it is part. On it's turn the element info descriptor has pointers to all it's composing minterm info descriptors. The element info descriptor also has a name and a pointer to link all the specified elements in a list, and finally it contains an array of pointers to minterm info descriptors. This is illustrated in figure 4.
When the number of layers NLAYS that are used in a technology is known to the extractor, it creates an array of pointers to minterms (layer combinations that are part of an element) of size \(2^{NLAYS}\). This way the number of elements in the array is equal to the number of possible layer combinations, so that it is possible to associate an element of the array with a specific minterm. The index of the array element is determined by the layers that are present in the minterm. Each layer has a bit mask which contains "NLAYS"-bits in which only one bit is set to "1". Every different layer combination occupies another bit "set to 1", so that any layer combination can be represented by only one word of bit length 'NLAYS', in which the bits corresponding with the present layers are set to 1.

In case a minterm (layer combination) is specified that doesn't specify the presence or absence of one or more layers, more than one array element will be associated with the minterm. The 'unused' layers can be seen as don't care conditions. In case different minterms can be associated with a single layer combination, the extractor will report an error and skip the layout extraction. In figure 5 the data structure of the minterm info descriptors, element info descriptors and minterm array is shown. The NULL-pointers in the minterm array of figure 5 indicate that the layer combinations forming the addresses of the NULL-pointers, don't exist or are not allowed, meaning that no minterms have been specified for these layer combinations.

The most important advantage of the proposed data structure is the speed at which elements and minterms can be recognized. This is due to the fact that the extractor doesn't need to search to find the minterm corresponding to a specific layer combination. However when the number of layers becomes "large", the array size may become too big to fit easily in memory, and the algorithm that constructs the data structure from the technology description may
Figure 5. Data Structure For The Element Recognition Algorithm.

become too slow. This problem can be overcome by considering that the number of layers used in any technology will seldom be larger than 12 to 15, which causes no problems to memory and speed. Other types of data structures like the binary tree and the hash table are less memory costing, and also fast, but not as fast as the proposed data structure. This is, because the binary tree search algorithm must check separately for each specified layer if it is present or not, while the hash table algorithm has to calculate the hash table index to find the linked list of elements/minterms for each present layer combination.
4. The Element Recognition Algorithm

The operation of the algorithm that recognizes elements is now very straightforward. It will get its layer information from the state ruler field that is currently being processed.

![Algorithm Code]

A state ruler field contains among other things the layer state variables `previous_layers` and `current_layers`, with `previous_layers` the state of the layers in the previous state ruler profile, and `current_layers` the state of the layers in the current state ruler profile (see section 2). The element recognition algorithm takes the current and previous layer combinations from the state ruler field to find the minterms associated with the state ruler field. If these minterms are not equal, the network data structure has to be updated, because a boundary edge of an element has been recognized. The routine that performs this action is shown in figure 6. When a start or stop edge of an element is recognized it is inserted into the network description data structure. A
start edge of an element means that the interior of the element is at the right hand side of the edge. Conversely, a stop edge of an element means that the interior of the element is located at the left hand side of the edge. Now the routines that construct and update the network data structure are called. This is the subject of the next section.

5. Construction Of The Network Data Structure

5.1 General Structure

The network data structure is illustrated in figure 7. It shows the state ruler, i.e. the linked list of fields that represent the vertical cross section of the layout at a certain horizontal position (section 2, 3), and the lists of instances of the elements that are recognized so far in layout. An element instance descriptor (EID) is made by the routines insert_start/stop_edge called in the element recognition algorithm (section 3), whenever an element with well defined boundaries is recognized in layout. An EID holds geometrical and nodal information that is sufficient to obtain all the relevant electrical properties of the recognized element instance.

Every EID is linked with the state ruler field in which it is found. Moreover, a state ruler field contains a pointer to the minterm-, and element info descriptor it belongs to. The state ruler field also contains information concerning the boundary edge which caused it to be created. From this information the network update routines can update the geometrical parameters and nodes of the recognized elements. The network itself is represented by a data structure of element lists, consisting of EID's. For each type of element the extractor recognizes, there is a separate EID-list that is linked with its corresponding element info descriptor as described in section 3.2.
Figure 7: Network data structure.

The algorithm of the network element construction/update routine (insert_start/stop_edge) is given in figure 8. The algorithm checks if the same element is present in the state ruler field (c_sr.up) above the current field (c_sr), and if these two fields are connected (top of current sr_field touches bottom of upper sr_field, or c_sr.up.bot = c_sr.top). Now the EID in the current state ruler field is merged with the EID in the upper sr_field. Merging means that all nodal and geometrical information is copied from one EID to the other, while the first EID sets a pointer to the updated EID. The same actions are applied for a connected element in the lower sr_field. The merged empty EID is not removed from the data structure, because it might have some links with other state ruler fields.

When no upper or lower connections are present, a new EID is created, and linked with the current sr_field. In case no EID is linked with the current sr_field and merging of EID takes place, the pointer to the EID in the lower or upper sr_field is copied into the current field.
if (boundary edge of element is START edge)
    [interior of element is at right hand side of vertic. edge]
begin
if (c_sr.current_element != c_sr.previous_element)
    [ current state ruler field is denoted by c_sr]
begin
    element_connect_type = NULL_CONNECT;
    if (c_sr.down.top == c_sr.bot AND
        c_sr.down.current_element == c_sr.current_element)
        [ lower and current sr_field are connected AND
        elements in current and lower sr_field are equal]
    begin
        merge_elements(c_sr.down, c_sr)
        element_connect_type += DOWN_CONNECT;
    end
    if (c_sr.up.bot == c_sr.top AND
        c_sr.up.current_element == c_sr.current_element)
        [ upper and current sr_field are connected
        && element in current and upper sr_field are equal]
    begin
        merge_elements(c_sr, c_sr.up)
        element_connect_type += UP_CONNECT;
    end
    if (element_connect_type == NULL_CONNECT)
    begin
        create new EID;
        link EID with current sr_field c_sr;
        link EID in element_descr list
        of element info descriptor;
    end
    update element geometries;
end
else [no element, but minterms did change]
update minterm geometries;
else if (boundary edge is STOP edge)
begin
    if (c_sr.current_element != c_sr.previous_element)
        update geometries of stopped element;
else
    update geometries of stopped minterm;
end

Figure 8: Algorithm Of The Network_element Update Routine
(insert_start_edge/insert_stop_edge)

Every time a state ruler field is processed completely and the
element and minterm instance descriptors are updated, the state
ruler field variables are updated as well.
5.2 Determination And Updating of Geometrical Parameters

For every element that is extracted some geometrical properties are determined. First, surface and perimeter are determined for all elements and element minterms that are recognized. These parameters can be used for capacitance and (very rough) resistance calculations. Second, a sorted list of edges that form the element outline are coupled with the corresponding EID. This sorted list only contains vertical edges, which indicate the left- and right boundaries of an element instance. No horizontal edges are linked within the sorted edge list, except for those horizontal edges that form the element nodes. This way of representing elements by edges, makes detailed resistance and transmission line calculations more easy. Algorithms which perform such actions are still in development. The way that nodes of element instances are determined will be discussed in the next section.

The information held within the EID is shown in figure 9. The EID contains entries to store its surface, perimeter, the added surface of its separate minterms, as well as a pointer to a list of edges. The other pointers are used to link the EID in its corresponding EID-list, and for merging purposes (see previous section).

![Diagram](image)

Figure 9: Element Instance Descriptor (EID) information.

Surface is calculated when an element stop edge is found in the current state ruler field. Now the difference between the current- and the previous-state ruler position can be computed and the result of this subtraction is multiplied with the state ruler field height. Finally, the computed surface is added to the EID's surface. For every separate minterm that is recognized within the element, the same procedure is applied.

The element perimeter is calculated by adding the height of the state ruler field to the EID's perimeter, every time an element start-, or stop edge is found. In case of a start/stop-edge, the EID's perimeter is further increased by subtracting the previous...
state ruler position of the lower/current state ruler field from the current state ruler position (in the current state ruler field) when current and lower field are connected and contain both the same element. The analogous actions are applied for the case that current and upper state ruler field are connected.

Capacitance can easily be estimated once surface, perimeter, and square-, respectively side wall-capacitance are known. Computation of the sum of the surfaces of minterm instances within the same element instance is not strictly necessary, but can be of practical use, when certain layout patterns are desired to belong to one element instance, while capacitances described by the element's minterms also have to be taken into account.

Resistance of long and narrow tracks can roughly be estimated by

\[ R = \frac{R_{\text{square}} \times \text{perimeter}^2}{(4 \times \text{surface})} \]

Calculation of errors (negligible when length/width > 10 for 4ummos processes), the check that the considered track is indeed long and narrow, and that nodes lay on either end of the track is not discussed here. Accurate resistance calculations can be done by using the boundary edges of the element instances, on which resistance computation algorithms are based that not have been completed yet (see section 1).

Element edges are represented by three coordinates: horizontal edges by its x-left, y-bottom, and x-right positions, vertical edges by its x-left, y-bottom, and y-top positions. Further on, vertical edges have additional type information to indicate that it is a start/stop edge of the element. The edges are sorted on their x-left and y-bottom position, for both horizontal and vertical edges. When horizontal and vertical edges have the same x-left, and y-bottom position, the horizontal edge comes first. This is shown in figure 10. The format chosen for edge records to represent vertical edges is equal to the format of the vertical edges by which the layout input data is represented in files (vin-files, see section 2). Vertical edges are made whenever an element starts or stops, and are linked within the element's edge list. However when the same element starts/stops in the current-, lower-, and/or upper-state ruler field at the same state ruler position, having the same nodes with other elements, only one vertical edge will be linked within the element's edge list.

5.3 Node Determination And Updating

All nodes that are found in a layout are located on certain vertical and/or horizontal edges. Therefore each edge contains an array of pointers to node descriptors, in case one or more nodes have been found on this edge. The size of such a node-pointer-array is equal to the number of interconnect layers that are specified by technology. The node descriptors contain the actual node information, such as the interconnect layer it is located in, its number/name and two pointers to the two EID in between the node is present. Furthermore, all the node descriptors are linked in a nodelist. This data structure is illustrated in figure 11. When the nodes present in certain layers have to be merged (as specified by the technology description) for various elements, we
Figure 10: Representation And Sorting Of Element Edges.

Figure 11: Element-Edge-Node- Data structure
use only a single edge record to link the EID with the node descriptor (via an array of node pointers). This is done because there is no need for detailed edge information in this case, and memory usage and run time are drastically reduced this way. Node merging for certain elements takes the effect that accurate resistance computations will be impossible.

The node determination and update routine searches for nodes in a nested tree of "if-then-else"-statements, which checks all boundaries between the current-, upper-, lower- and previous-state ruler field. Figure 12 shows the possible configurations of elements and nodes in and around the current state ruler field.

![Possible Node And Element Configurations](image)

Nodes are represented by "1111", "2222", "3333", "4444", "5555", "6666", and "7777". The elements present in a field by "e1", "e2", "e3", "e4", "e5", and "e6". It is possible that not all the indicated nodes exist, and/or some indicated elements are the same, which means that some of the indicated nodes are equal. Moreover, the nodes "6666" and "7777" are investigated, when the corresponding sr_field is processed, and are therefore irrelevant to the node determination routine.

The node update routine will start by checking if a vertical node is present ("1111" in figure 12). From this position the edges "2222", "4444", "3333", and "5555" are checked. Whenever elements "e1", and "e2" are present on either side of the edges in figure 12, these edges will contain nodes equal to those on the vertical edge, and are merged with each other. Other situations will cause that no merging takes place. Merging of nodes means, that multiple node descriptors are replaced by only one node descriptor to characterize the updated node. The node that is removed is unlinked from the node-list, and sets its parent pointer to the remaining node, so EID's having links with the unlinked node descriptors, still are connected with the updated node descriptors by means of "parent pointers". When "old" horizontal nodes are updated, ("2222", and "4444", in Fig.14), the third coordinate (x-right) in the edge_record is made equal to the current state ruler position, and the nodelength is recorded. These "old" edge records are accessed by means of two pointers to edges that are present in the state ruler field for respectively the upper-horizontal and down horizontal edge records. When new nodes ("3333", "5555") are found on horizontal edges, new horizontal
edges records are created and linked with the EID they are part of. Figure 13 visualizes the operation of the node determination routine.

```plaintext
if (no node descriptors have been made for vert. edge)
    make new node descriptors for edge vertical edge;
switch (node configurations)
begin
    [the edges "****", "xxxx", "2222", "3333", "4444", and "5555" may be edges at which no nodes are located.]

    (1) e2 e2 , (2) e1 e2 , (3) e1 e2 ,
    e2 e2 e1 e1 e2 e2

    (4) e1 e2 , (5) e1 e2 , (6) e1 e2 ,
    e1 e1 e2 e2 e2 e2

    (7) e1 e2 , (8) e1 e2 , (9) e1 e2
    **** xxxx 2222+3333
    e5 e6 e1 e1 4444+5555

    case (1 .. 9):
        if (edges "xxxx", "****" contain nodes)
            make new node descriptors for these edges;
            make new horizontal edge descriptors for edges at right hand side of vert. edge for instances of elements e1, and e2;
            merge nodes on the edges "-----";
            fill in third edge coordinate of horizontal edge records for edges at left hand side of vertical edge (linked with sr_fields);
            set pointers to horizontal up/down edge records for edges at left hand side of vert. edge to NULL in state ruler field;

Figure 13: The Node Update Routine

6. Implementation Results

In this section, the layout of a random counter will serve as an example to illustrate the capabilities and use of the extractor and simulation programs. Shown is in which way the hierarchy in a circuit layout is described by the layout data, and how this hierarchy is represented in the circuit description extracted from

2.68
layout. Exploitation of hierarchy will reduce the run time and the memory requirements of the extractor program.

Consider the functional circuit schematic of the random counter in figure 14. It consists of two parts: a 8-bit shift register, and a feedback circuit. The shift register on its turn consists of 8-latches and 8-selectors, while the feedback circuitry consists of three modulo-2 gates.

![Diagram of random counter]

**Figure 14(a):** functional schematic of random counter

![Diagram of shift register and feedback circuitry]

**Figure 14(b):** register level functional schematic of the feedback and shift register circuitry.

Figure 15 shows the model hierarchy present in the random counter.
To be able to express the hierarchy in a cell in the description of its layout, it is possible to reference cells that are described outside the present cell description. These cell references can be given a different name to distinguish different references to the same cell. The layout description of the random counter containing the cell references to the feedback cell and the shift register cell as well as the cell references to the latch-, select- and modulo2 cell is given in figure 16.
Figure 16: Hierarchical Layout description of Random Counter.

Figure 16 shows that the layout description of the random counter cell contains references to a feedback and a shift-register cell. Similarly the register cell contains references to a select cell, and the feedback cell contains references to a modulo2 cell. A plot of the layout of the random counter can be seen in plate 1, Chapter 2.2.

Part of the circuit description produced by the extractor when operating on this hierarchical layout is shown in figure 17, while the run times and memory requirements needed to extract each different cell to obtain the total circuit description are given in table 1. Notice that the same hierarchy can be spotted in the circuit description as in the layout description.
network rand_cnt(vss_13, vss_r3, vdd_13, vdd_r3, vdd_r2, phi2_r, phi1_r, vss_r1, run_r, p_l1d_r, vdd_r1, vdd_12, phi2_1, phi1_1, vss_11, run_1, p_l1d_1, vdd_11, out, in, fb_in)

{reg} sel_reg8 (vdd_r2, phi2_r, phi1_r, vss_r1, run_r, p_l1d_r, vdd_r1, vdd_12, phi2_1, phi1_1, vss_11, run_1, p_l1d_1, vdd_11, 24u, 21, 18, 15, 12, 9, 6, 3, out[0], out[1], out[2], out[3], out[4u], out[5], out[6], out[7], in[0], in[1], in[2], in[3], in[4u], in[5], in[6], in[7]);

{fb} feedback (vss_13, vss_r3, vdd_13, out[7], 3, out[5], 9, out[4u], 12, out[0], 24, fb_in);

network sel_reg8 (vdd_r2, phi2_r, phi1_r, vss_r1, run_r, p_l1d_r, vdd_r1, vdd_12, phi2_1, phi1_1, vss_11, run_1, p_l1d_1, vdd_11, out_i, out, in)

{Select[0 .. 7]} select {33, 29, 25, 21, 17, 13, 9, 5, 58, 57, 56, 55, 54, 53, 52, p_l1d_r, p_l1d_1, 58, 57, 56, 55, 54, 53, 52, 67, 66, 65, 64, 63, 62, 61, run_r, run_1, 67, 66, 65, 64, 63, 62, 61, 49, 48, 47, 46, 45, 44, 43, vdd_r1, vdd_11, 49, 48, 47, 46, 45, 44, 43, vdd_12, 84, 82, 80, 78, 76, 74, 72, 70, 83, 81, 79, 77, 75, 73, 71, vss_r1, vss_11, 83, 81, 79, 77, 75, 73, 71, in[0 .. 7], 41, 40, 39, 38, 37, 36, 35, 34);

{Latch[0 .. 7]} latch {111, 110, 109, 108, 107, 106, 105, vdd_r2, vdd_12, 111, 110, 109, 108, 107, 106, 105, 102, 101, 100, 99, 98, 97, 96, phi2_r, phi2_1, 102, 101, 100, 99, 98, 97, 96, 93, 92, 91, 90, 89, 88, 87, phi1_r, phi1_1, 93, 92, 91, 90, 89, 88, 87, 84, 82, 80, 78, 76, 74, 72, 70, 29, 25, 21, 17, 13, 9, 5, 1, out_i[0 .. 7], out[0 .. 7], 41, 40, 39, 38, 37, 36, 35, 34};

network feedback(vss_l, vss_r, vdd_l, vdd_r, a8, n_a8, a6, n_a6, a5, n_a5, a1, n_a1, fb)

{fb_1} mod2_fb (17, vss_l, 21, vdd_l, 14, fb, 12, 13, a1, n_a1);

{fb_5} mod2_fb (16, 17, 20, 21, 12, 13, 10, 11, a5, n_a5);

{fb_6} mod2_fb (vss_r, 16, vdd_r, 20, 10, 11, n_a8, a8, a6, n_a6);

2.72
```plaintext	network select (in_2, nsc_r, nsc_1, sc_r, sc_1, vdd_r, vdd_1, vssd, vss_r, vss_1, in_1, out)
{
  net {nsc_r, nsc_1};
  net {sc_r, sc_1};
  net {vdd_r, vdd_1};
  net {vssd, vss_1, vss_r};
  nenh w= 6  l= 4u (in_1, out, 3);
  nenh w= 6  l= 4u (sc_r, out, 4);
  nenh w= 6  l= 4u (nsc_r, 3, vssd);
  nenh w= 6  l= 4u (in_2, 4, vssd);
  ndep w= 6  l= 20 (out, vdd_r, out);
  cap 4u.08f (vssd, out);
  cap 3.90f (vdd_r, out);
  cap 4u.56f (vssd, out);
  cap 3.60f (vdd_r, vssd);
  cap 4u5.22f (out, gnd);
  cap 2.4u0f (in_1, gnd);
  cap 2.16f (2, gnd);
  cap 2.70f (4u, gnd);
  cap 16.33f (vdd_r, gnd);
  cap 14u.58f (nsc_r, gnd);
  cap 14u.76f (sc_r, gnd);
  cap 4u.56f (in_2, gnd);
  cap 32.54uf (vssd, gnd);
}

network latch (vdd_r, vdd_1, phi2_r, phi2_1, phi1_r, phi1_1, vss, shiftout, out_inv, out, in)
{
  net {vdd_r, vdd_1};
  net {phi2_r, phi2_1};
  net {phi1_r, phi1_1};
  net {shiftout, out};
  nenh w= 8u  l= 4u (3, vss, 7);
  nenh w= 8u  l= 4u (vss, shiftout, 4);
  nenh w= 8u  l= 4u (shiftout, vss, out_inv);
  nenh w= 8u  l= 4u (2, vss, 3);
  nenh w= 8u  l= 4u (phi1_r, in, 2);
  nenh w= 8u  l= 4u (phi2_r, 2, 7);
  nenh w= 8u  l= 4u (phi2_r, 3, 4);
  ndep w= 6u  l= 18u (shiftout, shiftout, vdd_r);
  ndep w= 6u  l= 17u (3, vdd_r 3);
  ndep w= 6u  l= 19u (out_inv, out_inv, vdd_r);
  ndep w= 6u  l= 17u (7, 7, vdd_r);
  cap 2.64f (4, 7);
  cap 2.16f (phi2_r, vss);
  cap 2.58f (vdd_r, vss);
  cap 3.54f (phi1_r, vss);
  cap 2.88f (phi1_r, in);
  cap 2.40f (phi2_r, 2);
  cap 2.64f (phi2_r, 3);
  cap 4.62f (phi1_r, vdd_r);
  cap 2.16f (vdd_r, shiftout);
  cap 54.44f (vss, gnd);
  cap 17.44f (2, gnd);
  cap 29.84f (3, gnd);
  cap 2.73
```
7. Conclusions and Suggestions

There are a number of conclusions and remarks we can make after having developed and implemented the algorithms discussed above. The most important are:

<table>
<thead>
<tr>
<th>circuit</th>
<th>elements</th>
<th>CPU-time</th>
<th>CPU-memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>latch</td>
<td>114</td>
<td>6.1 (s)</td>
<td>60 Kbytes</td>
</tr>
<tr>
<td>select</td>
<td>83</td>
<td>4.9 (s)</td>
<td>50 Kbytes</td>
</tr>
<tr>
<td>mod2_fb</td>
<td>76</td>
<td>4.7 (s)</td>
<td>50 Kbytes</td>
</tr>
<tr>
<td>feedback</td>
<td>241</td>
<td>5.6 (s)</td>
<td>60 Kbytes</td>
</tr>
<tr>
<td>shift_reg</td>
<td>350</td>
<td>7.8 (s)</td>
<td>80 Kbytes</td>
</tr>
<tr>
<td>random_counter</td>
<td>236</td>
<td>9.5 (s)</td>
<td>60 Kbytes</td>
</tr>
</tbody>
</table>

Table 1: Run time and memory requirements
1. The cells need to be extracted only once. That way we exploit the hierarchy and repetition present in the layout description of a cell.

2. The hierarchy in the network description is the same as the one in the layout description. This facilitates verification and interpretation of results obtained.

3. The algorithms developed are linear with respect to the number of elements extracted.

4. The characterization of elements by EC's enables us to extract any kind of electrical function in any technology, as long as it is based on the presence of a particular combination of masks.

5. By considering element boundaries, nodes are created implicitly whenever two boundary edges coincide.

6. Output can be made for any circuit simulator program like SPICE and the Delft Switch Level Simulator SLS [5], as well accurate resistance and transmission line calculation programs.

Algorithms that perform very detailed resistance, capacitance, and transmission line calculations, that make use of the sorted edge-lists as described in section 5 are in development and will in the near future be coupled with the extractor programs. These programs will especially be used to study the transmission line behavior of very long and narrow tracks in layout, that form the interconnect (routing) pattern between modules on a chip. Elaborate resistance calculations will not be performed on short tracks in small cells, since in this case resistance is small compared to the resistance of transistors in the same cells and should be neglected to do efficient extraction.

References


Next we will give a very simple example to illustrate the extraction algorithms. Consider the NMOS layout given in figure A1.

The letters (A-I) indicate vertical-, and node-edges.

Figure A1: Layout of NMOS example circuit.

This layout description is first transformed in a set of sorted line segment files. From these edges the routine extract builds the state ruler profiles. The first profile will contain two fields. When these fields are scanned by the routine recognize_elements they will be recognized as a metal_interconnect start edge, and a poly_interconnect start edge. This will result in the creation of two element instance descriptors. Since the connect type of both edge descriptors is NULL_CONNECT the update routine will create two new element instance descriptors (EID). Further on, for each element instance a vertical edge descriptor is created and linked with the EID's list of edges. No nodes are found on horizontal edges, so no horizontal edge descriptor will be created. After the first scan the network data structure looks like figure A2. The second profile will result from the first after the insertion of the diffusion start_edge. The profile will contain four fields, and the routine recognize_elements will recognize five boundary edges. These are: a metal_interconnect stop edge and a metal_diffusion capacitor start edge in the first field, diffusion_interconnect start edge in the second and fourth field, and a poly_interconnect stop edge and an enhancement transistor start edge in the third field. The network structure
Figure A2: Network structure after first scan.

after the second scan is given in figure A3,
The node determination routine will also create nodes at the edges (D), (E), (F), (G), and (H) (see Fig.A1), and are linked with the edges descriptors of the recognized elements which in turn are linked with their element instance's edge list. The edges (D), (F), and (G) form diffusion-nodes, the edge (E) forms a poly-node, while the edge (H) locates a metal-node. The edge- and node-data structure of the diffusion interconnect EID in the fourth state ruler field, and the metal interconnect EID in the first field after the second scan is given in Fig.A4. Note that horizontal edge descriptors are only created and linked with their EID-s in case nodes are present at these edges. Vertical edge descriptors are always created and linked with the EID's edge list.

Figure A3 : Network structure after second scan.
The third profile results from the insertion of the contact start edge. This results in the creation of a diffusion metal contact. The fourth, fifth etc. profiles are made and analysed in a likewise manner, resulting in the extraction of two polysilicon interconnections, two diffusion interconnections, a metal interconnection, a diffusion metal contact, and an enhancement transistor. When the entire layout is extracted, the last step is to transform the extracted description to a format suitable for verification, simulation etc.
Appendix B - Technology Description Of A NMOS Process (DELIETF Univ. of Techn.)

# definition of layers for all CAD tools #
layers:

- t_nd: hollow white;  # terminal layer for diff. layer
- t_nm: hollow white;  # terminal layer for metal layer
- t_np: hollow white;  # terminal layer for poly layer
- nm: solid blue term layer = t_nm;  # interconnect layer metal
- np: solid red term layer = t_np;  # interconnect layer poly
- nd: solid green term layer = t_nd;  # interconnect layer diffusion
- nc: stipple black;  # contact hole
- nx: stipple white;  # crossing
- ni: stipple yellow;  # implant for depl. transistor
- nb: stipple green;  # buried contact for connecting

# definition of elements for extraction and simulation purposes #
elements:

- transistor: metal_trans = (np) nd (nm) !ni !nx !nb !nc !t_nd,
  norm_trans = (np) nd !nm !ni !nb !nx !nc !t_nd;
- dep_trans = ni (np) nd !nm !nb !nx !nc !t_nd !t_nm,
  norm_depl = ni (np) nd !nm !nb !nx !nc !t_nd !t_nm;
- normal_poly = (np) !nd !nm !nc !t_nd !t_nm;
- diff_ic:
  norm_diff = (nd) !np !nm !nc !t_np !t_nm;
  met_ic:
  norm_met = (nm) !np !nd !nc !t_np !t_nd;
- crospoly = nx (np) (nd) !nm !nc !t_nm;
metpolcap:
mpcap = (nm) (np) !nd !nc !t_nd;

metdiffcap:
mdcap = (nm) (nd) !np !nc !t_np;

mepoldiffcap:
mpdcap • (nm) (np) nd nx !nc;

# contacts between different interconnect layers
difmcnt:
dmc = nc (nm nd) !np !t_np;
polmcnt:
pmtc = nc (nm np) !nd !t_nd;

burcnt:
bc = nb (nd np) !nx !nc !nm !t_nm;
methburcnt:
mbc = nb (nd np) !nx !nc nm;

threecnt:
trlc = nb nc (nm np nd);

# layer combinations that are to be ignored by the extractor
ignore:
ign1 = ni !np !nd !nc !nm !t_np !t_nd !t_nm,
ign2 = nx !np !nd !nc !nb !nm !ni !t_np !t_nd !t_nm,
ign3 = nb !np !nd !nc !nx !nm !ni !t_np !t_nd !t_nm;

# specification of the types of elements with some process
# parameters to be used by the extractor's output functions
# making switch-level/circuit description for sls[2] and SPICE
element_types:

<table>
<thead>
<tr>
<th>type</th>
<th>name</th>
<th>gate-layer</th>
<th>drain/source-layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>nenh:</td>
<td>transistor</td>
<td>np</td>
<td>nd</td>
</tr>
<tr>
<td>depl:</td>
<td>depl_trans</td>
<td>np</td>
<td>nd</td>
</tr>
</tbody>
</table>

# a empty string instead of a layer name means the bulk
# type name lay1 lay2 sqcap wallcap

| cap:     | metal_trans     | nm np      | .03     | 0     |
| cap:     | depl_trans      | nm np      | .03     | 0     |
| cap:     | diff_ic         | nd         | .04     | .15   |
| cap:     | met_ic          | nm         | .03     | 0     |
| cap:     | poly_ic         | np         | .06     | 0     |
| cap:     | poldiffcap      | np nd      | .06     | 0     |
| cap:     | poldiffcap      | nd         | .04     | .15   |
| cap:     | metpolcap       | nm np      | .03     | 0     |
| cap:     | metpolcap       | np         | .06     | 0     |
| cap:     | metdiffcap      | nm nd      | .06     | 0     |
| cap:     | metdiffcap      | nd         | .04     | .15   |

2.81
<table>
<thead>
<tr>
<th>cap</th>
<th>metpoldiffcap</th>
<th>nm</th>
<th>np</th>
<th>.03</th>
<th>0,</th>
</tr>
</thead>
<tbody>
<tr>
<td>cap</td>
<td>metpoldiffcap</td>
<td>np</td>
<td>nd</td>
<td>.06</td>
<td>0,</td>
</tr>
<tr>
<td>cap</td>
<td>metpoldiffcap</td>
<td>nd</td>
<td></td>
<td>.04</td>
<td>.15,</td>
</tr>
<tr>
<td>cap</td>
<td>difmtcnt</td>
<td>nd</td>
<td></td>
<td>.04</td>
<td>.15,</td>
</tr>
<tr>
<td>cap</td>
<td>polmtcnt</td>
<td>np</td>
<td></td>
<td>.06</td>
<td>0,</td>
</tr>
<tr>
<td>cap</td>
<td>burcnt</td>
<td>nd</td>
<td></td>
<td>.04</td>
<td>.15,</td>
</tr>
<tr>
<td>cap</td>
<td>metburcnt</td>
<td>nm</td>
<td>nd</td>
<td>.06</td>
<td>0,</td>
</tr>
<tr>
<td>cap</td>
<td>metburcnt</td>
<td>nd</td>
<td></td>
<td>.04</td>
<td>.15,</td>
</tr>
</tbody>
</table>

2.82
1. Introduction

This manual is meant as a guide to users who want to extract a network description from a circuit layout with the ext layout to circuit extractor. The user is allowed to specify the technology in which a circuit layout is described. The description of a technology concerning the extractor program falls out in three parts:

1. the specification of the layers used within the technology,
2. the specification of elements that cover all the possible layer combinations in a layout each having a different electrical behavior,
3. specification of the types of the specified elements for the extractor's output functions, that make network descriptions for sls [5] and SPICE simulators.

The extractor's output functions are fit to produce switch-level and SPICE descriptions from a circuit layout including the length and width of transistors, and the present capacitors. With these functions, switch-level-descriptions in any technology can be made, as long as the user gives the types of the specified elements and the process parameters (the square and side wall capacitance of capacitors) involved with them. Functions capable of accurate resistance and capacitance calculations are in development (P. Kazil, Dept. of Electrical Engineering, Univ. of Techn. Delft).

When desired the extractor will make a detailed extraction of a circuit layout including all boundary vertical element edges, and edges locating nodes, to allow the accurate determination of the electrical network for special layout patterns. For this to happen, the user must not specify layers in which nodes have to be merged for certain elements. The network information will be written into a output file which can be used as input for further processing tools. The operation of the extractor without node merging however will be time and memory costing. It is strongly suggested to use the extractor in this mode only for certain layout patterns. This is largely controllable by selectively extracting the cells with critical paths, and specifying a few number of elements (layer combinations with a certain electrical behavior) for extraction and no-node merging. This mode is automatically selected for each element that doesn't have node-merge-statements (node names between parenthesis).

Section 2 describes the ext-technology description language, and gives some examples of possible technology descriptions. Section 3 contains a table with the lexical items as tokens and delimiters. Section 4 contains a table with the total syntax description of the ext-technology description. The last section is gives
2. The Ext-Technology Description Language

2.1 Conventions Of Syntax Notation

It is a common practice to describe the syntax definition of a computer language in some meta language. The syntax definition of the ext-technology description is given in the meta language proposed by Wirth[12]. In this language the terminal symbols (tokens) are denoted by words between double quotes, by underscored words, or words in italic form, and non-terminal symbols by words in lower case. Alternatives are separated by a vertical bar, i.e. alb means a or b. Repetition is denoted by curly brackets, i.e. {a} stands for empty|a|a|a|a|a|a|a|a... Optionality is expressed by square brackets, i.e. [ab] stands for empty|a|b. Parenthesis serve for grouping, e.g. (ab)c means aclbc. Literals are enclosed in quote marks. Each production begins with a non-terminal symbol followed by an equality-sign and a sequence of terminals and non-terminals and terminated by a period.

2.2 General Structure of Technology Description

As stated in the introduction, the technology description concerning the extractor consists of four parts. This is illustrated in table 1:

| technology_descr = layer_descr element_descr element_types |

Table 1: Syntax of the technology description.

2.3 Layer Description

A layer is described by a name, for graphical purposes by a color and fill type, and in case it concerns an interconnect layer it's related terminal layer. This is to indicate that the layer can locate nodes and may form the connections between different cells. Moreover hierarchical extraction is impossible when interconnect layer don't have a related terminal layer, and different cells are connected with each other by this layers. The syntax is given in table 2.

| layer_descr = layers layer_spec {layer_spec}. |
| layer_spec = name "." color fill [term_layer]. |
| term_layer = name. |

Table 2: Syntax of the layer description.
Each layer-name and terminal-layer-name have to be unique. Different layers may not reference the same terminal layer and terminal layers have to be defined before they may be referenced. All these conditions are checked when the extractor is invoked.

### 2.4 Element Description

The element description must state which layer combinations that may occur somewhere in layout are relevant, and which are not. An element may consist of multiple layer combinations (minterms) that have to be threatened in the same way whenever the extractor encounters them in layout. Each element as well all its layer combinations must have unique names. When it is desired to specify some layer combinations that don't have to be extracted, the user can specify them by the name `ignore`. Further on, layer references that are parenthesized indicate that all nodes found in these layers must be merged for this minterm. The syntax of the element description is given in table 3.

```
elements = elements element {element}.
element = (name "":" minterms | ignore ":" minterms) ";".
minterms = minterm ("," minterm).
minterm = name "=" {minterm_layers}.
minterm_layers = layer.refs | ("layer.refs ")".
layer.refs = layer_ref {layer.refs}.
layer_ref = name | "!" name {name | "!" name}.
```

*table 3: syntax of the layer description.*

The layer combinations of different minterms may not be equal, nor overlap. Overlapping of minterms may occur in case not all existing layers have been referenced in one or more minterms. The above conditions will be checked when invoking the extractor.

### 2.5 Specification Of Element Types For The Extractor's Output

At present the extractor's output functions are able to produce input for switch-level-, and circuit simulators. The network description contains transistors and capacitors, but without resistors, which will not be of considerable importance within reasonably well designed small cells. Like stated in the previous sections, resistance and transmission line computing algorithms are in development. Specification of the types of elements is therefore restricted to transistors and capacitors. Moreover, at present only Nmos and Cmos transistors are accepted as input by the output functions, but the specification of other types of transistors will not cause the abortion of the layout extraction and the network data structure will be outputted in the output file `model_out`. The only process parameters to be specified are the square-, and the side wall-capacitance, because transistor models are for the current Nmos and Cmos processes already included in the simulator programs. The description of the element type specifications is given in table 4.
element_types = ( (transistor | capacitor) )
    ( ";" (transistor | capacitor)) ";".
transistor = [menh | dept | penh] ";" element_name layer_ref layer_ref.
capacitor = cap e ":" element_name
    layer_ref [layer_ref]
    float float.
layer_ref = name.
element_name = name.

Table 4: Syntax of the element type specification.

For transistors as well as capacitors only two layers have to be
specified: that is the gate and drain/source layers for transis­
tors, and the two layers by which the capacitor is formed. When
one of the two layers is not specified, the substrate is meant,
which is only allowed for capacitors. The element type descrip­
tion consists all thus of zero or multiple capacitor and transis­
tor statements, with transistors and capacitors only having two
layer references, and capacitors also having a square and a side
wall capacitance specification. The side wall capacitance specif­
ication is valid for the first layer reference. The extractor
will check if the layers referenced in the element types are also
referred in the element/minterm specification.

2.6 Examples

An example of an technology description in nmos is given in appen­
dix B of this report. Suppose the user only wants to know how the
transistors of a circuit are connected, with each other without
separately retaining capacitor/resistor data. Now all layer com­
binations that don’t specify transistors should be appointed to
one element. Figure C1 shows the element description in this
case. The layer description is the same as the one given in
appendix B.
elements:

ignore:
  dc1 = ni !np !nd !nc !nm !t(np) !t(nd) !t_nm,
dc2 = nx !np !nd !nc !nb !nm !ni !t(np) !t(nd) !t_nm,
dc3 = nb !np !nd !nc !nx !nm !ni !t(np) !t(nd) !t_nm;

enhancement:
  normal_enh = (np) nd !nx !nb !ni !nm !t_nm !t_nd,
  metal_enh = (np) nd !nx !nb !ni (nm) !t_nd;

depletion:
  normal_depl = (np) nd !nx !nb ni !nm !t_nd !t_nm,
  metal_depl = (np) nd !nx !nb ni (nm) !t_nd;

otherelms:
  diff_track = !np (nd) !nm !nc !t(np) !t_nm,
  diff_metal = !np (nd) (nm) !nc !t(np),
  poly_track = !nd (np) !nm !nc !t(nd) !t_nm,
  poly_metal = !nd (np) (nm) !nc !t nd,
  metal_track = !np !nd (nm) !t_np !t Nd,
  crossing = (np) (nd) nx !nb !nm !t_nm,
  cross_metal = (np) (nd) nx !nb (nm),
  buried = (np nd) nb !nx !nm !t_nm,
  bur_metal = (np nd) nb !nx !nc (nm),
  polmet_cont = (np nm) nc !nd !t_nd,
  poldifmet_cont = (np nm nd) nc nb !nx,
  difmet_cont = (nd nm) nc !np !t_np;

element_types:
  nenh: enhancement np nd,
  depl: depletion np nd;

Figure C1: Element Description For Simple Switch Level Extraction.

this way all relevant interconnecting layout patterns are con­
sidered to belong to one element, except the transistors. Every
different layer combination with a certain node-merge condition
(parenthesized layer names) is stated by a minterm, so that the
extractor knows which nodes it has to merge every time it
encounters another minterm. The network description is now
obtained by simply outputting the extracted transistor instances
with their nodes. Figure C1 also shows the element type specifi­
cation in this case.

Suppose the user wants a detailed extraction of poly-interconnect
layers without transistors or capacitors to calculate the resis­
tance of poly-interconnect patterns. The element description
would be like:
ignore:
next = ! np;

poly_track:
ptr = np;

element_types:
    #: # empty

which will extract all the layout patterns with poly to calculate the critical poly paths in a chip layout. Moreover nodes are not merged, and all boundary vertical edges and horizontal edges containing nodes are extracted and outputted in a the file model_out in case the -x option is specified when invoking the extractor. the extractor (see extract manual page in appendix C).

3. Lexical Conventions

| layers    | "layers".                                      |
| color     | "red" | "green" | "blue" | "yellow" |
|           | "orange" | "cyan" | "white" | "black" |
|           | "magenta" | "violet". |
| fill      | "hollow" | "solid" | "stipple". |
| elements  | "elements".                                     |
| ignore    | "ignore".                                        |
| element_types | "element_types".  |
| nenh      | "nenh".                                          |
| penh      | "penh".                                           |
| depl      | "depl".                                           |
| cap       | "cap".                                            |
| name      | letter { letter | digit } .                               |
|           | integer [exponent] .                               |
| integer   | digit (digit) .                                    |
| exponent  | ( "e" | "E" ) [ ( "-" | "+" ) integer ] .               |
| letter    | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
|           | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" |
|           | "w" | "x" | "y" | "z" .                                    |
|           | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
|           | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" |
|           | "W" | "X" | "Y" | "Z" .                                    |
| digit     | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" . |

Table 5. Lexical Conventions of the reserved keywords.

A comment starts on a line with "#" and ends at the end of the line.
4. Syntax Description

```
technology_descr = layer_descr element_descr element_types
layer_descr = layers layer_spec (layer_spec).
layer_spec = name ":" color fill [term_layer].
term_layer = name.
elements = elements element {element}.
element = [name ":" minterms
    | ignore ":" minterms] ";" .
minterms = minterm ("," minterm)
minterm = name "=" (minterm_layers)
minterm_layers = layer_refs | (" layer_refs ")
layer.refs = layer_ref {layer_ref}
layer_ref = name | "!" name (name | "!" name)
element_types = { (transistor | capacitor) }
    | (" | (transistor | capacitor)) ";" .
transistor = [menh | depl | menh] ";"
    element_name layer_ref layer_ref.
capacitor = cap ":" element_name
    layer_ref [layer_ref]
    float float.
layer_ref = name.
element_name = name.
```

Table 6. Total Syntax Description.
NAME

ext - layout to circuit extractor program

SYNOPSIS

ext (-cimrtx -pprocess -nname)

DESCRIPTION

Ext is a layout to circuit extractor program, for orthogonal layouts in various kinds of technology. When invoked normally, it will make a switch level network description with capacitors from the layout models named in the "exp_dat" file for the sls(1ICD)-simulator, and a circuit description containing transistors and capacitors for SPICE. The current working directory must be the project directory.

An option can be specified as follows:

-x When this option is given, the extractor will output its entire data structure of extracted elements in a file: their surface, perimeter, edges, coordinates, and nodes. In case no process is specified, the standard DELFT nmos process is taken.

-pprocess When this option is given, the user can specify a file in which a technology description resides.

-r When this option is given, the extraction is traced and every recognized minterm is printed.

-nname When this option is given, the name of the cell (model) that has to be extracted can be specified, but the conversion of this model must have succeeded.

-t Print extra information for transistors, like its left bottom coordinate, surface and perimeter.

-c Print extra information for capacitors, like its left bottom coordinate, surface, perimeter and node length.

-1 When this option is given, the information that the extractor obtains from the technology file is printed. This information is about:

layers: the fill, color, bit code and accessory terminal if so are printed.

element: the layers in which node merging takes place, including cross merging (merging of nodes in different interconnect layers) are printed, as well as, the layers that are part of the element definition, and the minterms belonging to this element.
minterm: every legal layer combination, with the name of minterm and element involved with it is printed.

element types: the name and type, of transistors with their gate and drain/source-layer, as well as the type and name of capacitors with their square and side wall capacitance for every layer that is part of the element specification is printed.

−m print memory allocation statistics: the memory that is in use by data types of different sizes as well as the free list are printed.

The invocation of ext needs the presence of an technology file whenever the p-option is specified.
Library process descriptions can be automatically accessed by setting the c-shell environment variable "PROCESS" to the name of the library process. E.g., the unix c-shell command setenv PROCESS cmos will set the current process to the library technology description cmos.

AUTHORS
A.A.J de Lange, J. Annevelink

SEE ALSO
A.A.J. de Lange, "Extract Users Manual" (Internal Report, Lab. for Network Theory, Dept. of Electr. Eng., Delft University of Technology), cldm(1ICD), convert(1ICD), sls(1ICD)

DIAGNOSTICS
Errors in the syntax description of the specified technology file as well as the faults occurring during extraction are reported. The diagnostics are meant to be self explanatory.

LIMITATIONS
In case full extraction is desired of large circuit layouts, the extractor may be unable to get enough core from the operating system. Extraction should now be done with node merging specified for each element as much as possible (see users manual). In this mode, not all edges and nodes retained for each element.

BUGS
Please report all bugs by electronic mail to "fons@dutentb.uucp"

2.91
FILES

proj_dir/model/layout/LA_vln  (input files) LA=LAyer
proj_dir/model/layout/exp_tid  (input file) terminal id's
proj_dir/model/layout/cnv_teq  (input file) terminal equiv.
proj_dir/model/circuit/model_sls (output file for SLS)
proj_dir/model/circuit/model_spc (output file for SPICE)
proj_dir/model/circuit/model_out (output file geometric
  and node network description)
  -cacdsrc/lib/process/nmos (DELFT nmos technology
    description file)
  -cacdsrc/lib/process/anmos (DELFT nmos technology descr.
    file without node merging)
  -cacdsrc/lib/process/cmos (MIETEC cmos technology descr.
    file with node merging)
A switch level simulator for digital MOS circuits which is capable of timing simulation is described. Because it is local-event-driven and performs simulation based on actual transistor and interconnection parameters, the simulator can be used as a tool for quickly verifying large circuits on their logic and timing behavior. A novel definition of "the state of a node" allows the simulator to approximate node voltages with analogue continuous voltage waveforms.

Summary

The SLS simulator is a simulator which falls somewhere between an ordinary logic simulator and a circuit simulator. It has advantages of both sides. The similarity with an ordinary logic simulator is that SLS runs with about the same speed, and that signal input and output descriptions are O's, I's and X's. The similarity with a circuit simulator is that the SLS simulator operates upon a network description containing transistors, resistors and capacitors, and internally deals with them in a way more or less similar to the way a circuit simulator does. That is, the SLS simulator determines logic states and delays based on actual circuit parameters and uses analog voltage waveforms internally. Although the result is less accurate as compared with a circuit simulator, the characteristics of most MOS digital circuits are such that it is possible to have a satisfying timing accuracy though.

The advantages of SLS can be summerized by: Easy, direct and fast verification of a circuit for its logic and timing behavior. Unlike gate-level logic simulators, there is no necessity of using gates in the network description and specifying delays by the designer, but all simulations can be done directly from a electrical network description, for example as produced by a circuit extractor.

The contents of this paper is as follows: The next section describes the theory of the switch level timing simulator. It is extracted from [1] (see also [2]). After the theory section the users manual for the programs involved with switch level simulation [3] can be found. Finally an example of simulation of an 8 bit random counter proceeds the reference list at the end of this paper.

This paper was produced as part of Task III of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
1. Introduction

Simulation of MOS digital circuits can be done at different levels. At the lowest level we have the circuit simulators (e.g. SPICE [4]). They are accurate but, because of long simulation times, simulation is limited to circuits with a few hundreds of transistors. As a result, these simulators are not suited for checking the behavior of large circuits. At a higher level we have the gate level logic simulation. This kind of simulation is much faster and can handle large networks, but it is not well adapted to the possibilities of MOS circuits, (e.g. it is not possible to represent the bidirectionality of a MOS pass transistor directly at the gate level). Moreover, the timing of the network can only be simulated with assignable delay. A hierarchical use of circuit and gate level simulators is often unsatisfactory because interactions at the circuit level are neglected.

The switch level model as proposed by Bryant [5][6] is able to represent MOS circuits directly in a logic simulator. Many simulators have been built according to this model, and their speed is close to the speed of the gate level simulators. The switch level model of Bryant characterizes the circuit elements (nodes and transistors) in classes of strength. The state which is assigned to a node is derived from the state of the class with the largest strength, reaching the node. To allow for this a classification of circuit elements is required, which can not always be done in a straightforward way. For example, when charge sharing between different nodes occurs, the user has to classify the node capacitances in "big" and "small" ones. Another disadvantage is that timing information is not available in the original switch level model of Bryant.

It is a well accepted practice to estimate delays in MOS circuits by modeling the circuit by RC networks which are driven by step voltages. Penfield and Rubinstein [12] proposed a method to bound the waveforms of nodes in RC tree networks, and Wyatt [13] showed how this method can be extended to more general RC meshes. Most switch or gate level simulators which are able to simulate delay, use this method to estimate delays [9][10][11]. But all of these simulators are purely logic simulators. This means that they have only logic node states like I or O. They are not able to deal in a reasonable way with multiple input changes within a short time at the same circuit stage (due to races for example).

The switch level simulator which is presented here is an extension of the simulator of Bryant in several ways. First, the circuit elements are modeled with analogue values instead of strength abstractions. In this way logic states are determined directly from actual circuit parameters and no classification of circuit elements is required. Next, the simulator is able to simulate the timing of the circuit. Not only because analogue circuit models are used, but also because the simulator uses analogue continuous voltage waveforms, the simulator has got some characteristics which are similar to those of circuit simulators. For example, because of the continuous voltage waveforms, simulation is able to deal in a reasonable way with multiple input changes within a
The extensions are backward compatible extensions which have no important drawback on the efficiency of the switch level principle. With the new simulation models and algorithms it is also possible to have the same simulation results as without timing, or with abstract circuit models, and simulation times remain comparable to those of gate level logic simulators.

The basic ideas of switch level timing simulation are the following:

- In order to maintain the switch level principle of event-driven vicinity evaluations, while at the same time allowing for timing simulation with analogue voltage waveforms, we introduce elementary functions. An elementary function is a function of time which is characterized from a class of functions by means of a set of parameters. By calculating the parameters of an elementary function during a vicinity evaluation, we predict the node voltage waveforms for a node for the time that its vicinity is not disturbed.

- The parameters of an elementary function are subdivided in those which describe the stable part of the waveform, and those which describe the dynamical part of it. In this way calculation clarity and flexibility results.

- To determine the voltage waveforms, a transistor which is "on", is modeled by a resistance between its drain and source, and each node has a capacitance to ground.

- The stable part of the node voltage waveforms is determined by resistance division, which accounts for the influence of nodes with a forced state, and charge sharing, which accounts for the influence of initial node charges. The dynamical part of the waveforms is estimated by using RC constants as described in [12][13][14].

- By using separate minimum voltage and maximum voltage waveforms, the X state is incorporated. Evaluations are done in such a way that worst and best calculations result in minimum voltage and maximum voltage waveforms. A min-max delay simulation allowing proportional deviations - which may be very valuable to detect time critical paths in a circuit - can also be performed in this way.

- By means of a logic switching threshold voltage, logic states are derived from the analogue voltage waveforms. In this way the relation is defined between the node state and the state of the transistors which are connected to it by their gate.

- A minimum stable voltage for the logic 1 state and a maximum stable voltage for the logic 0 state provides for safety margins when the analogue voltages are converted to logic states. When limits are chosen for these voltages, the simulator will reduce to a simulator which uses abstract circuit models in this way.
Although a switch level timing simulator will not be as accurate as a circuit simulator, it will be able to check large parts of layouts on their logic and timing behavior after a circuit extraction. It can be used as a tool for discovering logic faults because of wrong logic or improperly used transistors, and it can indicate timing problems or estimate speed performance. When transistors are used in a style different from usual design style, or when fast transient situations such as spikes occur, only a degeneration in accuracy will result.

2. The principle of the simulation

2.1 Models, signals and vicinities

We consider the network as consisting of nodes and transistors. Each node may have a capacitance to ground, and resistors are regarded as transistors which are always in the conducting state. The nodes of the network are distinguished between Forced nodes and Normal nodes. Forced nodes are the nodes which have a state which is forced in a one-directional way. For example, network input nodes or functional block output nodes are Forced nodes. The Normal nodes are the other nodes of the network. Each node has a minimum \( \text{uminit}(t) \) and maximum \( \text{umax}(t) \) voltage, and a logic state \( \text{LSTATE} \) : 1, 0 or \( X \), which is derived from these voltages. The logic state \( X \) denotes a logic state which is either 0 or 1.

A transistor is connected to nodes with its gate, source and drain. It is modeled by a capacitance to its gate, drain and source, and by a resistor and switch in series between its source and drain.

![Transistor model](image)

The state of a transistor is Open, Closed or Undefined, representing an open, closed or undefined state of the switch in the model. Besides the resistor we have three transistor types: Nenhancement, Penhancement and Depletion, and the state of them is determined by their gate in the following way.
The resistance and capacitance values which are used in the transistor model, depend on the type, dimensions and mode of the transistor. The mode of a transistor indicates the function of the transistor in the network. Modes of a transistor for example are the static mode, the pulldown mode, and the load mode.

A property of Forced nodes is that they block the influence from one Normal node to another Normal node. This, in addition to the switches in the transistor model, determines the vicinity of a node and thus allows for calculations restricted to the vicinity of it. A vicinity is defined as a set of (Normal) nodes which have mutual influence on each other, added with the (Forced) nodes which have influence on this set. For a Normal node n the definition of its vicinity is:

definition

The vicinity of a Normal node n is the set of nodes, including n, for which there is at least one path to n through Undefined or Closed transistors, with no intermediate Forced nodes in the path.

It can easily be seen that when a Normal node m is in the vicinity of node n, node n is also in the vicinity of node m. In fig. 2.2 an example of a vicinity is given.

fig. 2.2: Example of a vicinity (dashed line). O = Open, C = Closed and U = Undefined.
2.2 The state description of a node

In the switch level timing simulator computation is done for a node only when its vicinity is disturbed due to the changing of the state of transistors or Forced nodes. In this way full local-event-driven simulation is achieved, which - in addition with the more simple transistor model - provides a simulator which is several orders faster than a circuit simulator. In order to work with event-driven simulation and continuous waveforms simultaneously, we have to extend the state of the network. Therefore we introduce an elementary function e(t,p) for each node, which represents the node voltage u(t) during the time interval that the vicinity of the node is not disturbed. In e(t,p), p is a set of parameters to approximate the real u(t) during that interval, and these parameters constitute the state of each node in the network during simulation.

Hence a node voltage function u(t) for \( t_0 \leq t < t_n \) is described by

\[
    u(t) = e(t,p_1) c(t_0,t_1) + e(t,p_2) c(t_1,t_2) + \ldots + e(t,p_n) c(t,(n-1),t_n)
\]

where \( c(t_a,t_b) = 0 \) if \( t < t_a \)

\[
    = 1 \quad \text{if} \quad t_a \leq t < t_b
\]

\[
    = 0 \quad \text{if} \quad t \geq t_b
\]

is the characteristic function of the interval \([t_a,t_b] \). and where \( t_0, t_1, \ldots, t_n \) are the time points when the vicinity of the node is disturbed.

In our simulator, we make a linear approximation of the real voltages, and thus assume the elementary functions to be of a linear form. This leads to a definition of the elementary functions in the following way (see also fig. 2.3).

\[
e(t, iv, sv, Tt, tstab) = \begin{cases} 
    iv & \text{if } t < tstab-Tt \\
    iv + (sv-iv)(t-tstab+Tt) & \text{if } t \geq tstab-Tt \text{ and } t < tstab \\
    sv & \text{if } t \geq tstab 
\end{cases}
\]

where \( p = \{iv, sv, Tt, tstab\} \)

![fig. 2.3: Example of an elementary function e(t,p)](image)

In a logic simulator we have to work with the logic state X, which represents the uncertain states during simulation. Because a
single voltage function for each node doesn’t suffice then to characterize its behavior, we define each node to have a minimum voltage \( u_{\text{min}}(t) \) and a maximum voltage \( u_{\text{max}}(t) \) instead of one \( u(t) \). The real voltage of a node is always supposed to be somewhere between these two voltages. The functions \( u_{\text{min}}(t) \) and \( u_{\text{max}}(t) \) are represented by \( e_{\text{min}}(t), v_{\text{min}}, s_{\text{min}}, T_{\text{tmin}}, t_{\text{stabmin}} \) and \( e_{\text{max}}(t), v_{\text{max}}, s_{\text{max}}, T_{\text{tmax}}, t_{\text{stabmax}} \) respectively, with both of them defined in a similar way as \( e(t), iv, sv, Tt, tsab \).

During simulation, computation of \( e_{\text{min}}(t,p) \) and \( e_{\text{max}}(t,p) \) is done in such a way that worst and best cases with respect to transistors with an Undefined state and nodes with different minimum and maximum voltages, yield an \( e_{\text{max}}(t,p) \) which will always be above any possible \( e(t,p) \), and an \( e_{\text{min}}(t,p) \) which will always be below any possible \( e(t,p) \). This means that \( e_{\text{max}}(t,p) \) will always have a stable voltage which is an upper bound for the stable node voltage, and \( e_{\text{min}}(t,p) \) a stable voltage which is a lower bound, and that the function \( e_{\text{max}}(t,p) \) will always rise as fast and fall as slow as possible, while \( e_{\text{min}}(t,p) \) will always rise as slow and fall as fast as possible. The fast and slow transitions are reflected in minimum and maximum values of \( Tt \). The use of an \( e_{\text{min}}(t,p) \) and an \( e_{\text{max}}(t,p) \), also allows for a min-max delay simulation.

Now we will define the logic state \( L\text{STATE} \) of a node. From the real state of a node (described by its minimum and maximum elementary functions), it derives a state which is used to control the devices which are connected to the node. Although \( L\text{STATE} \) may depend on the devices, and slopes may be incorporated in it, we define an uniform \( L\text{STATE} \) for each node by

\[
L\text{STATE} = \begin{cases} 
1 & \text{if } u_{\text{min}}(t) > v_{\text{switch}} \\
0 & \text{if } u_{\text{max}}(t) < v_{\text{switch}} \\
X & \text{else}
\end{cases}
\]

where \( v_{\text{switch}} \) is the logic switching threshold voltage. When we approximate the real voltages of a common digital MOS circuit (resulting from a SPICE simulation) by linear functions we see that \( v_{\text{switch}} \) is about halfway between the minimum and maximum circuit voltage. (see fig. 2.4).

![fig. 2.4: The logic switching voltage vswitch](image)

Therefore we consider \( v_{\text{switch}} \) to be equal to \((vI-vO)/2\) for the whole network. With \( vI \) and \( vO \) defined by
In Fig. 2.5 an example is given how the logic state of a node depends on its minimum and maximum voltage. We see that it is possible that there are two logic state transitions within one time interval \((t_{1,3})\) in Fig. 2.5. The logic state can change from 0 to 1, or from 1 to 0 via the X state. This for example, will occur when a min-max delay simulation is performed.

The voltage \(v_{\text{switch}}\) forms an exact partition between the 0 and 1 state. When minimum and maximum voltage both stabilize just above or below \(v_{\text{switch}}\), a LSTATE of 1 or 0 will result. However, because for example the "Closed" resistances of enhancement transistors controlled by this LSTATE, are not valid for a voltage just above \(v_{\text{switch}}\), and an enhancement transistor will not completely be "Open" for a voltage just below \(v_{\text{switch}}\), it is more correct when stable voltages in the neighborhood of \(v_{\text{switch}}\) cause a logic state uncertainty. Therefore we introduce an X band for the stable voltages \(v_{\text{min}}\) and \(v_{\text{max}}\) by

\[
\begin{align*}
\text{LSTATE:} & \quad X, I, O, X, I \\
\text{v}_{\text{switch}} & \quad \text{emix(t,p)} \\
\text{emin(t,p)} & \quad \text{emax(t,p)} \\
\end{align*}
\]

Fig. 2.5: Example of the LSTATE in relation with the minimum and maximum voltage. At \(t_1\), \(t_2\), and \(t_3\) new elementary functions are determined.

When \(v_{\text{min}}\) or \(v_{\text{max}}\) appears to be between these bounds, we increase \(v_{\text{max}}\) to \(v_{\text{min}}\) when it is less than \(v_{\text{min}}\), and we decrease \(v_{\text{min}}\) to \(v_{\text{max}}\) when it is more than \(v_{\text{max}}\). In this way the X state will result as the stable state, when \(v_{\text{min}}\) or \(v_{\text{max}}\) appears to be in the X band.
2.3 Simulation procedure

During the simulation there is an event list which contains the nodes which will change logic state at a certain time. Together with the node, the time of the logic state transition and the next logic state is stored. The nodes are sorted according to their time of logic state transition. The simulation is done by repeatedly performing the following steps (see also the scheme at the end of this paragraph): The next first event (node) is fetched from the event list and each event which is also scheduled for that time is also fetched from the event list. When the logic state of such a node changed to X, while its stable state is O or I, the node is scheduled for a second logic state transition. First, each transistor which is connected by its gate to a node which was fetched from the event list, is updated by setting its conduction state according to the logic state of its gate. After this, the vicinity of the following nodes is evaluated: The vicinity of the drain and source of each transistor which was updated, when they are Normal nodes, and the vicinity of Normal nodes which are connected to a Forced node which was fetched from the event list through a Closed or Undefined transistor.

Evaluating a vicinity means that new minimum and maximum elementary functions are determined for each node of the vicinity, and that new events are scheduled, or pending events are rescheduled or retrieved, according to these new functions. The time of a possible logic state transition of a node is calculated from its elementary functions by means of the definition of LSTATE. When a node changes logic state in this way, it has to be scheduled on the event list only when it is essential. That is, when it is an output or has transistor gates connected to it. Fetching a group of events from the event list, updating the transistor states, and then evaluating the relevant vicinities, is called a simulation step. During a simulation step flags are set to prevent unnecessary reevaluations within the same simulation step. After a simulation step there may have been scheduled new events for the same time as the last simulation step was. These events, which are a result of the vicinity evaluations during the last simulation step, have been scheduled with zero delay. All simulation steps which occurred at the same time during a simulation, are together called one current time step. When there are more simulation steps within a current time step than the logic depth of the circuit, an oscillation is going on in the network which will never stop. These oscillations are caused by race conditions in feedback loops in the network. They for example may occur with flip-flops. Because a race condition implies an undefined state, the nodes which are still scheduled on the event list for a logic state transition to O or I at the current time, are set to X by assigning VO and VI to their minimum and maximum voltages. When doing this for each resulting simulation step within the current time step, the right minimum and maximum voltages will be calculated automatically in this way.

Determining a new minimum and maximum elementary function consists of two phases. First we have to determine new svmin and svmax values (stable voltage determination: chapter 3). And secondly we
have to determine a new $t_{stastbmin}$, $t_{stastbmax}$, $T_{tmin}$ and $T_{tmax}$ (dynamical behavior determination: chapter 4). A new $iv_{min}$ and $iv_{max}$ are given by the actual minimum and maximum voltage at the current time.

The scheme of the simulation procedure is summarized below. The current time is represented by $t_{curr}$, the variable $logic\_depth$ is considered to be equal to the maximum logic depth of the simulated network, and the function $time\_next\_event$ returns the time for which the next event on the event list has been scheduled.

```plaintext
simulate () {
    for (all nodes) {
        $iv_{min} = sv_{min} = v_0$;
        $iv_{max} = sv_{max} = v_1$;
    }
    for (all transistors) {
        state = Undefined;
    }
    fill event_list with state transitions of Forced nodes;

    while (event_list is not empty) {
        $t_{curr} = time\_next\_event ();$
        current_time_step ();
    }
}

current_time_step () {
    step = 1;
    while (time_next_event () == $t_{curr}$
        && step <= logic_depth ) {
        simulation_step ();
        step = step + 1;
    }

    while (time_next_event () == $t_{curr}$) {
        for (all nodes n on event_list for $t_{curr}$,
            which will change logic state to 0 or 1) {
            $iv_{min} = sv_{min} = v_0$;
            $iv_{max} = sv_{max} = v_1$;
        }
        simulation_step ();
    }
}
```

2.102
simulation_step () {
    for (all nodes n on event_list for tcurr) {
        update state of transistors which are by their gate connected to n;
    }

    for (all nodes n on event_list for tcurr) {
        retrieve n from event_list;
        evaluate vicinities of the Normal nodes which are
        - drain or source of transistors which were updated
        - through a Closed or Undefined transistor connected to n, when n is Forced;
    }
}

3. Stable voltage determination

The determination of new stable voltages svmin and svmax for the nodes of a vicinity which is disturbed, is done by a resistance division and a charge sharing. The resistance division accounts for the influence of Forced nodes on the vicinity by means of the resistances between the nodes, and charge sharing accounts for the influence of the initial charges present in the vicinity. The final svmin and svmax which are assigned to a node, are the minimum and maximum of the values found by resistance division and charge sharing.

When svmin or svmax is found to be between vminI and vmaxO, svmin will be decreased to vmaxO when svmin was more, and svmax will be increased to vminI when svmax was less. In this way the stable state X will result.

3.1 Resistance division

The idea behind resistance division is to assume for each node one dominant minimum resistance path to Forced nodes with the I state (with total minimum path resistance I_r), and one dominant minimum resistance path to Forced nodes with the 0 state (with total minimum path resistance O_r), and then compute the stable node voltages by $sv = vO + (vI - vO) O_r / (O_r + I_r)$.

Minimum path searching is done by the following algorithm. For each node in the vicinity it determines the total minimum path resistance "r" to a set of "target nodes" and a pointer "path" to the next node of this path, in a time about proportional to the number of nodes. The front_list which is used in the algorithm, contains the front nodes of the searching process. The searching process starts on the nodes connected to the target nodes by one transistor.
for ( all nodes n in the vicinity ) n.path = NULL;

for ( all transistors t in the vicinity ) {
    if ( transistor t makes a connection between a
target node t_n and a Normal node n which
is not a target node ) {
        if ( n_n.path == NULL ) {
            n_n.path = t_n;
            n_n.r = t.resistance;
            put n_n on front_list
        } else if ( n_n.r > t.resistance ) {
            n_n.path = t_n;
            n_n.r = t.resistance;
        }
    }
}

while ( front_list is not empty ) {
    n = node on front_list with minimum r;
    delete n from front_list;
    for ( all transistors t with drain or source is n ) {
        c_n = node connected by t to n;
        if ( c_n.type != Forced and
            c_n is not a target node ) {
            resist = n.r + t.resistance;
            if ( c_n.path == NULL ) {
                c_n.path = n;
                c_n.r = resist;
                put c_n on front_list
            } else if ( c_n.r > resist ) {
                c_n.path = n;
                c_n.r = resist;
            }
        }
    }
}

Because there may be transistors in the Undefined state, or Forced
nodes with an X state, the above algorithm is done in four ways.
It is done with I state Forced nodes as target nodes and only
going through transistors in the Closed state. This yields worst
case minimum paths to I state Forced nodes by means of wcI_path
and wcI_r. Next, 0 state Forced nodes as target nodes and only
going through Closed transistors will yield worst case minimum
paths to O state Forced nodes by means of wcO_path and wcO_r.
Thirdly, I and X state Forced nodes as target nodes and going
through Closed and Undefined transistors, yielding best case
minimum paths to I state Forced nodes by means of bcI_path and
bcI_r. And finally with O and X state Forced nodes as target
nodes, and going through Closed and Undefined transistors,
yielding best case minimum paths to O state Forced nodes by means
of bcO_path and bcO_r. The stable voltages svmin and svmax for a
Normal node are then found by

2.104
svmin :
if ( bcO_path == NULL )
    svmin = vI;
else if ( wcI_path == NULL )
    svmin = vO;
else
    svmin = vO + (vI - vO) \frac{bcO_r}{bcO_r + wcI_r};

svmax :
if ( bcI_path == NULL )
    svmax = vO;
else if ( wcO_path == NULL )
    svmax = vI;
else
    svmax = vO + (vI - vO) \frac{wcO_r}{wcO_r + bcI_r};

The above resistance divisions hold only for nodes which are in the static current flow. That is, when the paths to I state forced nodes and 0 state forced nodes don't have a common part. When the four paths together do have a common part, we have to trace the path back and calculate the voltages in the above way for the last node lying on this common part. The resulting voltages are assigned to all nodes lying on the common part. A common part which has to be traced, exists as long as bcO_path, bcI_path, wcO_path and wcI_path are pointing to the same node, or wcO_path and/or wcI_path are NULL. When bcO_path and bcI_path are different or wcO_path and wcI_path are different, the ultimate last common node is reached. But when bcO_path and bcI_path together, are different from wcO_path and wcI_path together (wcO_path or wcI_path may be NULL again), the paths may be temporarily apart, and we have to trace both paths to find a possible next common part. This last case will occur only sometimes when there are Undefined transistors lying in a parallel branch which is not in the static current flow (see fig. 3.1).
Example of a resistance division (see fig. 3.1): For each node the minimum resistance paths and values, and the svmin and svmax, are given

\[ \begin{align*}
v_0 &= 0 \quad R_{\text{enh}} = 1 \\
v_1 &= 5 \quad R_{\text{depl}} = 10 \\
\end{align*} \]

<table>
<thead>
<tr>
<th>node</th>
<th>bcI_r</th>
<th>wcI_r</th>
<th>bcO_r</th>
<th>wcO_r</th>
<th>svmin</th>
<th>svmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>11</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>5/12</td>
<td>5/12</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>10/12</td>
<td>10/12</td>
</tr>
<tr>
<td>c</td>
<td>11</td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>10/12</td>
<td>10/12</td>
</tr>
<tr>
<td>d</td>
<td>12</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>10/12</td>
<td>10/12</td>
</tr>
<tr>
<td>e</td>
<td>12</td>
<td>13</td>
<td>4</td>
<td>5</td>
<td>10/12</td>
<td>10/12</td>
</tr>
<tr>
<td>f</td>
<td>13</td>
<td></td>
<td>5</td>
<td>-</td>
<td>10/12</td>
<td>10/12</td>
</tr>
<tr>
<td>g</td>
<td>13</td>
<td>14</td>
<td>5</td>
<td>6</td>
<td>10/12</td>
<td>10/12</td>
</tr>
</tbody>
</table>

### 3.2 Charge sharing

To do a charge sharing, we first split the vicinity into groups of nodes. Two nodes are said to be in the same group when there is at least one path between them with all transistors in the Closed state. So the groups within a vicinity will be connected to each other by means of Undefined transistors. Nodes within a group can be considered as a unit with charge sharing, because they will always be connected to each other. We will determine a svmin and svmax for each group, and to each node of the group these values will be assigned.

Now we first consider all Undefined transistors to be in the Open state. That is, we first consider the isolated groups. For each group we can compute a first minimum and maximum stable voltage by first computing the total minimum and maximum charge of the group and then dividing these values by the total group capacity.
Minimum and maximum charge are found for each node by multiplying the node capacity with minimum and maximum actual voltages respectively. A logical state is assigned then to each group according to its minimum and maximum voltage by:

\[
\text{isolated group state} = I \quad \text{if } svmin^1 \geq vminI
\]
\[
\quad = 0 \quad \text{if } svmax^1 \leq vmaxO
\]
\[
\quad = X \quad \text{else}
\]

Next, a new minimum and maximum stable voltage for each group is determined by considering the possibility that a group minimum voltage may decrease, or a maximum voltage increase, when the group is connected to other groups. Here we use the information that we can classify the groups, according to their isolated group state, in three kinds, with each kind having its own minimum and maximum voltage bounds. The capacitances of these groups (excluding the group whose voltages we want to calculate) are summed in the variables capI, capO and capX. The new group voltages are then found by:

\[
svmin^2 = \text{MIN} (\frac{qmin+vO.(capO+capX)}{cap+capO+capX}, \frac{qmin+vO.(capO+capX)+vminI.capI}{cap+capO+capX+capI})
\]
\[
svmax^2 = \text{MAX} (\frac{qmax+vI.(capI+capX)}{cap+capI+capX}, \frac{qmax+vI.(capI+capX)+vmaxO.capO}{cap+capI+capX+capO})
\]

The second value of the MIN and MAX operators will be chosen when the first value appears to be more than vminI (I groups will decrease lower bound then), or less than vmaxO (O groups will increase upper bound then), respectively.

The values for svmin^2 and svmax^2 can be obtained within a time proportional to the number of nodes in the vicinity and are always safe. In fact they are often too pessimistic when there is more than one group in a vicinity. Based on the topology of the groups in the vicinity, the minimum can be increased or the maximum can be decreased, sometimes.

We say that a group is Strong when svmax^2 and svmin^2 are both below vmaxO, or when they are both above vminI. Otherwise a group is called Weak. Strong groups are able to stay O or I, no matter how they are connected to the other groups. By this property they can also block the X setting influence from groups to other
groups. It can be seen that all Strong groups in a vicinity do have the same logic state.

We now say that each group which has an isolated group state unequal to the state of the Strong groups, is unprotected. If there are no Strong groups, each group is said to be unprotected. We perform the following algorithm on the vicinity: Each group which is unprotected sets the unprotection of its neighbors which are not Strong. New unprotected groups do the same to their neighbors. After doing this recursively, to Weak groups which are not unprotected an adjustment is made by increasing $s_{\text{min}}$ to the minimum of the voltages $s_{\text{min}}$ of the not unprotected Weak groups and the voltages $s_{\text{min}}$ of the Strong groups (when the isolated group state was 1), or decreasing $s_{\text{max}}$ to the maximum of the voltages $s_{\text{max}}$ of the not unprotected Weak groups and the voltages $s_{\text{max}}$ of the Strong groups (when the isolated group state was 0).

When there are Forced nodes in the vicinity when a charge sharing is done, the groups with Forced nodes in them are eliminated from the vicinity. In this way separate agglomerations of groups will be obtained which are not connected to each other (see fig. 3.2). The different agglomerations of groups are evaluated one after another in the same way as described above.

---

fig. 3.2: Groups and agglomerations in a vicinity.

fig. 3.3: Example of a charge sharing with Weak and Strong groups.

Example of a charge sharing with Weak and Strong groups (see fig. 3.3): There are three groups, each consisting of only one node. For each group the successive values which are calculated, are shown.
4. Dynamical behavior determination

To simulate the timing behavior of the network, the dynamical behavior parameters $t_{\text{stabmin}}$, $t_{\text{stabmax}}$, $T_{\text{tmin}}$ and $T_{\text{tmax}}$ have to be determined. In combination with the other parameters ($v_{\text{min}}$, $v_{\text{max}}$, $s_{\text{min}}$ and $s_{\text{max}}$), we then are able to predict the voltage waveforms on the nodes, and we can calculate at what time the logic states of the nodes will change.

The determination of the dynamical behavior parameters $t_{\text{stabmin}}$, $t_{\text{stabmax}}$, $T_{\text{tmin}}$ and $T_{\text{tmax}}$ is given by a general determination of the parameters $t_{\text{stab}}$ and $T_{\text{t}}$. The dynamical behavior parameters of the minimum and maximum elementary function are found by setting Undefined transistors and X state Forced nodes in such a state and choosing node voltages from their minimum and maximum voltages in such a way, that $e_{\text{min}}(t,p)$ rises as slow as possible and falls as fast as possible, and $e_{\text{max}}(t,p)$ rises as fast as possible and falls as slow as possible. In case of a min-max delay simulation, the fast rising or falling waveform is sped up with the proportional delay deviation and the slow rising or falling waveform is delayed with it.

The point is that we have to choose a $t_{\text{stab}}$ and $T_{\text{t}}$ for each node which results in an elementary function $e(t,p)$ which is as close as possible to the real $u(t)$. As a result of modeling a transistor by a switch, resistor and capacitors, we will have a vicinity which is an RC network. Therefore we will first pay attention to RC networks driven by one voltage step, of which the theory is known, and then try to extend the solutions to the requirements in our simulator model.

4.1 RC networks driven by one voltage source

Consider a network consisting of nodes which are connected by means of linear resistors and each node having a linear capacitance to ground. The network is driven by a voltage step on one input node (see fig. 4.1).
In [12] upper and lower boundaries for the voltage functions are derived when the network is a tree with the input being the root of the tree, and when all initial node voltages are equal. The upper and lower boundary voltage functions are complex functions based on three time constants. To a node \( i \) one of these constants: \( T_{Di} \), is equal to the first-order moment of the input response. When the resistance \( R_{ki} \) is defined as the resistance of the portion of the (unique) path between the input and node \( i \), that is common with the (unique) path between the input and node \( k \), then \( T_{Di} \) is found by the following sum over all the capacitances in the network.

\[
T_{Di} = \sum_{k} R_{ki} C_k
\]

e.g. in fig 4.1 \( T_{D3} = R_1 C_1 + (R_1 + R_2)C_2 + (R_1 + R_2 + R_3)C_3 + (R_1 + R_2 + R_3 + R_4)C_4 + (R_1 + R_2 + R_3 + R_4 + R_5)C_5 \)

Although the upper and lower boundary functions are not directly suited for our linear approach, it is also proved in [12] that when the voltage step is from \( v_1 \) to \( v_2 \) at \( t=0 \), the node voltage \( u_i(t) \) on node \( i \) satisfies

\[
\int_0^\infty (v_2-u_i(t))dt = (v_2-v_1)T_{Di}
\]

This provides us with the opportunity to choose the best fitting linear approximation. When we have \( iv_i = v_1 \) and \( sv_i = v_2 \) then

\[
\int_0^\infty (v_2-e_i(t,p))dt = (v_2-v_1)(t_{stab} - \frac{1}{2}T_{Ti})
\]

When we assume that the elementary function immediately starts rising or falling, we have \( t_{stab} = T_{Ti} \). The best fitting approximation is then found by \( T_{Ti} = 2T_{Di} \). Then

\[
\int_0^\infty (u_i(t)-e_i(t,p))dt = 0
\]

Which means that the area of \( e_i(t,p) \) above \( u_i(t) \) is just as large as below it (see fig. 4.2).
4.2 Linking theory to simulation

The above discussion was founded on theory. But although our simulator uses already a simplification of the real circuit, the above theory can't be used directly in our simulator. We have to adapt the theory to the practice of our more general simulation model. The result will be that the simulator finds correct solutions in situations which are described above, and that the results will be close to reality when more complicated situations occur.

First we have to face the fact that a vicinity can not always be seen as a network driven by one input. Often there are Forced nodes with different states in the same vicinity, and when charge sharing is done, it isn't even possible to identify a real driver. We therefore state the following: When we want to find a $T_{Di}$ for rising voltages, we first say the driver of the vicinity to be the united set of all Forced nodes with state 1. If they are not present in the vicinity, we choose the node with the largest charge $C(v_f-v_0)$. In a similar way we choose for falling voltages first the 0 state Forced nodes and then the node with largest charge $C(v_I-v_f)$.

Next it isn't always true that all nodes will reach the same stable voltage. This is often a result of the presence of Forced nodes with different states. We state that a smaller difference between $sv$ and $iv$ will have less influence on $T_{Di}$, and therefore we determine $T_t$ in the following way: We determine a $T_{Di}^V$ instead of a $T_{Di}$. When $sv_k$ and $iv_k$ are the stable and initial voltage of node $k$, $T_{Di}^V$ is found by

$$T_{Di}^V = \sum_{k} R_{ki} C_k (sv_k - iv_k)$$

$T_t$ for node $i$ is found then by $T_t_i = 2T_{Di}^V/(sv_i - iv_i)$. 

---

**Fig. 4.2** : approximation with elementary function

A way of determining $T_{Di}$ when the network is not a tree but a general RC mesh, is described in [13]. Here it is shown that the $R_{ki}$'s are elements of an $R$ matrix which gives each node voltage as linear function of the currents in the capacitors. In [14] finally, it is shown that $T_{Di}$ can also be calculated when the nodes do have different initial voltages. Each product term in the expression for $T_{Di}$ has to be multiplied then by $(v_2-u_k(0))/(v_2-v_1)$. 

4.2 Linking theory to simulation

The above discussion was founded on theory. But although our simulator uses already a simplification of the real circuit, the above theory can't be used directly in our simulator. We have to adapt the theory to the practice of our more general simulation model. The result will be that the simulator finds correct solutions in situations which are described above, and that the results will be close to reality when more complicated situations occur.

First we have to face the fact that a vicinity can not always be seen as a network driven by one input. Often there are Forced nodes with different states in the same vicinity, and when charge sharing is done, it isn't even possible to identify a real driver. We therefore state the following: When we want to find a $T_{Di}$ for rising voltages, we first say the driver of the vicinity to be the united set of all Forced nodes with state 1. If they are not present in the vicinity, we choose the node with the largest charge $C(v_f-v_0)$. In a similar way we choose for falling voltages first the 0 state Forced nodes and then the node with largest charge $C(v_I-v_f)$.

Next it isn't always true that all nodes will reach the same stable voltage. This is often a result of the presence of Forced nodes with different states. We state that a smaller difference between $sv$ and $iv$ will have less influence on $T_{Di}$, and therefore we determine $T_t$ in the following way: We determine a $T_{Di}^V$ instead of a $T_{Di}$. When $sv_k$ and $iv_k$ are the stable and initial voltage of node $k$, $T_{Di}^V$ is found by

$$T_{Di}^V = \sum_{k} R_{ki} C_k (sv_k - iv_k)$$

$T_t$ for node $i$ is found then by $T_t_i = 2T_{Di}^V/(sv_i - iv_i)$. 

---

2.111
As a conclusion of the previous paragraph, $T^V_{Di}$ can be found in two different ways. We can use the tree method, and we can use the $R$ matrix method. Only the second one takes into account the parallel branches in an exact manner, but the first one will be a faster method. With the tree method it is possible to determine a minimum and a maximum $T_t$ for all nodes in a time about proportional to the number of nodes.

First a minimum resistance path search with the driver as the target, is done by the algorithm as described with "resistance division". Automatically a tree is formed in this way. Similar as with resistance division, we search for a worst case path and a best case path. Then the node charges $C_k(s_{k-1} - v_k)$ ($= q_k$) are accumulated from the 'leaves' of the tree to the 'driver'. That is, when $aq_k$ is the accumulated charge at node $k$, and the path to the driver for node $k$ points to node $k-1$, then

$$aq(k-1) = q(k-1) + aq_k$$

For the minimum $T_t$, the best case path is followed to this, and for the maximum $T_t$, the worst case path - or by a local absence of the worst case path the best case path - is followed. For the minimum $T_t$, charges are not accumulated through transistors with an undefined state. The $T^V_{Di}$ is then found for each node by back tracing the paths in reverse order. When $r_i$ is the minimum path resistance to the driver for a node $i$, and the path to the driver for node $i$ points to node $i-1$, then

$$T^V_{Di} = T^V_{D(i-1)} + r_i aq_i$$

when the path is through an undefined state transistor and a minimum $T_t$ is determined. And

$$T^V_{Di} = T^V_{D(i-1)} + (r_i - r_{i-1}) aq_i$$

when the path is through a Closed state transistor or a maximum $T_t$ is determined.

2.112
1. INTRODUCTION

This manual is meant as a guide to users who want to simulate their network with the sIs simulator. The acronym sIs stands for Switch-Level Simulator, and the simulator can be used for simulating the logical and timing behavior of digital MOS circuits. In the simulator transistors are modeled by grounded capacitors and a switched resistor. Each node in the network has a state 0, 1 or X (for unknown), and each transistor has a state open, closed or undefined. Many characteristics of MOS circuits can be modeled accurately, including: ratioed, complementary and precharged logic; dynamic and static storage; pass transistors; busses; and charge sharing. Because the simulator performs local-event-driven simulation, also large networks, with thousands of transistors, can be simulated in a reasonable time.

The sIs simulator is capable of simulating a MOS transistor network at three levels.

1. purely logic simulation based on network topology and transistor types, without considering the actual circuit parameters.
2. logic simulation based on actual circuit parameters (node capacitances and transistor and interconnection resistances are used to determine logic states).
3. logic and timing simulation based on actual circuit parameters (node capacitances and transistor and interconnection resistances are used to determine logic states and delays).

The original switch-level model – which is only capable of a simulation at the first level – was introduced by R.E. Bryant. Descriptions of it are given in [5] and [6]. The principle of the sIs simulator is described in [1].

Section 2 in this manual gives an overview of the sIs package of programs.
Section 3 describes the syntax and semantics of the network description language.
Section 4 contains the syntax and semantics of the command language.
2. DESCRIPTION OF THE SLS-PACKAGE

The switch level simulation package consists of a number of programs. In figure 1 the relation between the three main programs is outlined. In this figure you will find the following symbols:

Rectangle  Rectangles stand for executable programs. The name of a particular program is printed inside the rectangle.

Punch-card  A punch-card symbol stands for one file. A deck of punch-cards stands for multiple files.

Arrow  An arrow denotes the information flow between programs and files. Programs can read/write information from/to several different files.

Figure 1. Information flow diagram of the SLS-package.
The SLS-package involves three main programs.

1. **sls_mkdb** maps a hierarchical **sIs** network description contained in one or several files into a network description in database format.

2. **sls_exp** maps the network description in database format into a binary (i.e. non readable) file which serves as input file for the switch level simulator. Actually a collection of binary files can be created in the database, because first a binary file for each subnetworks is created once. The expansion program links the binary files of the subnetwork into one binary file of the total network.

3. **sIs** is the switch level simulation program and reads input from two files:
   - A binary file: This file is generated by **sls_exp** and contains the network description in **sIs** format.
   - A command file: This file contains the simulation commands and must be generated manually by the designer.

**sIs** generates two output files:

- **modname.out**: This file is the normal readable and printable output in table format. Here modname stands for the model that is simulated.
- **modname.res**: This file is not readable and serves as input for several post processing facilities or for other simulations.

Other programs that belong to the package involve post processing facilities producing different sorts of output.
3. THE DESCRIPTION OF NETWORKS

3.1 General conventions.

It is common practice to describe the syntax of a computer language in some meta language. The syntax definition of SLS is described in the meta language proposed by Wirth [15]. This language has two types of symbols:

terminal symbols These symbols must be present literally. They are denoted by words between double quote marks, or words printed in italic font.

non-terminal symbols These are denoted by words typed in lowercase.

Each production rule begins with a non-terminal followed by an equal-sign and a sequence of terminal and non-terminal symbols and meta characters terminated by a period. The meta characters imply:

Alternatives | A vertical bar between symbols denotes the choice of either one symbol or another symbol. (i.e. a | b means either a or b)

Repetition {} Curly brackets denote that the symbols in between may be present zero or more times. (i.e. { a } stands for empty | a | aa | aaa | ...)

Optionality [] Square brackets denote that the symbols between them may present. (i.e. [ a ] stands for a | empty)

Parenthesis () Parenthesis serve for grouping of symbols in meta character expressions. (i.e. ( a | b ) c stands for ac | bc)

Next the following rules are applied:

- A comment begins with "/*" and is terminated by "*/" as in the C-language. Every character in between is discarded by the parser. You are not allowed to nest comments.

- Every name is an identifier which must begin with a letter and may be followed by an arbitrary number of letters, digits and underscore characters.

- Uppercase and lowercase letters are distinct.
The blank, tab and newline are separators. Between two symbols an arbitrary number of separators may appear.

Values are specified in S.I. units, and may appear with one of the following scaling factors:

\[
\begin{align*}
    f &= 1.0 \times 10^{-15} \\
    p &= 1.0 \times 10^{-12} \\
    n &= 1.0 \times 10^{-9} \\
    u &= 1.0 \times 10^{-6} \\
    m &= 1.0 \times 10^{-3} \\
    k &= 1.0 \times 10^3 \\
    M &= 1.0 \times 10^6 \\
    G &= 1.0 \times 10^{12}
\end{align*}
\]

Keywords are reserved words and may not be used as names by the designer.

**TABLE 1. Reserved keywords in network descriptions**

<table>
<thead>
<tr>
<th>cap</th>
<th>penh</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>res</td>
</tr>
<tr>
<td>ndep</td>
<td>terminal</td>
</tr>
<tr>
<td>nenh</td>
<td>tr</td>
</tr>
<tr>
<td>net</td>
<td>tf</td>
</tr>
<tr>
<td>network</td>
<td>w</td>
</tr>
</tbody>
</table>

The basic lexical constructs are:

- **power_ten** = "1""0"["f""p""n""u""m""k""M""G"]
- **f_float** = float("f""p""n""u""m""k""M""G")
- **float** = (integer).integer[exponent]
- **exponent** = ("D""E""d""e")["-""+"]integer.
- **identifier** = letter | digit | "." | "+" |
- **character** = digit | letter | special.
- **integer** = digit | digit.
- **letter** = "a""b""c""d""e""f""g""h""i""j""k""l""m""n""o""p""q""r""s""t""u""v""w""x""y""z""A""B""C""D""E""F""G""H""I""J""K""L""M""N""O""P""Q""R""S""T""U""V""W""X""Y""Z".
- **digit** = "0""1""2""3""4""5""6""7""8""9".
- **empty** = "
- **special** = any character that is not a letter or digit.
Examples:

```
power_ten : 10n 100p 1000u
f_float : 1.3e-9 2.0n 6.8k 0.0004
identifier: this_IS_an_identifier
string : "this is a string/8*
```

### 3.2 Network Description

**TABLE 2. Syntax of the network part.**

<table>
<thead>
<tr>
<th>Network Decli</th>
<th>=</th>
<th>Network_Dcl (Network_Dcl).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network decl</td>
<td>=</td>
<td>Network identifier decl_part ntw_body.</td>
</tr>
</tbody>
</table>

The network part begins with keyword **network** followed by the name of the network, a declaration part describing the interface to the outside world and a body specifying the instances of devices and subnetworks.

### 3.3 Declarations

**TABLE 3. Syntax of the declaration part.**

<table>
<thead>
<tr>
<th>Decl_part</th>
<th>=</th>
<th>&quot;(&quot; Term_decls &quot;)&quot;.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term_decls</td>
<td>=</td>
<td>Term Decl &quot;;&quot; Term Decl.</td>
</tr>
<tr>
<td>Term Decl</td>
<td>=</td>
<td>Terminal Term &quot;,&quot; Term.</td>
</tr>
<tr>
<td>Term</td>
<td>=</td>
<td>Identifier [&quot;[&quot; Range_list &quot;]&quot;].</td>
</tr>
<tr>
<td>Range_list</td>
<td>=</td>
<td>Range &quot;,&quot; Range.</td>
</tr>
<tr>
<td>Range</td>
<td>=</td>
<td>Integer &quot;,&quot; Integer.</td>
</tr>
</tbody>
</table>

The declaration part is bracketed by a left and right parenthesis. It must contain at least one terminal declaration. A terminal declaration begins with the keyword **terminal** followed by a list of terminals. A list of terminals consists of one or more terminals separated by commas. Terminals may have an array structure to a maximum of 10 indices.

Examples:

```
network register (terminal in[1..4,1..4], out[1..8])
network latch (terminal in, out;
               terminal phi[0..1], vss, vdd)
```

### 3.4 Network Structure

The network body (ntw_body) is the statement part (stmt_part) bracketed by curly brackets ({}). The statement part may contain instance, net and null statements. The scope of the terminals and instance names is local to the defined network.
TABLE 4. Syntax of the network body.

<table>
<thead>
<tr>
<th>ntw_body</th>
<th>= &quot;{&quot; stmt_part &quot;}&quot;.</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt_part</td>
<td>= statement (statement).</td>
</tr>
<tr>
<td>statement</td>
<td>= inst_stmt</td>
</tr>
<tr>
<td>inst_stmt</td>
<td>= [inst_struct] inst_def.</td>
</tr>
<tr>
<td>inst_struct</td>
<td>= &quot;{&quot; identifier [range_list] &quot;}&quot;.</td>
</tr>
<tr>
<td>inst_def</td>
<td>= transistor_def</td>
</tr>
<tr>
<td>null_stmt</td>
<td>= &quot;:&quot;.</td>
</tr>
<tr>
<td>net_stmt</td>
<td>= net &quot;{&quot; net_spec &quot;}&quot; &quot;:&quot;.</td>
</tr>
<tr>
<td>net_spec</td>
<td>= vector_list</td>
</tr>
<tr>
<td>vector_list</td>
<td>= &quot;{&quot; net_nodes &quot;}&quot; &quot;{&quot; net_nodes &quot;}&quot;.</td>
</tr>
<tr>
<td>net_nodes</td>
<td>= node_ref &quot;&quot; node_ref.</td>
</tr>
<tr>
<td>transistor_def</td>
<td>= ttype [attributes] connect_list &quot;:&quot;.</td>
</tr>
<tr>
<td>function_def</td>
<td>= ftype [attributes] connect_list &quot;:&quot;.</td>
</tr>
<tr>
<td>resistor_def</td>
<td>= resistor f_float connect_list &quot;:&quot;.</td>
</tr>
<tr>
<td>capacitor_def</td>
<td>= capacitor f_float connect_list &quot;:&quot;.</td>
</tr>
<tr>
<td>call_def</td>
<td>= identifier connect_list&quot;&quot;.</td>
</tr>
<tr>
<td>attributes</td>
<td>= attribute (attribute).</td>
</tr>
<tr>
<td>attribute</td>
<td>= attr_label &quot;=&quot; attr_val.</td>
</tr>
<tr>
<td>attr_label</td>
<td>= w</td>
</tr>
<tr>
<td>attr_val</td>
<td>= f_float.</td>
</tr>
<tr>
<td>connect_list</td>
<td>= &quot;{&quot; connect &quot;}&quot;</td>
</tr>
<tr>
<td>connects</td>
<td>= connect &quot;&quot; connect.</td>
</tr>
<tr>
<td>connect</td>
<td>= internal_ref</td>
</tr>
<tr>
<td>internal_ref</td>
<td>= [ &quot;{&quot; index_list &quot;}&quot; ] &quot;&quot; node_ref.</td>
</tr>
<tr>
<td>node_ref</td>
<td>= integer</td>
</tr>
<tr>
<td>index_list</td>
<td>= index &quot;,&quot; index.</td>
</tr>
<tr>
<td>index</td>
<td>= integer</td>
</tr>
<tr>
<td>range_list</td>
<td>= range &quot;&quot; range.</td>
</tr>
<tr>
<td>range</td>
<td>= integer &quot;..&quot; integer.</td>
</tr>
</tbody>
</table>

3.4.1 instance_statement. The instance statement constitutes the actual place of a device, function or subnetwork in the network being defined. An instance may optionally have a name and must have a definition part. The instance name is especially convenient for selecting particular terminals of the instance as necessary in the simulation command file. The instance may have an array structure, which is specified by a range_list between square brackets after the instance name. The general format of an instance definition is:

```
name attributes connect_list
```

In each definition a node connection list is specified. This list can be interpreted in two ways:
instance major order  If the list is placed between parenthesis we define it to be in instance major order. Instance major order means that when the instance has array structure, for instance from 1 to 4, the connection list will be interpreted as follows, first the terminal connections for the first array element, then that of the second array element etc.

parameter major order  If the list is placed between curly brackets we define it to be in parameter major order. Parameter major order means that if the instance has array structure, the connection list will consist of first the first terminals of all array elements then the second terminals of all array elements etc.

Examples:

{ im[1..3] } inverter { i1, o1, i2, o2, i3, o3 }

{ pm[1..3] } inverter { i1, i2, i3, o1, o2, o3 }

If the instance is an array of elements one can specify internal connections in the connection list. With an internal connection one can directly interconnect terminals of the array elements of the instance. An internal connection is denoted by selecting a formal terminal of an instanced (possibly array) element and is put into the right place in the connection list.

Example:

{ inv[1..3] } inverter { i1, [1..2].o, [1..2].i, o3 }

where inverter was defined as

network inverter (terminal i, o) {
    .
    .
}
3.4.1.1 transistor definition. A transistor definition contains three arguments.

type specifies the type of the transistor. The following three types can be chosen:

1. nenh
2. penh
3. ndep

attributes The second argument is optional and specifies the attribute values of the device. A transistor can have two attributes for specifying the length and width of a transistor. If omitted the default values for length and width of the transistor are 4 micron for slo.

connection The third argument consists of three or a multiple of three node references, depending on the structure of the instance. The sequence of the nodes is important as far as the gate node is concerned. The gate node must be specified as the first node, while the sequence of the source node and drain node is arbitrary.

Examples:

nenh w=6u l=8u (g1, d1, s1);
nenh w=6u l=8u (g1, d1, s1):
nenh (g1, d1, s1, g2, d2, s2);
ndep (g1, g2, d1, d2, s1, s2):

3.4.1.2 function definition. A function is a built-in function to the simulator. A function definition contains three arguments:

type The following function types are available:

1. nand
2. nor
3. and
4. or
5. exor

attributes The attributes which are optional specifies the rise time and fall time of the output when one of the inputs changes of state.

connection The connection list of a function may contain an arbitrary number of inputs and one output. The order is first the inputs then as last terminal the output.
Examples:

nand (a, b, c, e, d, out)
{ nand1 } nand (a, b, c, e, d, out)
nor tr=10n tf=5n (a, b, c, e, d, out)

3.4.1.3 resistor_definition. The resistor definition is specified by the keyword resistor, the resistance value in Ohm and two node references. The resistance value may be a floating point number and may have a multiplication factor denoted by a letter as described in the section 1.

3.4.1.4 capacitor_definition. The capacitor definition is specified by the keyword capacitor, a capacitance value in Farad and two node references. The simulator assumes that all capacitors have one node connected to ground. The capacitance value may be a floating point number with a multiplication factor.

3.4.1.5 call_definition. A network call defines the instance of a subnetwork. The sequence of the connection list is the same as the sequence of the terminal declaration of the called network.

3.4.2 net_statement. The net statement gives the opportunity to coalesce, cluster and define nodes. This statement is especially convenient if the network description is extracted from a layout description. In a layout description terminals of ten represent the same electrical node and it would be tedious to specify a stimulus for each of these terminals.

coalesce Normally all specified terminals between curly brackets are coalesced to one electrical node.

cluster A list of parenthesized lists of terminals means that the the terminals between the parenthesis will be coalesced element wise. The number of elements in all parenthesized lists must be equal!

define If a terminal in the list between curly brackets is not defined it will be defined and can function as local node.

Examples:

net { a[1..8] b c }
net { {a[1..4]} (b c d e) }
net { {a[1..4]} }

2.122
3.5 Examples

3.5.1 latch_example. A latch is a two phase clocked register element. Figure 2 shows the transistor diagram and the network description of the latch circuit. The circuit consists of three inverter stages. Two of these stages are connected as a flipflop by phi2_l. The input signal is clocked by phi1, whereas the output stage is clocked by phi2_r.

Figure 2. Transistor diagram and network description of the latch circuit.

Note that the two terminals phi2_l and phi2_r are coalesced to one terminal named phi2 in the net declaration, for these terminals represented the same electrical node.
3.5.2 hierarchical_latch_example. The hierarchical network description of the latch is shown in figure 3. The latch consists of three inverter networks connected by pass transistors. Inside each inverter network you will find terminals that have equal names. These terminals represent feed-throughs in the inverter and are electrically equivalent.

network invert (terminal i, o, vdd, gnd)
{  
enh w=8u l=4u (i, o, gnd);  
nde p w=8u l=4u (o, vdd, o);  
}

network latch (terminal vdd, vss, phi1, phi2, out, in)
{  
net {phi2, phi2_r, phi2_l}; /* equivalent nodes */  
{inv[1..3]} invert (6, 9, vdd, vss,  
9, 7, vdd, vss,  
10, out, vdd, vss);  
enh w=8u l=4u (phi1, in, 6);  
enh w=8u l=4u (phi2_l, 6, 7);  
enh w=8u l=4u (phi2_r, 9, 10);  
}

Figure 3. Hierarchical description of the latch circuit.
3.5.3 \textit{latch\_with\_nand\_example}. The previous latch example only contained transistors. Besides transistors, resistors and capacitors it is also possible to use function elements (ands, ors, nands, nors and exors) in the network description. In the following example the network inverter of the hierarchical latch circuit has been changed. The nenhancement and depletion transistor have been substituted by a nand gate with one input. The total network description is now.

```
network invert (terminal i, o)
{  nand tr=5n tf=3n (i, o);
}
```

```
network latch (terminal vdd, vss, phi1, phi2, out, in)
{  net {phi2, phi2\_r, phi2\_l}; /* equivalent nodes */
      {inv[1..3]} invert (6, 9, 9, 7, 10, out);
      nenh w=8u l=4u (phi1, in, 6);
      nenh w=8u l=4u (phi2\_l, 6, 7);
      nenh w=8u l=4u (phi2\_r, 9, 10);
}
```

The last node name of the nand statement gives the output of the nand. In this example there is only one input node for the nand, so only one node precedes the output node. However, it is possible to use an arbitrarily number of input nodes for a function. For the nand a rise time of 5 nsec. and a fall time of 3 nsec. have been specified.
3.5.4 *shift_example*. Figure 4 shows the transistor diagram and the network description of a circuit which we will call *shift*.

![Transistor diagram and network description of the shift circuit](image)

```plaintext
network shift (terminal vdd, vss, phi1, phi2, in, out)
{
  nenh w=6u 1=4u (in, vss, 1);
  nenh w=6u 1=20u (vdd, 1, vdd);
  nenh w=8u 1=4u (phi1, 1, 2);
  cap 400f (2, gnd);
  nenh w=8u 1=4u (phi2, 2, 3);
  nenh w=6u 1=4u (3, vss, 4);
  ndep w=6u 1=16u (4, 4, vdd);
  res 20k (4, out);
  cap 150f (out, gnd);
}
```

**Figure 4.** Transistor diagram and network description of the shift circuit.

In the shift circuit, the first inverter has an enhancement transistor that is being used as a load. The inverted input signal is clocked by phi1 onto a large capacitance. When phi1 becomes low and phi2 high, the charge of the storage capacitance is shared between this capacitance and the gate capacitance of the second inverter stage. Because the gate capacitance is several times smaller than the storage capacitance, the gate will get the same state as the storage capacitance. The resistor and capacitor on the output represent wire resistance and capacitance.
3.5.5 \texttt{flipflop}\_example. Figure 5 shows the transistor diagram and the network description of a flipflop circuit. In this circuit the two nand circuit stages are cross coupled together by attaching the output of one nand to an input of the other.

\begin{figure}
\centering
\includegraphics[width=0.7\textwidth]{flipflop_diagram}
\caption{Transistor diagram and network description of the flipflop circuit.}
\end{figure}
4. SIMULATION AND SIMULATION COMMANDS

4.1 Simulation Control Commands

The simulation control commands must reside in a separate file (the command file). This file contains the signal descriptions of the network inputs and the commands which specify options and output format.

**TABLE 5. Syntax of the simulation control part.**

```
simulation_part = simulation_stmt ( ';' | newline )
   { simulation_stmt ( ';' | newline ) }.
simulation_stmt = set_cmd | print_cmd
   | option_cmd | empty.
set_cmd = set node_refs "=" signal_exp
   | set node_refs to node_refs from string.
signal_exp = value_exp { value_exp }.
value_exp = value [ "*", duration ].
value = h | i | z | f | "(" value_exp ").".
duration = integer | "=".
print_cmd = print node_refs.
option_cmd = option option { option }.
option = simperiod = integer
   | sigunit = f_float
   | outunit = power_ten
   | outacc = power_ten
   | level = integer
   | process = string
   | tdevmin = f_float
   | tdevmax = f_float
   | step = [ on | off ]
   | print races = [ on | off ]
   | maxdepth = integer
   | only changes = [ on | off ]
   | print info = [ on | off ]
   | maxpagewidth = integer.
   | vh = f_float
   | vminh = f_float
   | vmaxl = f_float
   | maxvicin = integer
   | maxvicin = integer.
node_refs = ref_item { ref_item }.
ref_item = full_node_ref | ".",.
full_node_ref = { inst_name "." } node_ref.
inst_name = identifier [ "[" index_list "]" ].
node_ref = integer | identifier [ "[" index_list "]" ].
index_list = index { ",", index }.
index = integer | range.
```
The following terminals are the reserved keywords in the command file.

**TABLE 6. Reserved Keywords for Commands**

<table>
<thead>
<tr>
<th>changes</th>
<th>maxaccmaxin</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>off</td>
<td>sigunit</td>
</tr>
<tr>
<td>from</td>
<td>on</td>
<td>simperiod</td>
</tr>
<tr>
<td>h</td>
<td>only</td>
<td>step</td>
</tr>
<tr>
<td>info</td>
<td>option</td>
<td>tdevmax</td>
</tr>
<tr>
<td>l</td>
<td>outacc</td>
<td>tdevmin</td>
</tr>
<tr>
<td>level</td>
<td>outunit</td>
<td>to</td>
</tr>
<tr>
<td>maxdepth</td>
<td>print</td>
<td>uh</td>
</tr>
<tr>
<td>maxnvicin</td>
<td>process</td>
<td>umin</td>
</tr>
<tr>
<td>maxpagewidth</td>
<td>races</td>
<td>umax1</td>
</tr>
</tbody>
</table>

For an explanation of the rest of the terminals (e.g. newline, f_float, etc.) the reader is referred to the network syntax description part in this manual.

An example of node_refs is:

```plaintext
cin 4, counter1.latch_a.112
total.submod[0..7,3].in[1..4]
```

The explanation to the simulation commands is given in the next paragraphs.

### 4.1.1 set_command

The set command describes the pulse signals applied to the network through the input terminals. A node with a signal attached, is by definition an input node. A signal is described by a signal expression, which consists of a list of value expressions. A value expression specifies the level (i.e. \( l = 0 \), \( h = 1 \) and \( x = \) undefined), the absence of a forced signal (i.e. \( f = \) free state) or a repetitive value expression, all with an optional duration. The duration specifies how many times the value expression is added to the preceding value expressions. By absence of the duration the default 1 is assumed. The values \( l \), \( h \), \( x \) or \( f \) are pulse functions with width is 1. So for the following set command:

```plaintext
set node1 = 1*2 h*-
```

the process of generation is shown in figure 6.
A tilde (-) expresses infinite duration. When no infinite duration is specified in the signal expression, after the duration of the signal expression the signal value of the node remains its last value. When more than one set command is given for a particular node, the different signal expressions are concatenated.

The other way to apply a signal to a node is by using a signal description present in a .res file and which is a result of a previous simulation. Then the first group of nodes is followed by another group of nodes and a file specification. The second group of nodes must be nodes which were printed when the .res file denoted by file_spec (specified without the extension .res) was made. The number of nodes in the first group must be equal to the number of nodes in the second group because the node to node assignment is done in pairs. For example, to set the signal descriptions of nodes in[1..8] equal to the simulation results of nodes out[1..8] which are in file counter.res:

```
set in[1..8] to out[1..8] from "counter"
```
4.1.2 print_command. The print command specifies nodes that must be printed out. The sequence of the nodes printed out is the same as specified in the print command. When a comma is specified in the print command an empty column is printed on that place in the output. In this way simulation output values can be printed more clearly. The output will be printed in a file named to the network and tagged with the extension ".out". If the network name is longer than ten characters it will be truncated to ten characters.

4.1.3 option_command.

4.1.3.1 simperiod. Specifies the end time (expressed in sigunit) of the simulation time interval. The start of the simulation is always at \( t = 0 \). When simperiod is not specified, the simulation will stop when the network has stabilized after the last input change.

(default simperiod = endless)

4.1.3.2 sigunit. Specifies the unit of signal duration (in sec.) in the set command and the unit of simperiod.

(default sigunit = 1)

4.1.3.3 outunit. Specifies the unit of time that will be used in the simulation output.

(default outunit = sigunit)

4.1.3.4 outacc. Specifies the unit of the least significant decimal of the time in the simulation output. It must hold that outacc <= outunit.

(default outacc = outunit)

4.1.3.5 level. Specifies the level of simulation. There are three simulation levels:

level 1
The simulator uses abstractions for transistors and nodes to calculate logic states. The conduction of nenhancement and penhancement type transistors is always considered then to be equal and much larger than the conduction of depletion transistors, and the capacitances of all nodes are considered to be equal. A logic high or low state is only assigned when the conduction path to a high or low input has a much larger conduction than the paths to
other input nodes, or - with charge sharing - only when all nodes have the same logic state. Else the undefined state is assigned.

level 2
With level 2, logic state determination is based on actual parameters of transistors and interconnections. In this way the simulator can deal with charge sharing between nodes with different capacitances and states, and with wide and narrow transistors in series, to calculate logic states. For each node the simulator determines an analogue voltage by resistance division or charge sharing. Resistance division is done by searching for a minimum resistance path $h_r$ to input nodes with the high state, and a minimum resistance path $l_r$ to input nodes with the low state, and then finding the node voltage by $v = \frac{h_r}{h_r + l_r}$. Charge sharing is done by dividing the total charge of a group of nodes by the total capacity of the group. From the analogue node voltages logic states are derived by a minimum voltage for the high state ($v_{minh}$) and a maximum voltage for the low state ($v_{maxl}$). When simulation is done at level 2, dimensions of transistors must be present in the network and process information is used.

level 3
The simulator can also be used to simulate the timing of a network. Based on the parameters of the transistors and the interconnections (which must be represented by resistors and capacitances in the network file), the times of the logic state transitions are determined by means of RC constants. Internally, the simulator uses linear voltage waveforms to approximate the node voltages. Logic state determination at level 3 is done according to level 2, and at the start of the simulation interval first a steady state of the network is determined according to level 2. To simulate with timing, transistor dimensions must be present in the network file and process information is used. Only at level 3 the rise and fall delays of the function elements (nand's, nor's etc.) are used during simulation.

(default level = 1)

4.1.3.6 process. Specifies the process file which contains the process information (e.g. transistor model capacitances and resistances) which is used when simulation is done at level 2 or 3. First the working directory is searched for this file. When it is not found there, an ICD library directory is searched.

(default process = "nmos4u700A")

4.1.3.7 tdevmin,tdevmax. By specifying minimum and maximum timing deviations it is possible to do a min-max delay simulation at level 3. The delays of the logic state transitions are multiplied
by \texttt{tdevmin} and by \texttt{tdevmax}, to obtain a min-delay and a max-delay for each logic state transition. For example, when a node normally changes from 0 to 1 after 10 ns., a \texttt{tdevmin} = 0.6 and a \texttt{tdevmax} = 1.3 will cause that the node will change from the 0 to the X state after 6 ns., and to the 1 state after 13 ns. In this way it is possible to determine how the final logical state of the simulated network depends on timing deviations. Note: Strictly speaking the simulator speeds up and slows down voltage waveforms by \texttt{tdevmin} and \texttt{tdevmax}. Therefore the actual min and max delay may be slightly different from as found by multiplication by \texttt{tdevmin} and \texttt{tdevmax}.

\texttt{(default tdevmin = 1, tdevmax = 1)}

4.1.3.8 \texttt{step}. When \texttt{step} is on, the values of nodes that are specified by the \texttt{print} command, are printed immediately after a change has occurred in the value of at least one node. In this way the simulation of the network can be debugged by observing all logic state transitions occurring on the printed nodes. It must be realized here that even during simulation without timing, the simulator always performs simulation steps to update logic states. When \texttt{step} is off, the values of the output nodes are printed only after each new stabilization of the printed nodes at a particular time.

\texttt{(default step = off)}

4.1.3.9 \texttt{print_races}. When feedback loops are in the network (e.g. flipflops), there sometimes may occur zero-delay oscillations during the simulation because of undecided races. The simulator will automatically put the relevant nodes in the undefined state. Print races is on will cause the simulator to give a list at the end of the simulation output, which shows where and on which time these races appeared.

\texttt{(default print races = on)}

4.1.3.10 \texttt{maxdepth}. With the maximum logic depth of the network (\texttt{maxdepth}), the simulator decides when an oscillation because of undecided races is occurring (see option \texttt{print races}). When \texttt{maxdepth} is specified too small the simulator will put nodes in the undefined state which shouldn't, and when it is much too large it will cause unnecessary long simulation times when oscillations occur. However, the simulator detects that the maximum logic depth is always less than the number of nodes in the circuit.

\texttt{(default maxdepth = 100)}
4.1.3.11 only_changes. By using only changes is on, in the output file only the logical values which really change at that moment will be printed. For an output signal which doesn't change a "," will be printed. In this way it becomes more clear which signals are changing at a particular moment and this can especially be more convenient for timing simulations.

(default only changes = off)

4.1.3.12 print_info. Usually the only extra information that is given in the simulation output is the number of nodes in the network. When print info is on also the number of transistors, resistors, capacitors, function elements and logical events during simulation, are printed in the output file.

(default print info = off)

4.1.3.13 maxpagewidth. Specifies the number of characters on a line in the output file. However, when maxpagewidth ≥ 80 and all printed nodes will still fit on one line of 80 characters, the number of characters on a line in the output file will be 80.

(default maxpagewidth = 132)

4.1.3.14 vh,vminh,vmaxl. Simulation with level 2 or 3 requires a minimum stable voltage (vminh) for the high state, a maximum stable voltage (vmaxl) for the low state, and a voltage vh for input nodes with the high state (vl = 0 always). When a stable voltage of a node is calculated to be between vminh and vmaxl, an undefined logic state will be assigned to the node. Usually the variables vh, vminh and vmaxl are read from the process file. But it is also possible to overwrite these values by specifying them in the command file. Enlarging vmaxl for example, causes that nodes will be set to the low state instead of the undefined state more often. It must hold that vmaxl < vh/2 < vminh.

(default vh, vminh, vmaxl are read from the process file)

4.1.3.15 maxvicin,maxtvicin. Switch-level simulation consists of evaluations of vicinities of nodes. To use computer storage space in an efficient way, the simulator assumes that there is a maximum number of nodes (maxvicin) and a maximum number of transistors (maxtvicin) in a vicinity. Most times the default values of maxvicin and maxtvicin will be sufficient. However, when the default maxvicin and maxtvicin appear to be not sufficient (and some set commands are not forgotten !), maxvicin and maxtvicin can be enlarged.
4.2 Examples

4.2.1 latch_example. The simulation control file of the latch circuit is as follows:

```c
/* latch simulation commands */
set in  = (h*4 l*4)*2
set phi1 = (h*1 l*1)*-
set phi2 = (l*1 h*1)*-
set vdd = h*-
set vss = l*-
option simperiod = 10
print vdd vss phi1 phi2 in out
```

The set commands result in the following input pulse sequences:

```
in
phi1
phi2
```

The signals vdd and vss have a constant value during the simulation. The simulation period is specified by option simperiod.

The node values of vdd, vss, phi1, phi2, in, out will printed in the file "latch.out" as follows:
4.2.2 *shift_example*. The following command file has been used to simulate the shift circuit:

```plaintext
set vss = 1*
set vdd = h*
set phi1 = (h*200 l*200)*2
set phi2 = (l*200 h*200)*2
set in  = l*400 h*400
set 3  = l*1 f*
option sunit = ln
option outacc = 10p
option level = 3
print phi1 phi2 in, 2 3 out
```

Because two enhancements are being used in series, and charge sharing is occurring on some places, the shift circuit has to be simulated with level 2 or 3. To illustrate timing simulation, level 3 has been chosen here. We have to choose a proper input unit and output accuracy with the options sunit and outacc in order to see the delays in the simulation output. Node 3 is initialized in the low state in this command file.
The output file "shift.out" will be as follows:

<table>
<thead>
<tr>
<th>time</th>
<th>p</th>
<th>p</th>
<th>i</th>
<th>2</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>in 1e-009 sec</td>
<td>h</td>
<td>h</td>
<td>n</td>
<td>u</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>0.00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>200.00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>200.21</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>205.43</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>400.00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>412.80</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>600.00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>600.21</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>618.84</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

network: shift
nodes: 10

4.2.3 flipflop_example1. With the flipflop circuit a race condition is possible. When in[1] and in[2] both change state from 0 to 1 simultaneously, the voltage of both out[1] and out[2] will start falling. The transistors with their gate connected to out[1] and out[2] will start to turn off and the falling of the voltages of out[1] and out[2] will stop. In practice however a race condition exists because the voltage of one output will fall somewhat faster than the voltage of the other output and the flipflop will be driven in a stable state where the fastest output is low and the other output is high. The flipflop circuit is first simulated without timing, the following command file is used:

```
set vss = l*-
set vdd = h*-
set in[1..2] = l*100 h*100
option sigunit = 1n
option outacc = 10p
option level = 2
option step = on
print in[1..2], out[1..2]
```

2.137
The output file "flipflop.out" will be as follows:

```
SLS
version: 2.0
SIMULATION RESULTS

time   | t t
in 1e-009 sec | n n
| * *
| 1 2 * *
| * * 1 2

0.00 | 0 0 x x
0.00 | 0 0 1 1
100.00 | 1 1 1 1
100.00 | 1 1 0 0
100.00 | 1 1 1 1
100.00 | 1 1 0 0
100.00 | 1 1 1 1
100.00 | 1 1 x x

network: flipflop nodes: 7
```

Because option step is on has been used in the command file, we see that the state of each node is printed after each simulation step. Because simulation is done without timing, out[1] and out[2] will change state to 0 at t = 100 simultaneously. The transistors will be turned off then, node out[1] and out[2] will become 1 again, and the circuit will start oscillating because of this undecided race. After a number of simulations steps equal to the logic depth of the circuit (which the simulator assumes to be equal to the number of nodes in the circuit) has occurred at t = 100, the simulator automatically puts the oscillating nodes in the X state.
4.2.4 flipflop_example2. A second simulation of the flipflop circuit is done with timing, the command file is as follows:

```
set vss = 1
set vdd = h
set in[1..2] = 1*100 h*100
option sigunit = 1n
option outacc = 10p
option level = 3
option step = on
print in[1..2], out[1..2]
```

The output file "flipflop.out" will be as follows:

```
S L S
version: 2.0
S I M U L A T I O N  R E S U L T S

time | i i   o o
in 1e-009 sec | n n   u u
         | * *   t t
         | 1 2   **
         | * *   1 2

```

```
0.00   0 0   x x
0.00   0 0   1 1
100.00 1 1   1 1
108.53 1 1   0 1
```

```
network: flipflop
nodes: 7
```

Because now capacitances are used to calculate times of logic state transitions, and the capacitance of node out[1] is less than the capacitance of node out[2], node out[1] will 'win the race' and become 0.
4.2.5 \textit{flipflop} \textit{example3}. Finally a min-max delay simulation of the \textit{flipflop} circuit is done. The min-max delay has been specified by means of \texttt{tdevmin} and \texttt{tdevmax}. Node out[1] will now change from \texttt{I} to \texttt{O} via the \texttt{X} state. During the interval that out[1] is \texttt{X}, out[2] also becomes \texttt{X}, and the \texttt{X} state will result as the stable state for out[1] and out[2]. In this way the simulator indicates that the final logic states depend on the timing, and that wrong logic states can result when circuit parameters have 50\% deviations.

The following command file was used for simulation:

\begin{verbatim}
set vss = 1*
set vdd = h*
set in[1..2] = 1*100 h*100
option sigunit = 1n
option outacc = 10p
option level = 3
option tdevmin = 0.5 tdevmax = 2
option step = on
print in[1..2], out[1..2]
\end{verbatim}

The output file "\textit{flipflop.out}" will be as follows:

\begin{verbatim}
SLS
SIMULATION RESULTS

<table>
<thead>
<tr>
<th>time</th>
<th>in 1e-009 sec</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0 0</td>
<td>x x</td>
</tr>
<tr>
<td>0.00</td>
<td>0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>100.00</td>
<td>1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>104.26</td>
<td>1 1 x 1</td>
<td></td>
</tr>
<tr>
<td>104.75</td>
<td>1 1 x x</td>
<td></td>
</tr>
</tbody>
</table>

network: flipflop

\end{verbatim}
The random counter circuit as described in the paper about the layout to extractor in this book, can be simulated with the SLS simulator. A simulation at level 1 and a simulation at level 2 of the random counter circuit will be given here as an example. The simulation at level 1 required 8 sec. of CPU time on a HP-9000 series 500 computer, and the simulation at level 3 required 14 sec. For both simulations the command file and output file are listed.

Simulation at level 1 (logic):

```
set vss_r1 vss_r3 = 1*
set vdd_r3 vdd_r2 vdd_r1 = h*
set phi1_r = (h*1 l*1)*
set phi2_r = (l*1 h*1)*
set p_ld_r = h*1 l*
set run_r = l*1 h*
set in[0..7] = h*
option simperiod = 32
option level = 1
print phi1_r phi2_r run_r p_ld_r, in[0..7],
print out[0..7], fb_in

SLS Simulation Example
Random Counter
```

---

**SL S**

**V**ersion: 2.0

**S I M U L A T I O N **

**R E S U L T S**

| Time | p | p | r | i | i | i | i | i | i | i | i | i | i | i | o | o | o | o | o | o | o | o | o | o | f |
| 0 s  | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h | h |
| 1 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 3 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 4 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 5 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 7 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 8 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 9 s  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 10 s | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 11 s | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 12 s | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

2.141
Simulation at level 3 (logic and timing):

```
set vss_r1 vss_r3 = l*-
set vdd_r3 vdd_r2 vdd_r1 = h*-
set phi1_r = (h*1 l*1)*-
set phi2_r = (l*1 h*1)*-
set p ld_r = h*1 l*-
set run_r = l*1 h*-
set in [0..7] = h*-
option simperiod = 32
option level = 3
option process = "nmos4u700A"
option sigunit = 1u
option outunit = 1n
option outacc = 10p
print phi1_r phi2_r run_r p ld_r, in [0..7],
print out [0..7], fb_in
```
## SLS Simulation Example

### Random Counter

<table>
<thead>
<tr>
<th>Time (1e-009 sec)</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>1.00</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>2.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>3.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>4.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>5.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>6.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>7.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>8.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>9.00</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>10.00</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

*Simulation version: 2.0*
<table>
<thead>
<tr>
<th>SLS Simulation Example</th>
<th>Random Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>11014.21 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>11024.40 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>11025.43 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>12000.00 I 1 0 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>13000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>13005.83 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>13005.97 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>13006.04 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>13012.15 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>14000.00 I 1 0 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>15000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>15005.97 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>15012.15 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>15045.85 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>16000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>17000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>17055.83 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1706.04 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>17013.52 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>18000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>19000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>19005.83 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>19012.15 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>19030.55 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>20000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>21000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>21005.97 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>21014.21 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>21018.68 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>22000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>23000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>23012.15 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>23013.52 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>23024.40 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>24000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25005.83 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25005.97 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25010.34 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25013.52 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25024.40 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25040.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25049.06 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>25049.06 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>26000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>27000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>27012.15 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>28000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>28000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>29000.00 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>29005.83 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>29035.87 I 0 1 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
### SLS Simulation Example

**Random Counter**

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
<th>Data</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30000.00</td>
<td>1 0 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>31000.00</td>
<td>0 1 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>31012.15</td>
<td>0 1 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>31013.52</td>
<td>0 1 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>31024.40</td>
<td>0 1 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32000.00</td>
<td>1 0 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Network:** rand_cnt  
**Nodes:** 110
References


Chapter 3

Discrete Synthesis Tools
AUTOMATIC ROUTING IN A GENERAL CELL ENVIRONMENT

H. Cai and P. Dewilde
Department of Electrical Engineering
Delft University of Technology
The Netherlands

Abstract

A complete, highly automatic and fast routing system intended for general cell VLSI circuits, is presented. Routing is divided in a two phase scheme: global routing and detailed routing. Detailed routing is based on dividing the routing area into routing channels and using a channel router repeatedly to realize the interconnections in each channel. Algorithms on channel structure construction, channel ordering, global routing, planar power supply net routing, optimal relative block positioning and detailed routing on two or three layers are outlined. The channel congestion problem is handled in both the global routing phase and the detailed routing phase. Blocks and channels are not restricted to rectangular shapes. Placement of the blocks can be adjusted by the routing system guaranteeing 100 percent completion of routing. A prototype implementation of the routing system is being tested with promising results. The sequence of functions is mutually consistent. Many of them contain new algorithms.

1. Introduction

The increasing integration density and scale of VLSI circuits necessitates the use of sophisticated automation tools to speed up the layout design. Routing the interconnections is a very time consuming part of the chip design process. Very successful automatic routing algorithms have been developed for the standard cell concept. But, because of increased chip complexity and the availability of predesigned macro cells, (we call them blocks), of varying sizes --RAM,ROM,ALU,REG--, of handcrafted custom cells and of collections of standard cells, the standard cell concept alone is not enough for advanced VLSI circuits. A routing system in general cell environment becomes an indispensable part of every custom IC layout system.

The routing problem can briefly be stated as follows. For a given placement of the blocks with a given orientation and power consumption of each block, and given a set of interconnection nets, each net is defined by a set of block terminals, the task is to perform the required interconnections in as small a rectangular area as possible [1]. In our system the local routing will be done on blocks that are not restricted to rectangular shapes. They can have any rectilinear shape of arbitrary size. An example of a block is given in Fig. 1. The user should specify the four sides (top, left, bottom and right) around the contour of the block, see broken lines in Fig. 1. Each side consists of a set of polygon edges. Terminals can only be positioned on edges which have proper direction in relation to their side, namely horizontal for top and bottom edges and vertical for left and right edges. In Fig. 1

This paper was produced as part of Task I of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.

3.1
terminals can be positioned on the edges drawn in bold lines. As an example, circuit blocks and their placement are shown in Fig.2.

![Figure 1. An example of a block](image1)

We have developed a highly automatic, complete and fast routing system intended for general cell VLSI layouts, but also applicable for standard cell layouts. It incorporates a series of algorithms, such as channel structure construction, global routing, planar power supply net routing, optimal relative block positioning and detailed (channel) routing. Some basic ideas have already been reported in [2] and [3].

This paper will present the routing system in a more extended and consistent way paying attention to the integration of the system's components. Section 2 gives a short overview of the system. Section 3 provides the definition and construction algorithm for the channel structure. Section 4 presents the global routing algorithm. In section 5 a method is outlined for planar routing of the power supply and ground nets and variable wire width calculation of these nets. Section 6 explains the preparation work needed before routing each channel. In section 7 a new fast channel routing algorithm will be discussed. Section 8 presents an algorithm to determine the block relative position in order to minimize the channel densities and ensure easy routing of other channels which are adjacent with the current one. Section 9 concludes the paper by giving some implementation results and conclusions.

3.2
2. System overview and its properties with respect to others

The routing process is divided into a number of steps. Fig. 3 gives an overview of the system.

We start out from a slicing structure placement [4] of the blocks. The spaces between the blocks are divided into routing channels. The channel structure are constructed using a similar method as in [5]. Only 'T' type channel intersections occur. Because of the slicing type placement, the order of the routing channels is easily determined.

Figure 3. System overview
The global routing is carried out with the help of a channel intersection graph. The goal of global routing is to minimize the total wire length. Although this goal cannot be achieved with an algorithm of reasonable complexity, we are able to obtain good results with methods that perform single net optimization, yet taking into account of channel congestions, so as to include the influence of other nets. An efficient cost function is used to find the optimal path for each net. The cost function is a combination of the path length and channel congestions. The nets are routed sequentially in a certain order.

Power supply and ground nets are routed in a separate stage. Since these nets must be routed in one single metal layer with variable width, it is inefficient to do the power net routing in the same way as the signal net routing. This important issue is often ignored by many routing systems [6,7,8]. Up to this point the blocks are treated as rectangles, as a matter of fact a block is replaced by the smallest rectangle enclosing the block. Interactive programs are also available for the routing steps above.

Before routing each channel, a relative position of the two adjacent blocks along the channel is determined. The relative position of the blocks is chosen such that the channel density is minimized and easy routing of other channels which are adjacent with the current one is ensured. An efficient procedure by LaPaugh [9] is modified and used for the block positioning which is much more sophisticated than the procedure used in [6].

Although many known channel routers can be adopted [10,11,12] in our system, we have developed a fast grid-based greedy channel router, because of the lack of features we wanted, such as the routing of rectilinear shaped channels, three layer routing and position nets at the channel ends on the desired side of the channel (in order to reduce the routing congestion in the channel intersection area). A postprocessor is used to optimize the routing results.

After the detailed routing of each channel, the blocks adjacent to the channel are merged into a hierarchical block which participates in the further merging process. In the merging process no distinction is made between a basic block and a hierarchical block. The space between the blocks will be adjusted automatically by the channel width, so a 100 percent routing is always achieved and avoiding compaction after routing which is very time consuming for large problems [13].

The system developed is independent of the design rules and layout description syntax. Although three layers can be used in MOS layout for interconnections, in most applications routing is constrained to two layers. The metal layer is used in the channel length direction and polysilicon layer is used to cross under the metal layer. The third layer, diffusion, should be used only on a very limited scale. The connectivity is achieved by vias from one layer to the other. More details of the algorithms will be given in the next sections.
3. The construction of channels

Given a placement of blocks, the spaces between blocks are partitioned into a set of regions, called channels. It is within these channels that all signal nets, power and ground nets must be routed. The channels are represented by line segments as illustrated in Fig. 4. The width of a channel is only determined after the detailed routing. A basic requirement of channel routing algorithms is that the position of all terminals along the two sides of a channel must be fixed. A channel cannot be routed independently of the other channels, that is, in an arbitrary order. In the detailed routing, a processing order of channels has to be determined to obey the rule that a channel \( i \), which is perpendicular to another channel \( j \), is to be processed prior to channel \( j \). So we will assign to every channel a routing order. Cyclic channel precedence constraints is prevented by the slicing structure placement.

![Figure 4. A channel structure](image)

3.5
The channel construction and ordering algorithm:

Step 1:
Put the four outmost channels onto the channel stack, first the two vertical channels, then the two horizontal channels. Define the boundary of the block set using the four outmost channels.

Step 2:
Cut the block set into two subsets using a straight horizontal or vertical line. Put this line (channel) onto the channel stack. This new channel is used to define the boundaries of the two subsets.

Step 3:
If there is still a subset containing more than one block, Goto step 2.

Step 4:
Pop the channels from the channel stack, yielding the channel routing order.

Procedure I: Channel construction and ordering algorithm

An example is shown in Fig. 4. The routing order is V2, V3, H2, V4, H1, H3, V1 and V5.

4. Global routing

At the stage of global routing, for each signal net a general path through the channels is searched, so as to minimize the total wire length and the signal delays. The path is partitioned into a sequence of segments and the segments are assigned to channels. Each channel may be broken by the channel intersections into one or more subchannels. For the purpose of global routing a channel routing graph $G = (V, E)$ is constructed from the channel structure. Each vertex $v_i \in V$ represents a subchannel $s_i$ or a channel intersection point $c_i$ and undirected edges $E$ represent their adjacency. We have chosen this routing graph because it is also very suitable for our interactive version of the global router. An example of such a channel routing graph is shown in Fig. 5. In this way global routing problem is reduced to find a minimum cost path connecting a net on $G$.

The signal nets are routed sequentially according to the "easiest net first" order: let $N_i$ be the number of terminals in the net $i$, and $T_i$ be the total number of terminals positioned inside the smallest rectangle which covers all terminals in net $i$. Then, the net to be routed is selected in increasing order of $aN_i + bT_i$, where $a$ and $b$ are constants. The idea behind this ordering is that nets with many terminals and which interfere most with other nets are put off until the end, because they have the most alternative paths compared to the other nets.

The minimum cost path algorithm is based on Dijkstra's vertex expansion algorithm. When connecting a net, starting from the source vertex $S$ and with the goal of reaching the given target vertex $T$, the vertex which has the minimum cost to the source vertex will expand to its adjacent vertices. The procedure is
repeated until the target vertex is reached.

For a multiterminal net, the following procedure is employed:

**Procedure II: Global routing of a multiterminal net**

Step 1: The connected subnet is initialized at the gravity terminal, that is the terminal nearest to the gravity point of the terminals in the net.

Step 2: Find the terminal with shortest rectilinear distance to the gravity terminal and connect it by the minimum cost path algorithm to the connected subnet.

Step 3: Mark the path.

Step 4: If there is still an unconnected terminal

\[\text{Goto step 2;}\]

From the electrical point of view, we have chosen this procedure instead of finding a general minimum rectilinear Steiner tree, because we prefer a 'star' shaped path rather than a minimum length 'ring' shaped path, see Fig. 6. We observe that in a 'star' shaped path the signal delay between any two terminals is optimal, although the total wiring length might not be.

Furthermore, internally connected terminals can be handled very efficiently. These terminals often occur at data busses when busses run through a block. These internally connected terminals are grouped into a terminal group and the gravity point of the terminal group represents the group. An example of such a terminal net is given in Fig. 7.

To avoid channel congestions, especially in the center channels, we use furthermore a cost function which is a function of the net length and channel congestion. Let a path consists of a
Figure 6. Global routing of a multiterminal net

Figure 7. An example of an internally connected net.

set of subchannels \( S = \{s_1, s_2, \ldots, s_n\} \). The cost function \( C_s \) of the path is defined by:

\[
C_s = \sum_{i=0}^{n} (d_i + \alpha \cdot c_i^2)
\]

where \( d_i \) is the length of subchannel \( s_i \), \( c_i \) the congestion value of subchannel \( s_i \), and \( \alpha \) is a constant. The congestion value of a subchannel is initialized at 0 and increased by 1 each time a net is passing through this subchannel. This cost function tries to smooth the congested channels. When a subchannel is getting congested, it will exercise a great resistance to the coming nets because of the quadratic behavior of the congestion function.
5. Power supply net routing

Power supply (VDD) and ground (GND) nets are handled in a more specialized way than signal nets. Since power supply nets must be routed in one single layer with variable width, it is inefficient to handle power net routing in the same way and at the same time as signal net routing. In order to allow the power and ground nets to be routed on one single layer, we must insure that they do not cross each other. Other signals have to take a poly underpass each time they cross these nets. The VDD and GND nets are routed from opposite sides of the chip and form an interdigitated pattern. The net wires form a tree with its root at the pad and its leaves at the sides of the blocks. An example of the power net trees of a placement is shown in Fig. 8. In this approach only two sides of a block are reached by one of the power nets, the other two sides by the other net. This doesn't seem to be a big restriction, since predefined blocks normally have terminals of the same net on the opposite side of the block and they are usually internally connected.

![Figure 8. Power net routing](image)

Once the power net trees are constructed, design-rule-dependent wire widths are calculated by binary tree traversal, depending on the current requirements through each branch of the tree. The user gives each block's maximum current requirement. The maximum current through a wire segment is the sum of its children's maximum currents plus its own current. After every wire's maximum current is known, multiplying by a technology constant gives every wire's minimum width.

Note that because we do not know the channel widths yet at this stage, in contrast to [13], the generation of the actual layout of power net wires is postponed to the detailed routing step. By then the wire widths calculated in this step will be needed. The main advantage of this approach is that at the detailed signal net routing phase no consideration need to be made of already routed power wires and no compaction is needed after
routing caused by overestimated channel widths. The complexity of the power nets routing step is very low. How the power wires are laid out will be explained in the next section.

6. Preparation of the channel routing

Before the channel routing is to be applied to a channel, some important preparation work must be done. First of all, the relative position of the left and right side blocks along the channel has a great influence on the width of the current and its top and bottom adjacent channels. An optimal relative position of the blocks is determined. The detailed algorithm to determine this position is given in Section 3.

Next, the positions of the terminals on each side of the channel and the connection requirements to the top and bottom adjacent channels must be extracted from the global routing information. Since we have developed a grid-based channel router and we do not want to force the cell designers to lay out the terminals on an equidistant grid system, the terminals on both sides of the channel must first be routed on grid positions along the channel. This is done by using a simple river routing technique. Fig. 9 shows an example of a channel environment. Since only polysilicon terminals are allowed on the channel sides, the terminals on layers other than polysilicon will be changed to polysilicon layer by contact holes. We call this process terminal gridding and layer adjustment.

![Figure 9. Terminal gridding and layer adjustment](image)

Furthermore power supply wires have to be laid out separately along the left and right sides of the channel with variable width determined before. The area used by the river router can be shared by the power wires, because the power wires are on the metal layer, see Fig. 9. The power wires will closely follow the contour of the block sides. In vertical channels the VDD wire is always on the left side and GND wire on the right side. In horizontal channels the VDD wire is always on the bottom side and GND wire on the top side.
Additionally, it is also important to optimize the position of nets at the ends of the channel that is being routed in order to ease the routing at later stages, because routing each channel with minimum width individually does not always minimize the total routing area. The position of the nets which come out from the end of the base channel may complicate the routing of the cross-piece channel when two channels meet at a "T" type intersection. For example, in the case of the crossing shown in Fig. 10, the solution 10(a) needs two tracks in the top adjacent channel for routing the nets 1 and 2. However, in the case of Fig. 10(b) only one track is needed, which is shared among the two nets. Clearly, if possible, we should place outgoing nets to the same side of their direction in the adjacent channel. If the net goes to both directions in the adjacent channel, then the position of the net is not important. Thus according to the global routing information, we divide the outgoing nets into three groups: The "R" group consisting of nets going right, the "L" group for the left going nets and the "I" group for the rest. We will specify for each net its preferred side and pass it to the channel router. Since the router generates one track at a time alternating from both sides, it is easy to implement this net position preference requirement. Experience shows that a remarkable reduction of the congestion in the channel crossing area can be achieved by this technique.

![Figure 10. Channel intersection area](image)

Finally, channels are not restricted to a rectangular shape. A description of the rectilinear shaped channel envelope is generated for the channel router as follows: for a horizontal channel, at each grid position along the top and bottom sides of the channel a deviation value Dev(x), related to the top level of the envelope on that side must be specified, see Fig. 11. But the channel width is adjusted so as to realize the interconnection requirements. Consequently, a 100 percent routing is always guaranteed. Note that the class of channels is restricted to rectilinear channels with no terminals on the vertical edges of the envelope and to legally widen the channel there must exist at least one track that runs completely through the channel without hitting a vertical edge of the channel envelope.
We have implemented a new grid-based channel routing algorithm, where the grid distance is design-rule depended. It is based on an immediate placement of segments on a track in the channel, followed by a placement of segments on the next track at the opposite side of the channel until all nets are connected. This is so called track-by-track approach. Operations that take place on a track are controlled by the positions of unbound or multiple terminals. This algorithm has a number of advantages compared to the other channel routers. First of all, the simplicity of the algorithm leads to efficient implementation. There is no need for detection of routing constraints and unlimited depth of doglegging is done automatically without special dogleg detection. Furthermore, the algorithm has many useful abilities, such as, three layer routing, routing of rectilinear shaped channel and net position preference handling. Although optimization is only locally performed, on difficult examples results are very close to optimal. For simplicity and without affecting generality, henceforth we limit our discussion to rectangular horizontal channels. Polysilicon layer run in the vertical direction, metal and diffusion layer run in the horizontal direction.

The algorithm presented is locally optimal for routing two or three layers. Basically, it considers one track at a time and it provides a maximum density on this track. During the routing of each new track, results of the former tracks are used as channel boundary. The only operations allowed on a track is a connection between multiple terminals, leaving at least one grid position free at the next track, or a connection towards a free position, leaving the original grid position free. These operations are the base of the algorithm, see Fig. 12, where 0 indicates free positions.

We shall distinguish two types of operations, one that controls the operations allowed on the free positions, and one that concerns the multiple terminals at the same side, including nets going out at the left and right channel ends. For both operations two modes exist; one we call "efficient steps", and the other we call "inefficient steps". Efficient steps are steps which do not cause the use of an extra track. Fig. 13 gives some examples.
Figure 12. Basic operations of the channel router

Figure 13. Efficient and inefficient routing steps

For multiple terminals the grouping at one side is efficient when no terminal occurs at the other side (Fig. 13(a)), or when all (remaining) terminals at both sides can be connected together (Fig. 13(b)). In Fig. 13(c) the grouping in an early stage may cause the use of an extra track which is inefficient. For the other type of operations efficiency is shown in Fig. 14. Compared with other algorithms in Fig. 14(a) a dogleg situation occurs while the situation in 14(b) can be compared with a constraint solution. The situation in 14(c) is efficient if the terminal net opposite the free position can be connected at once. If not this operation will be inefficient, 14(d). A similar definition exists for the operation that can be taken on a free column. If the terminals occur at either side and if the terminal at one side is situated on the left side of the free column, and the terminal on the other side on the right side of the free column then the connection will be efficient (Fig. 14(a)). Otherwise it will be inefficient (Fig. 14(b)).

During the routing of each track first all efficient connections are considered. A local maximum is searched to determine which connections will be made. Hereafter all remaining (still possible) inefficient connections are considered. The algorithm starts at the top side, and alternates between the top and bottom sides. After each track generation the boundary of the channel is propagated one track towards the channel center. In each step a new track is considered using the results of the last two tracks generated as new terminal lists. Consideration is given to the net position preferences.

In the three layer routing mode the algorithm examines a track two twice, the first time for the metal layer, and the second time for the diffusion layer with the two restrictions that
Figure 14. Routing efficiency examples

(i) nets going out to the left and right channel ends are forbidden and (ii) polysilicon to diffusion vias cannot be made on the places where already a via has been made to metal layer. If the channel is not rectangular a track is divided into several subtracks according to the channel envelope.

After the channel is completed, a postprocessor is invoked. The postprocessor scans the channel from left to right, column-by-column. It eliminates the redundant via holes and changes vertical polysilicon wire segments not crossed by horizontal metal segments of other nets into the metal layer, see Fig. 15. The results of the channel router are promising. It routes the famous "Deutsch difficult example" in the best solution known, namely, in 19 tracks using two interconnection layers, see Fig. 16 and in 11 tracks using three interconnection layers.

Figure 15. Channel routing postprocessing
8. Relative position of blocks

In order to minimize the chip area, it is necessary to realize the routing with minimum width for each channel. Although most known channel routers achieve channel width that are usually within the channel density (i.e. the maximum number of wires that intersect any vertical cross section of the channel), the problem of minimizing the channel density by shifting the two blocks laterally along the two channel sides has escaped attention in the case of non-rectangular channels (for the rectangular case we found one publication [9]). Also, one should bear in mind that there are strong interactions between the adjacent channels. So we cannot just shift the blocks to achieve minimum density for the current channel, but we also have to take into consideration the top and bottom adjacent channels. From our experience we found that planning the channel environment to produce the globally optimal routing is as important as optimizing each channel's routing.

In our system the blocks in the placement are floating, in the sense that only the relative relation is fixed between them. Obviously, the relative position of the blocks along a channel sides has a great influence on the width of the current and the adjacent top and bottom channels. In this section, we consider the problem of determining a relative position of the blocks (they can also be hierarchical blocks) along each channel such that the total area of the layout is minimum. The layout area is composed of the blocks themselves and the area needed for interconnection of the terminals on these blocks. The interconnection is made by using horizontal and vertical tracks in the channels. Since global optimization of the layout area is a much harder problem, we solve it locally by optimizing the relative position of the two blocks along each channel just before processing that channel. We aim at finding a relative position \( d \) (offset) that minimizes the area of the Minimum Enclosing Rectangle (MER). MER is defined as the smallest rectangle enclosing both blocks and the top and bottom tracks of "R" and "L" nets from the current channel, see the dotted rectangle in Fig. 17.

Figure 16. The Deutsch difficult example
Thus, let $W_d$ and $L_d$ be the width and length of MER at offset $d$ respectively. The optimization algorithm is shown below:

Minimize the MER area procedure:

```plaintext
S_{MINMER} = MAX_INTEGER;
for (offset d from $d_{min}$ to $d_{max}$)
{
    calculate $W_d$ at offset $d$;
    calculate $L_d$ at offset $d$;
    if ($W_d \times L_d < S_{MINMER}$)
    {
        $S_{MINMER} = W_d \times L_d$;
        optimal_offset = $d$;
    }
}
```

Procedure III: minimize the MER area.

8.1 Calculating the MER width

The width of MER is the sum of the widths of the two blocks and the channel between them. The widths of the blocks are fixed. The width of the channel, however, depends on the relative positions of the blocks. The width of a block is defined as the width of the smallest rectangle enclosing the block. The width of the channel is defined as the width of the biggest rectangle fitting inside the rectilinear channel envelope, see Fig. 17. We do not know the channel width exactly before the channel routing has been carried out, however the minimum achievable width of a channel is $3.16$. 

3.16
fairly accurately measured by the channel density. We will first describe an exhaustive algorithm to compute the channel density for all offsets and then a faster algorithm using an idea by LaPaugh \(9\).

The density of the channel depends on the relative lateral position of the channel's sides, given the positions and net numbers of terminals along the top and bottom side of a channel and a list of nets going out from the left and right ends of the channel. This relative lateral position is the channel's offset. Note that we consider the case of horizontal channel in this section. An example of a channel is shown in Fig. 11.

Formally, the channel density \(T_d\) at offset \(d\) for a rectangular channel is defined as:

\[
T_d = \max_x \sum_{i=1}^{n} d_i(x,d)
\]

where

\[
d_i(x,d) = \begin{cases} 
1 & \text{if } l_i(d) \leq x \leq r_i(d) \text{ and } l_i(d) \neq r_i(d) \\
0 & \text{otherwise}
\end{cases}
\]

and \(l_i(d)\) is the leftmost terminal of net \(i\) when the offset is \(d\), \(r_i(d)\) is the rightmost such terminal. \(d_i(x,d)\) is the contribution of net \(i\) to the density at \(x\) when the offset is \(d\). Due to the presence of the "bay" area in the rectilinear channels, the number of tracks can be reduced at each \(x\) by using this "bay" area for routing, see Fig. 11. The density of a rectilinear shaped channel is defined as the density of the biggest rectangular area fitting inside the channel.

As input data we have four lists, \(T\), \(B\), \(R\), and \(L\) and the channel envelope description. \(T\) contains \(x\) coordinates and net numbers of terminals along the top side sorted from left to right. \(B\) contains the same information for bottom terminals. \(R\) and \(L\) are two lists containing net numbers going out to the right and the left channel ends. The positions of terminals is given relative to the leftmost terminal on the same side. The absolute position of a bottom terminal is equal to its relative position; the absolute position of a top terminal is obtained by adding the offset to its relative position. The offset is the position of the leftmost top terminal with respect to the leftmost bottom terminal, with distance to the left being negative.

The algorithm will compute the density at each of the \(w\) relevant positions where \(w\) is the sum of the length of the top and bottom channel sides. For each offset, the density is calculated by scanning the channel from left to right. The density is initialized to the number of nets in the \(L\) list. The incremental change of the density at each \(x\) is calculated during the sweep across the channel and the maximum is remembered.

While scanning the channel to compute the density, the algorithm distinguishes the positions of terminals: whether it is at the beginning, in the middle, or at the end of a net. The nets in the \(L\) list are considered as net starting positions; the nets in
the R list are considered as net ending positions. In processing a terminal the following cases may occur: if the terminal is singular at the given absolute position, then the density decreases by one if the net is ending at it and increases by one if the net is beginning, otherwise the density will be unchanged. If there is a second terminal at the same absolute position on the opposite side and they have different net numbers then they are processed separately like in the previous case. If they belong to the same net, then if there is no other terminal connected to the same net, the density is unchanged, else they are treated as in the case of one terminal.

Furthermore at positions where a change in one or the other deviation values occurs in the channel envelope, the density will be changed accordingly. Our channel router takes advantage of these "bay" areas along the channel to reduce the channel width. If the difference of the deviation value between two neighboring grids is $k$ and if $k > 1$ then the density should not be increased by $k$ directly. The change should be anticipated $k$ steps ahead, by increasing the channel density with steps of one unit. Fig. 18 shows an example of such a situation. In this way we can consider these grids as pseudo terminals having unique net numbers on the positions introduced. Such a terminal is a beginning terminal if it corresponds to an uphill deviation change, i.e. the channel is getting narrower. As a result the density will be increased by one. It is considered an ending terminal if it corresponds to a downhill deviation change, i.e. the channel is getting wider, in which case the resulting density is decreased by one. These terminals are then merged in the T and B lists. The scanning of the channel takes $O(n)$ operations, where $n$ is the number of nets, including the pseudo nets explained above. Since there are $w$ relevant offsets, the overall complexity of the algorithm is $O(wn)$.

Figure 18. Channel bay area

Since the previous algorithm depends on the channel length, it can be inefficient when the terminals along the sides are sparse. Therefore we present an algorithm which is channel length independent. The idea is originally proposed by LaPaugh [9] for rectangular channels.
It is based on the following observations. First, the density calculation gives exactly the same results from one offset to the next except when top and bottom terminals become aligned. Therefore, at most $2n^2$ offsets may yield different densities. The densities need to be updated only for these offsets. The densities of the other offsets are equal to the density of their previous offset. The algorithm will process the alignment of the terminals and calculate the value of the next offset.

The second observation concerns the updating of the local density by the previous exhaustive algorithm which scans across a channel. Those terminals which are not currently aligned with another terminal or were not aligned at the last offset of the algorithm undergo the same update for the new offset as for the last, so the last change of density can be used again for the new offset. Therefore the only changes from one offset to the next occur in the vicinity of aligned terminals. The new algorithm will keep track of changes of densities at each position from one offset to the next.

The complexity of this algorithm has been proven to be $O(n^2 \log n)$ [9]. An example of a channel with different offsets which yields different channel widths is shown in Fig 19.

![Channel example with different offsets](image_url)

**Figure 19.** Channel example with different offsets, (a) offset = 0, density = 11, (b) offset = 4, density = 7
8.2 Calculating the MER height

The channel length is defined as the distance between the leftmost terminal and rightmost terminal along the channel sides. Taking into account the block heights and the number of tracks above and below the blocks from the R and L groups at both ends of the current channel (as shown in Fig. 17), we can calculate the height of the MER easily at each offset by measuring the block heights extended by the "R" and "L" tracks. Although this does not account for the precise shape of the adjacent channels, we obtain a good estimation of the height very fast and still achieve a reasonable result.

8.3 Local optimum versus global optimum

It is hard to tell exactly how the local optimization affects the global optimal solution. From experiments one sees that this procedure generally improves the global solution. The algorithm has the tendency to reduce the dead area caused by the different block sizes and to produce hierarchical blocks that deviated less from a rectangle. This also makes the channel routing easier.

9. Concluding remarks

The routing system constructed on the basis of the algorithms described in this paper has been implemented in the programming language C, and has been operating on a SEL_GOULD computer and a HP9000 computer under the UNIX operating system. Fig. 20 (or the color plate) and Fig. 21 show two examples of routed chips. The statistics for these two examples are given in the table in Fig. 22. The result obtained by the routing system is as good as the hand packed layout.
Figure 20. Routing result example: chip1
Figure 21. Routing result example: chip2 (Courtesy: Philips Telecommunication Institute)

<table>
<thead>
<tr>
<th></th>
<th>Chip1</th>
<th>Chip2</th>
</tr>
</thead>
<tbody>
<tr>
<td># blocks</td>
<td>13</td>
<td>18</td>
</tr>
<tr>
<td># terminals</td>
<td>140</td>
<td>984</td>
</tr>
<tr>
<td># terminals/block</td>
<td>10.7</td>
<td>54.6</td>
</tr>
<tr>
<td># nets</td>
<td>40</td>
<td>376</td>
</tr>
<tr>
<td># terminals/net</td>
<td>3.5</td>
<td>2.6</td>
</tr>
<tr>
<td>chip width(um)</td>
<td>2499</td>
<td>6844</td>
</tr>
<tr>
<td>chip length(um)</td>
<td>2477</td>
<td>6294</td>
</tr>
<tr>
<td>CPU time(GOULD)</td>
<td>57 sec.</td>
<td>16 min.</td>
</tr>
</tbody>
</table>

Figure 22. Two layers routing results statistics

Although the system is not yet perfect it has been proven to be a powerful tool to speed up drastically the design process of...
VLSI chips. Especially, when it is used in combination with our interactive placement and routing programs, fast feedbacks are made possible and some human intelligence can also be added into the solutions. This system can also be used for two metal layer process with polysilicon as the third layer. The system is hierarchical in the sense that after the routing is completed, the overall dimensions of the chip, including bonding pads positions, are known. This enables the inclusion of this chip as a "sub-chip" element when the system is called for placement and routing at the next higher level of hierarchy.

References

Parametrised building blocks.

G. Kleissen, M. Roelofs, Chr. van Veenendaal,

ICD, Enschede (The Netherlands).

and

L. Spaanenburg,

Twente University of Technology,
Enschede (The Netherlands).

Abstract. The module generator MoDG is presented. It facilitates to generate any sliced layout from a cell library and a floorplan. MoDG is part of the VLSI SPIRIT, in which it coorperates with a block assembler to construct VLSI in an hierarchical fashion.

1. Introduction.

Assembly and compilation are two widely differing, yet isomorphous design activities. Both reflect the influence of computer science (in terminology and in philosophy), but are actually part of a different approach to designing. This can be pictured by Gajski's Y-chart. The three axis represent the three views on a circuit under construction. A typical design sequence follows a cardiodic route. In the top-down design strategy, the travel starts with an abstract function for the external behaviour. Then the function is divided into parts, which reveals the structural implication of modeling. The next step to the geometry axis defines the floorplan. In subsequent phases the behaviour, structure and geometry of subparts and parts thereof are defined. In the bottom-up strategy, the same path as above is taken, but in the opposite direction. First the smallest functions are put together, then larger assemblies are made and so on. The key word here is assembly, while in the top-down strategy one has to go for generation. And the tool for generation is the compiler.

This paper was produced as part of Task V of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
2. Silicon compilation

Depending on which side the Y-chart is entered, a silicon compiler can be functional or structural (see fig 1) [3]. The functional approach can be split into functional architectural and functional behavioural. With a "functional behavioral" approach, a behavior description of the design is compiled into a layout. The compiler looks for the best architecture, and creates the design. Due to the complexity of the general behavioral compilation problem, behavioral compilers tend to concentrate on a limited domain, which can be implemented with a limited set of target architectures such as PLA's and data paths.

```
silicon compilation
-----------------------
|                      |
| structural           |
| functional           |
-----------------------
| architectural         |
| behavioral            |
```

Fig 1 different kinds of silicon compilation

In the architectural approach, the circuit is described in terms of architectural primitives. The primitives are described in a functional way. The compiler does some logic synthesis and minimization, but in a limited basis; also the compiler takes care of placement, routing, and transistor sizing. Examples of architectural compilers are MacPitts, Bristle blocks and Arsenic.

**Mac Pitts**[2] uses a fixed target architecture (data-path, control section and finite state machine sequencers). The VLSI circuit can be specified by an algorithmically high-level language. The generator generates the needed logic. The datapath is built out of bit_slices, and the control logic is implemented with a Weinberger array. Parts of the layout are automatically shared if the same hardware can be used, by multiplexing. The generality of Mac Pitts is limited because it is a fixed nMOS depletion-load process.

**Bristle blocks**[8] compiles a high level language into a data-path structure, which is built out of procedural cells, of which location and flavor of the connection points between cells are specified. Procedural cells are little programs that can draw itself, stretch themselves and
compute power requirements. These procedural cells make Bristle blocks more flexible than Mac Pitts, which uses fixed cells. An other example is Arsenic[10]. The two users inputs are a high-level functional description and constraints (allowable space, time and power). These are automatically decomposed into smaller parts, which are all described by function and constraints. The needed cell are generated according to the given constraint and function. The most striking fact is that the leafcells are generated automatically. This is a top-down approach, in contrast with Mac Pitts and Bristle blocks which are top-down bottom-up. Generated cells are often not that optimal as handcraft designed cells. Therefore the result of Arsenic for very regular circuits (i.e. a lot of the same cells are placed), will be worse than when using handcraft designed leafcells.

In the structural approach the silicon compiler does not investigate alternative structures that would have the same functionality. The circuit is described by specifying its components subcircuits and their interconnections in a hierarchical manner. The users has to do all decomposition and design of the leafcells by hand. Placement and routing algorithms are very important. Maybe a better name for this approuch is silicon assembly. Examples are Lubrick[11], Ldmc[7].

Lubrick is a hierarchical silicon assembler. The basic element is a cell (a virtual box with something inside, and output/input terminals). More complex cells can be built by using topological and geometrical operators on basic cells. In Lubrick special attention is given to wiring of cells. Wiring is merged together when building the layout (lazy evaluation of wiring). Used operators in Lubrick are: abut, channel and river-route.

Ldmc looks a little bit like Lubrick except the lazy evaluation of wiring and the channel and river-route.

Just as programming languages compilers generate assembly-language programs from programs written in high-level languages, modern silicon compilation generates layout (topology), functional description (simulation model) from high-level architectural descriptions. These descriptions are given in terms of structures, "known" to the compiler, such as PLA's, RAM, ROM, ALU. Such structures can be described functionally, in size, or in terms of mathematical, logical, or state equations.
2.1 Silicon compilers with module generators

The "known" structures are often called the modules. Silicon compilation would be feasible if we can generate these modules automatically, with a module generator(s) (a functional architectural silicon compiler). It consists of a functional compiler and a set of module generators. The functional silicon compiler is a preprocessor, which synthesizes the high level input language into lower functional behaviours, that can be handled by the different module generators (see fig 2). Mac Pitts and Bristle blocks have nearly the same structure.

![Diagram of possible silicon compiler](image)

The floorplanner is the manager who places all the different parts, and calls the different tools (e.g. router).

2.2 Module generators

A module is a functional part of the design, and is as every design determined by structure, physics and behaviour. The structure of a module is fixed to a small domain, and therefore only a small set of functions can be realized with a module (e.g. ALU, PLA). A silicon compiler need to have a lot of module generators, to realize all kinds of functions. The physics is represented by the leafcells (mask primitives), which have a basic function (see fig 3).
The behaviour has to be translated into an actual floorplan, that describes how the leafcells must be assembled to create the desired function. This is a top-down bottom-up approach.

The first generation module generators are specially written programs for generating one kind of module. The behaviour was compiled in placement of leafcells. They calculate the exact position of the leafcells. Writing these programs is not an easy task, because a lot of geometric calculations have to be made, to get the leafcells on the right place.

A much better approach is to split the generator into two parts. The first part generates the relative ordinal placement of the leafcells (the floorplan generator), and the second part is an assembler program which calculates the real position of the leafcells out of the relative placement (the silicon assembler, see fig 4).

```
+---------------+    +---------------+    +---------------+
| floorplan     |    | silicon      |    | LAYOUT       |
------| generator 1 |------| assembler    | LAYOUT |
+---------------+    +---------------+    +---------------+
      | input       |    | +---------------+
+---------------+
      | generator 2 |    |
+---------------+
```

Fig 4 Multiple module generators with a silicon assembler.

But still for every module generator a new program has to be
written. We only want to have one module generator. Examples of "the multi-program" approach are the module generators MKPLA and MKROM written in LDMc [7].

It seems possible to use only one generator, because of the apparent similarities between the different types of modules. The type-dependant information is concentrated in a small first input stage of the generator, that will be processed by using the same module-generator (see fig 5).

```
+---------------+      +---------------+
|users input 1 |      |users input 2|
      +---------------+      +---------------+  
      |syntax compiler|      |syntax compiler|
      +---------------+      +---------------+  
      |                  |      |                  |
      |                  |      |                  |
      +---------------+      +---------------+  
      |floorplan        |      |generator        |
      +---------------+      +---------------+  
      |                  |      |assembler        |
      +---------------+      +---------------+  
      |                  |      |                  |
      |                  |      |                  |
      +---------------+      +---------------+  
      |                  |      |                  |
      |                  |      |                  |
      +---------------+      +---------------+  
      |                  |      |                  |
      |                  |      |                  |
      +---------------+      +---------------+  
      |                  |      |                  |
      |                  |      |                  |
      +---------------+      +---------------+  
      |                  |      |                  |
      |                  |      |                  |
      +---------------+      +---------------+  

Fig 5 One module generator concept with a small different input stage to create the a module input syntax.
```

```
<p>|
|---|------------------------------------------------|
|   | functional decomposition                       |
|   |------ silicon                                 |
|   | v compilation                                 |
|   | one module                                    |
|   | generators---generator-------&gt;| module assembling|
|   |                                     |
|   | standard cell                                 |
|   |                                     |
|   | gate array                                   |
|   |                                     |
|   | symbolic                                     |</p>
<table>
<thead>
<tr>
<th></th>
<th>v mask</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>architectural generality</td>
</tr>
</tbody>
</table>
```

Fig 6 moving generators towards silicon compilation

3.30
2.3 One module generator

The different design strategies are depicted in fig 6. The one-module concept moves the model generators in the direction of silicon compilation, because architectural generality is increased. Silicon compilation is feasible by decomposition of the high level input language, into functions that can be realized with module generators. The modules are assembled together and the whole layout is ready.

To realise the one-module-generator concept, we have to take a close look at the specifications of a module generator. The generator has to map the behaviour of the module into a layout with a known floorplan, using mask primitives. The layout is determined by behaviour, floorplan and leafcells. The one-module-generator can only be realized if all the three constraints (behaviour, physics and structure) are inputs of the program. This one-module-generator concept is a recent topic of research. Therefore let us look to some examples of these generators as GENESIS [4] and the SDA generator [5].

GENESIS is a silicon compiler development system. It is an environment in which users can create their own custom module generators. It has three development input languages:
1. development of functional simulation models
2. development of building rules and formulation of block compiler algorithms
3. development of menus and windows.

With GENESIS a completely new module generator can be generated. It is a very complex tool which makes it easier to create module generators.

According to SDA it is a generator that can generate different module generators by using three inputs:
1. array structure template (with optional optimizing procedures)
2. library of leafcells
3. personality matrix (optional)

Arrays of the same leafcells can be specified in the template. Different leafcells in one array have to be specified in the "personality matrix". This personality matrix describes the exact placement of the leafcells. This means that not the behaviour is given in this matrix. By giving the exact placement of the leafcells this generator looks more like an assembler. This "assembler behaviour" is caused by the fact that the floorplan information is not well separated from the behavioural information. It is a
must that the behaviour is separated from the floorplan to get a one-module-generator concept. The optimizing procedures are used to improve the layout by deleting unused parts. The SDA generator does no lazy evaluation of wiring, and therefore the optimizing procedures are very complex.

The separation of floorplan and behaviour means that, the floorplan only must contain information about placement of the different parts of the design. The behaviour input may not contain any floorplan information, but only behaviour. Different behaviours have to fit in one floorplan. This means that the must have some freedom. The behaviour determines the actual floorplan which describes layout of the module. (eg. number of inputs).

3. Silicon assembly: generating layout from a floorplan

The floorplan describes the relative placement of the leafcells in the total layout (see fig 7). The assembler calculates the real location of the leafcells, and creates the layout.

```
+-----------+   +-----------+   +-----------+
| floorplan |   | silicon |   | assembler |
+-----------+   +-----------+   +-----------+
| leafcells |   | LAYOUT |
+-----------+
```

Fig 7 Inputs of a silicon assembler

The layout is described in a hierarchical way. The basic elements are leafcells. It has a virtual box (with something inside) and input and output terminals. These basic elements are used to build more complex cells (which can again be used to build new complex cells). Only the relative placement is given, the assembler calculates the exact position, which is a two dimensional problem. It would be a lot easier if it could be reduced to a one dimensional problem. For defining the syntax of the floorplan and the specification of the assembler, we first have to look to the properties of a module.
3.1 Properties of a module.

A module is a structured design which has some regularity. The regular parts are called the models. The module generator generates the models separately, they will be routed later on. A model is built out of slices (a model is sliceable). These slices are built by abutting other slices or primitives (the leafcells). This module structure is depicted in fig. 8.

![Module Structure Diagram](image)

Fig. 8 The module

The slices are built by abutting subslices or maskprimitives in one direction (horizontal or vertical). Now the two dimensional placing problem is reduced to an one dimensional problem.

Nearly all wiring is declared in the leafcell, and automatically built when abutting the leafcells. This is in contrast with place and route strategy, where it has to be routed later on (see fig 9.b).

3.1.1 Abutment. Abutment is a very often used technique in VLSI. Parts (slices) are placed next to each other. The connection between two cells are automatically made when calls are placed correctly (see fig 9.a). This means that
the leafcells have to be designed, i.e. pitch of terminals and corresponding layer of connecting terminals have to be correct.

Wiring and components are declared in the leafcells. This wiring can be global, i.e. wiring which is used to connect different (leaf)cells. All other wiring (which stays is the cell) is local. Wiring through a cell is now possible by defining these (global) wires in the cells. This wiring-through saves a lot of area.

An alternative for abutment is the place and route strategy. The cells are placed and a router wires all the terminals (see fig 9.b). Extra area is needed for this routing. The layout problem gets more complicated and the layout will be bigger.

![Fig 9 layout strategies](image)

Abutting makes the module generating easier. But the leafcells are more difficult to design, because the wires have to be declared in the cells and terminal pitch has to be correct. The cells have to be placed in such a way that the connections are right. The exact position of the abutting cell has to be calculated according to an abutment strategy.

3.1.2 Abutment strategies The module assembler generates mask layout, by placing the leafcells according to the floorplan. The module is assumed to be sliceable, and
therefore the assembler only need to have one basic operator; the abut operator (horizontal and vertical). Several abutment strategies are possible. One of them is corner aligning, as used in the SDA generator [5]. The abutment is done by aligning the corners of a users declared box. This is the easy way of abutting, which doesn't guarantee correctness (i.e. wrong intercell wiring).

Because correctness is the most important constraint a better approach is terminal stitching. The cells are only abutted if all the terminals fit, this guarantees that the composite will be correct. You may think that all the problems are moved to the designer, because he has to define the terminals on the right place. This means careful leafcell design. But corner aligning has the same problem because the wires must make contact.

3.1.3 Wiring in the module A big disadvantage of abutment is that the global wiring (intercell connection) is built out of parts which are hidden in the leafcells. The whole layout has to be expanded to be able to check these global wires. These connections of global wiring is important, because leafcell layout is assumed to be correct, and the whole layout will be correct if all leafcells are connected correctly.

The finished layout is totally flat, and an enormous data reduction could be achieved, if global wiring is merged together, when building the layout (lazy evaluation as used in Lubrick[11]) or by postprocessing (as Polymerge of Lattice Logic). For a big chip this postprocessing takes a long time. So lazy evaluation of the wiring looks to be a better solution.

4. MoDG a module assembler

MoDG is a silicon assembly special intended to assembler modules. The inputs are a floorplan and a library with leafcells. MoDG has to fit in an existing SPIRIT environment. In the next paragraph this environment is explained.

4.1 MoDG in the SPIRIT environment

The SPIRIT environment consists of a set of tools wich work on a database. MoDG will be one of these tools and has to work on this database too. GLE is one of these tools and can show the contents of the database on a grafic display (see fig 10).
Fig 10. MoDG in the Spirit environment.

The most important views of the database are the layout and network view (see fig 11). The circuit is hierarchical defined as models which consists of:
- communication with the environment: the terminal files
- abstraction to the environment: the information files
- local autonomy: the contents of the model defined as layout primitives (the boxes) and calls to sub-models (the model call files)

As stated before, a silicon compiler is not only the generation of layout. It is very important that the layout is correct. The layout is correct when the layout is built
correctly (correct by construction). This can be the case if layout is generated automatically.

Designrules are often violated when assembling the layout by hand, and the designrules have to be checked with a DRC (design rule checker). The electrical correctness can be checked with different network simulators. These simulators need a network description as input. It is possible to extract the network from the layout with a network extractor. But it is easier to generate the network parallel with the layout. This is not difficult because the silicon assembler knows which terminals are connected when building the layout, thus also the network is known.

The network of the module is not automatically generated in this version of the module assembler MoDG. This is done because it is possible that SPIRIT gets a new database with a real database manager. At the moment it is not known which jobs the manager takes care of. This version of MoDG only generates the layout.

4.1.1 The input of MoDG The assembler needs two inputs:
   - a floorplan
   - a leafcell library

The floorplan is a text file, of which the syntax has to be defined. The leafcell library (mask primitives) are captured in the database. These primitives can be designed by hand (GLE), automatically with a module generator (MoDG) or with a symbolic notation (MKCELL).

4.1.2 The output of MoDG MoDG creates a new module in the database, which can be used by all other tools (also MoDG).

4.2 Floorplan syntax

The floorplan describes how the module has to be made. A module is defined as a set of models. These models are sliceable as depicted in fig 8. The names of the module and used models have to be specified in the floorplan. Each model has to be declared by abutment of slices. A slice is defined as a leafcell or by abutment of other slices. The names of the leafcells which are used in the model definition are alias names. The names of the leafcells which have to be used must be specified in a leafcell alias list.

The abut direction is not explicit given in the floorplan. The abut direction is implicit given with hierarchical nesting level of the module. The highest level (LEVEL 0), is the level on which the model is declared. The slices in
this level are abutted horizontal (on the right side of). A next level exists (LEVEL 1) if a slice of a previous level consists of other slices (see fig 12.b). These slices are abutted vertical (on top of).

```
+-----+------+
| left-| right- |
| slice1| slice2 |
|       |       |
+-----+------+
```

```
+-----+------+
| slice1 |     |
|       |     |
+-----+------+
```

```
+-----+------+
| slice1 |     |
|       |     |
+-----+------+
```

```
+-----+------+
| slice3 |     |
|       |     |
+-----+------+
```

```
+-----+------+
| c | c | c | c |
|   |   |   |   |
+-----+------+
```

```
+-----+------+
| d | b | d | c |
|   |   |   |   |
+-----+------+
```

```
+-----+------+
| c | c | c | c |
|   |   |   |   |
+-----+------+
```

Fig 12. abutment according to level with its floorplan syntax.

The slices are abutted horizontal if the nesting level is even (see fig 12.a and fig 12.c). If the nesting level is odd, they will be abutted vertical (on top of).

This automatically chosen abut direction (according to the nesting level), obliges the user to create a well structured floorplan, by dividing the module in a structured way.

The complete syntax is given in appendix 2. It is possible that an extra nesting level has to be introduced to get a wanted abut direction, e.g. if the highest level has to be abutted vertical.
4.3 The program MoDG

The floorplan is parsed to the internal datastructure with a parser (see fig 13). This parser is generated with LEX and YACC. A good error recovery is included which helps the user to get a error free syntax. The needed leafcells are read from the input database.

```
+----------------+   +----------------+
| evaluate       | -> | generated layout |
+----------------+   +----------------+

floorplan +-------1--------+ generated
1 evaluate 1
1 1

+-------1--------+ floorplan +-------1--------+
| v       | 1 layout |
+-------+ 1 1 1

leafcells +-------1--------+ stripped
1 +-------1--------1 1 leafcells

------------->Iread|strip|writeI----+ stripped +----------------+
| leafcells
| MODGENpath
| MGuserpath <--ENVIRONMENT
| systempath current directory v

v directory v
+----------------+ +------------------------+
| in database 1 | 1 output database 1 |
| - leafcells 1 | - stripped leafcells 1 |
+----------------+ +------------------------+
```

Fig 13 the program MoDG

The used leafcell library is set with the "unix environment" in a shell script. A good example of a shell script is given below. For every new path (or floorplan) a new script must be made. The used leafcells are first searched in MGuserspath, then in MODGENpath and finally in the current directory.

```
"shell script"

MODGENpath= <system database>
MGuserpath= <users database>
export MGuserpath
export MODGENpath
MoDG <floorplan>
```

Fig 14 invoking MoDG by using a shell script
4.4 Stripping the leafcells

MoDG uses lazy evaluation of wiring. To settle this, MoDG must know the wiring in the leafcell. The database doesn't support this wiring strategy. Therefore MoDG must look into the leafcells, and find the wires. Only wires which can merge with other wires are useful for lazy evaluation. These wires must "end" on a terminal (the terminal-wire see fig 15).

```
terminal --v
---****   ---****   ---****   ---****
    ***   |      ***   |      *\*   \ width
---****   ---****   ---****   ---****
  \>-|<-|\-
  \ depth
```

Fig 15 finding terminal wires for lazy evaluation

This "end" means that the width of the terminal and wire must be equal, and the depth of the terminal must lay within the wire. The terminals are internal saved in the corresponding terminals. A terminal wire is global if both ends of a straight wire "end" on a terminal (i.e. both terminals have the same terminal wire). All other terminal wires are local. The leafcells of the input library are stripped by skipping all their terminal wires. The stripped leafcells are written to the output database. These stripped leafcells are used to build the module.

The generation of the layout starts after all the leafcells are read.

4.5 Evaluating the floorplan

A module is a set of models. The models are made seperately by abutting a list of arguments. These arguments are slices or leafcells, which can also be built out of other (sub)slices or leafcells.

The floorplan is evaluated depth-first. The slices are evaluated if all the aruments are known (i.e. layout is ready). All the arguments are abutted one by one, and the slice is growing. The slice is ready if all arguments are abutted, and the slice is written to the database. The possible operators are abut and array. The array operator
is just a repetition of abut-statements, so the heart of the program is the abut function.

4.6 Abutment with terminal stitching.

The abut function calculates the placement of a leafcell or slice. The used abut function uses the terminal stitching procedure, the abut direction is chosen according the hierarchical nesting level (see fig 12).

To abut two slices, the terminals of the adjacent sides are divided into "terminal groups". Slices abut if each "terminal groups" of one slice fits with a group from the other slice. A terminal group is a set of terminals which lay on top of each other, all within the width of the widest terminal (see fig 16).

Two terminal groups are said to "fit" if at least one terminal of each group fits. The terminal fits if the terminals have the same mask layer, and if the width of the cross section of these two abutted terminals, is equal to the width of the smallest terminal (see fig 17). In other words, the smallest terminal must "fit" in the other one.
Two slices fit if all their terminal groups fit. The only exception is that terminals of type vdd or vss may shift-out (see fig 18).

Fig 17 terminal fit

Fig 18 shift-out of terminals

4.7 Lazy evaluation of wiring

The position of the slices is fixed after the abutment. A copy of the placed leafcell is made with this placement. Terminals which don't connect are connected, and terminal wires are merged together where possible. Wires only merge if the width of the terminals are equal (and have the same layer). All terminals of the adjacent sides of the abutting slices are updated according to this concept (see fig 19).
There are several cases for updating fitting terminals. These are given in the table given below.

<table>
<thead>
<tr>
<th>same</th>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>merge wires together</td>
<td>expand wire</td>
</tr>
<tr>
<td></td>
<td>(also makes a connection)</td>
<td>of smallest terminal</td>
</tr>
<tr>
<td>term1</td>
<td>local</td>
<td>global</td>
</tr>
<tr>
<td>term2</td>
<td>local</td>
<td>global</td>
</tr>
<tr>
<td></td>
<td>make wire</td>
<td>update term2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete all the adjacent terminals</td>
<td>(the slices are copies)</td>
</tr>
</tbody>
</table>

Fig. 20 updating the terminals "lazy evaluation"

The terminals of the made slice are the terminals of the parts, without the deleted adjacent terminals.
The terminal wires are lifted into a higher level. This lifting stops if the wire is no longer a terminal wire, at that moment the wire is placed in the slice which is being evaluated.

4.8 The leafcells

The used leafcells are defined in the leafcell alias list and are read from the library database. They can be read with user defined rotation or mirroring.

<table>
<thead>
<tr>
<th>4.8</th>
<th>The leafcells</th>
</tr>
</thead>
</table>

| R0  | rotation 0   |
| R90 | rotation 90  |
| R180| rotation 180 |
| R270| rotation 270 |
| MX  | mirror X-as  |
| MY  | mirror Y-as  |

All leafcells must be declared with the orientation as used in the floorplan, and they must fit without rotating the leafcells. The leafcells must not contain model calls of subcells.

4.9 Design of the leafcells

Cells can be designed with help of MoDG. MoDG always generates an output layout, even when leafcells are missing. The generation is stopped if a not defined leafcell is placed or when two slices (or leafcells) don't abut. From this erroneous situation a notice is given and the parts that caused the error are written into the database, which can be inspected by the user. He can use this generated output to design or redesign the leafcells.

5. Discussion

MoDG is a well structured program, capable of generating modules. Many modules have been generated with MoDG (e.g. PLA, MMU, ROM), in a very short period, without much difficulty. The result shows that it is a first step to the one-generator approach. The yet implemented MoDG is a very useful tool for generating layout from a textfile. The designer could have done the same by using a layout-editor, and carefully abutting the cells by hand. Actually the results produced by MoDG are potentially better because of the mentioned precise abutment on terminal boundaries and the lifting of the global wiring into a higher level; that guarantees their correct connection and no post processing
is necessary. So the tedious and error prone job of placing
some hundred of the same or (worse) slightly differing
pieces of layout should belong to the past.

When analysing these module generations of mentioned try­
outs we can see that the floorplan-input ( which actually
generates the module ) is too complex. The designer has to
give to much information about relative placement of the
leafcells. He only wants to give the behaviour of the
module, and the position of input and output terminals. For
instance if we look to the PLA generator the only relevant
input information is the programming of and-plane, or-plane
and position of the planes. The size of the ground_slice or
input_slice can be determined from the and_plane and
or_plane, and should not be specified in the floorplan. The
designer should not be bothered with this kind of leafcell
placement, but instead should be allowed to indicate these
dependancies in a generic floorplan.

In current version of MoDG the behaviour is given by the
placement of leafcells in a slice. It would be better if
the behaviour is written as output=function(input),
implicitly defining a sparse matrix. If a relation is given
a leafcell is placed, if not a default cell is placed.

To bring in both desired improvements a new version of MoDG
will be created, of which the old part of MoDG is only a
part, playing the role of an intelligent assembler (which it
plays very well as mentioned, and it can be used as an
stand-alone tool). Characteristics of this new version will
be:

1. Interdependencies in slices of different parts can be
specified. The depending slices will be generated
automatically.

2. A more abstract syntax for describing the behaviour
will be developed. This syntax will be close to the
syntax of the module which has to be generated. In
fact, this level of abstraction of the generator is
close to the type-specific part of the whole process
that was mentioned in paragraph 1.2.

Bibliography.
systems" Addison Wesley, California.
the level of architectural description." Electronics.
toward silicon compilation." IEEE Circuits and
Devices.
Appendix 1-1

Manual MoDG

1. Introduction

MoDG is a module generator, capable of generating different kinds of modules. It uses two inputs:
- floorplan input file
- leafcell library

MoDG assembles the leafcells according to the floorplan. The placing is determined from the leafcell terminals. This means that the leafcells must be carefully designed. The generated layout is written into the Spirit database, and can be used by other Spirit tools (eg. GLE to show the layout see fig1.).

![Diagram showing MoDG in the Spirit environment]

2. Invoking MoDG

The used leafcell library is set with the "unix environment" in a shell script. A good example of a shell script is given below. For every new path (or floorplan) a new script must be made. The used leafcells are first searched in MGuserspath, then in MODGENpath and at last current
"shell script"

MODGENpath= <system database>
MGuserpath= <users database>
export MGuserpath
export MODGENpath
MoDG <floorplan>

The floorplan is a textfile written in the MoDG floorplan syntax (see appendix A). An example of a MMU floorplan with it's generated layout is depicted in appendix B.

```plaintext
+-----+------+
| left | right |
|slice1|slice2|
+-----+------+

a) MMU
LEVEL 0
hor. abutment

SYNTAX:
model MMU = abut( left_slice, right_slice);
right_slice = abut( slice1, slice2, slice3, slice1);
left_slice  = ...
slice1 = ...
slice2 = array(4) (c );
slice3 = abut (d,b,d,c)
modelend

Fig 3. abutment according to level with its floorplan syntax.
```
3. Slices of a module

A module is built by abutting slices (see fig 3.a). The module is built by abutting slices (see fig 3.a).

The abutment direction is determined by the nesting level. The highest level (LEVEL 0), is the level on which the model is declared. The slices in this level are abutted horizontal (on the right side of). A next level is exists (LEVEL 1) if slice of precious level consists out of other slices (see fig 3.b). These slices are abutted vertical (on top of).

The slices are abutted horizontal if the nesting level is even (see fig 3.a and fig 3.c). If the nesting level is odd, they will be abutted vertical (on top of).

This automatically chosen abut direction (according to the nesting level), obliges the user to create a well structured floorplan, by dividing the module in a structured way.

It is possible that an extra nesting level has to be introduced to get a wanted abut direction, e.g. if the highest level has to be abutted vertical.

---

a) ONE group; term1, term2 and term3 in one group

b) TWO groups; term1 and term2 in one group term3 in one group

Fig 4 terminal groups
4. Abutting slices and leafcells

To abut two slices, the terminals of the adjacent sides are divided into "terminal groups". Slices abut if each "terminal group" of one slice fits with a group from the other slice. A terminal group is a set of terminals which lay on top of each other, all within the width of the widest terminal (see fig 4).

Two terminal groups are said to "fit" if at least one terminal of each group fits. The terminal fits if the terminals have the same mask layer, and if the width of the cross section of these two abutted terminals, is equal to the width of the smallest terminal (see fig 5). In other words, the smallest terminal must "fit" in the other one.

Two slices fit if all their terminal groups fit. The only exception is that terminals of type vdd or vss may shift-out (see fig 6).

--- diagram 5 terminal fit ---

--- diagram 6 shift-out of terminals ---
5. The leafcells

The used leafcells are defined in the leafcell aliaslist and are read from the library database. They can be read with user defined rotation or mirroring.

- R0: rotation 0
- R90: rotation 90
- R180: rotation 180
- R270: rotation 270
- MX: mirror X-as
- MY: mirror Y-as

All leafcells must be declared with the orientation as used in the floorplan, and they must fit without rotating the leafcells. The leafcells must not contain model calls of subcells.

6. Design of the leafcells

Leafcells can be designed with help of MoDG. MoDG always generates an output layout, even when leafcells are missing. The generation is stopped if a not defined leafcell is placed or when two slices (or leafcells) don't abut. From this erroneous situation a notice is given and the parts that caused the error are written into the database, which can be inspected by the user. He can use this generated output to design or redesign the leafcells.

7. Wiring in the module

In MoDG the global wires which are declared in the leafcells, are lifted into a higher hierarchical level, and are merged together when ever possible. Wiring can be allocated by placing a empty cell. The terminals of the empty leafcell are wired through if necesarry.

8. The output of MoDG

The output of MoDG is written into the Spirit database. The models are placed in the directory where MoDG is invoked (current directory). The leafcells are copied into this directory. These leafcells differ from the original leafcells, because global wiring is missing. The leafcells may not contain other subcells, because these leafcells are not automatically copied into the database.
MoDG syntax.

Syntax description of the floorplan as specified in YACC

```plaintext
floor_plan  ::=  MODULE NAME LACC spec_list RACC
spec_list   ::=  MODELLIST LACC model_list RACC
                LEAF_ALIAS LACC leaf_alias_list RACC
                MODELSPECS LACC model_spec_list RACC
model_list  ::=  NAME
                |  model_list C NAME
leaf_alias_list ::=  leaf_alias
                |  leaf_alias_list SC leaf_alias
leaf_alias   ::=  NAME LACC NAME RACC
                |  NAME LACC NAME SHIFT_OUT shift_line RACC
shift_line   ::=  LACC terminal_list RACC
terminal_list ::=  NAME
                |  terminal_list C NAME
model_spec_list ::=  model_spec
                |  model_spec_list model_spec
model_spec   ::=  MODEL slice_list MODELEND
slice_list   ::=  slice
                |  slice_list SC slice
slice        ::=  NAME EQ operator LPAR name_list RPAR
operator     ::=  ABUT
                |  ARRAY LPAR INT RPAR
                |  EMPTY
name_list    ::=  name
                |  name_list C name
name         ::=  NAME
                |  NAME LBRAC NAME RBRAC
```

3.52
Terminal definitions as specified in LEX

MODULE ::= "module"
MODELLIST ::= "modellist"
LEAF_ALIAS ::= "leaf_alias"
MODELSPECS ::= "modelspecs"
SHIFT_OUT ::= "shift_out"
MODEL ::= "model"
ABUT ::= "abut"
ARRAY ::= "array"
PROG ::= "prog"
EMPTY ::= "empty"
MODELEND ::= "modelend"
SC ::= ";
C ::= ",
EQ ::= "=
LACC ::= "{
RACC ::= "}"
LPAR ::= "(
RPAR ::= ")"
LBRAC ::= "[
RBRAC ::= "]"
INT ::= digit_list
digit_list ::= digit
digit_list digit
digit_list ::= /*empty*/
NAME ::= letter chars
chars ::= char
Appendix 2-3

<table>
<thead>
<tr>
<th>chars</th>
<th>char</th>
</tr>
</thead>
</table>

char ::= letter
      | digit.

text ::= [a..zA..Z_]
digit ::= [0..9]
Appendix 3-1

An example of a MMU floorplan

```
fig 1. floorplan of a MMU

FLOORPLAN SYNTAX:

module mmu_chip {
    modellist { mmu }
    leaf_alias {
        matrixcel { mcel
            shift_out {vdd_l,vdd_r,vss_l,vss_r}
        };
        registerselect { rsel
            shift_out {vdd_l,vdd_r,vss_l,vss_r}
        };
        registercel1 { rgc1
            shift_out {vdd_l,vdd_r,vss_l,vss_r}
        };
        registercel2 { rgc2
            shift_out {vdd_l,vdd_r,vss_l,vss_r}
        };
        registercel3 { rgc3
            shift_out {vdd_l,vdd_r,vss_l,vss_r}
        };
        switch_adr { comp
            shift_out {vdd_l,vdd_r,vss_l,vss_r}
        };
        wirecel11 { pas3
            shift_out {vdd_l,vdd_r,vss_l,vss_r}
        };
        wirecel12 { pas4
```
modelspecs {
  model mmu = abut (l_slice,r_slice);
  l_slice = abut (mxslice,
               mxslice,
               mxslice,
               mxslice,
               mxtop);
  mxslice = array (3) (matrixcell);
  mxtop = abut(switch_adr, registercell1,
               switch_adr, registercell1,
               switch_adr, registercell1);

  r_slice = abut (regslice,
               regslice,
               regslice,
               regslice,
               rgtop);
  regslice = abut(registerselect,registercell2,
                  wirecell1, registercell2,
                  wirecell1 ,registercell2);
  rgtop = abut (wirecell2, registercell3,
               registercell3, registercell3 )
}
modelend

shift_out (vdd_1,vdd_r,vss_1,vss_r)
}

Floorplan description of PLA.

module pla1 {
  modellist { pla }
  leaf_alias {
/*hjkhkhhhlhgfthdfldf module tytghjugydfghlik
ahkjhjhjkkkkk*/
    input { plain
      shift_out ( l_vdd, r_vdd , l_vss, r_vss )
    };
    plp
    { plaplp
      shift_out ( t_vdd, b_vdd , t_vss, b_vss )
    };
    prog0
    { placl0 mx
    };
    prog1
    { placlb my R180
      shift_out ( t_vss, b_vss )
    };
    prog2
    { placll
      shift_out ( t_vss, b_vss )
    };
    prog3
    { placlr
      shift_out ( t_vss, b_vss )
    };
    gnd
    { plagnd
      shift_out ( l_vss, r_vss )
    };
    out
    { plaout
      shift_out ( l_vdd, r_vdd , l_vss, r_vss )
    };
    aoe
    { plaaoc
      shift_out ( t_vss, l_vss, r_vss )
    };
    rplp
    { plaplp R90
      shift_out ( t_vdd, b_vdd , t_vss, b_vss )
    };
    rprog0
    { placl0 mx R90
      shift_out ( t_vss, b_vss )
    };
    rprog1
    { placlb my R270
      shift_out ( t_vss, b_vss )
    };
    rprog2
    { placll R90
      shift_out ( t_vss, b_vss )
    };
    rprog3
    { placlr R90
      shift_out ( t_vss, b_vss )
    };
    rgnd
    { plagnd R270
      shift_out ( l_vss, r_vss )
    };
  }
}
Appendix 4-2

```plaintext
rout
{
plaout
shift_out { l_vdd, r_vdd, l_vss, r_vss }
}

w1
{
o_a_w1
shift_out { l_vss }
}

w2
{
a_o_c_w2
shift_out { l_vss }
}
```

```plaintext
modelspecs {
  model pla = abut( a_plane, a_o_con, o_plane );
  a_plane = abut (pla_in,
    gndslice,
    a_slice1,
    a_slice2,
    a_slice3,
    a_slice4,
    gndslice );
  pla_in = array(6) (input);
  gndslice = array(6) (gnd);
  a_slice1 = abut (plp, prog1, prog0, prog0, prog1, plp);
  a_slice2 = abut (plp, prog1, prog1, prog0, prog0, prog0);
  a_slice3 = abut (plp, prog1, prog1, prog0, prog1, prog0);
  a_slice4 = abut (plp, prog1, prog0, prog0, prog0, prog0);
  a_o_con = abut( w1,
    aoc,
    aoc,
    w2 );
  o_plane = abut (o_pull, o_slice1,
    o_slice2, pla_out );
  o_pull = array(6) (rplp);
  o_slice1 = abut (rprog1, rprog0, rprog0, rprog0, rprog1, rprog0);
  o_slice2 = abut (rprog3, rprog1, rprog3, rprog1, rprog3, rprog1);
  pla_out = array(3) (rout)
}
```

3.58
Fig. 1 Layout of PLA.
Chapter 4

The Piecewise Linear Simulator
This paper describes the principles of operation and main features of the piecewise linear, mixed level simulator, as developed in the ICD system. The simulator is capable of handling mixed analog digital circuits, and exploits latency and hierarchical structuring in order to be able to handle large systems. The component models are all piecewise linear, and not part of the program itself, but are part of the circuit description, and hence can be read from the circuit library. This provides for an enormous flexibility to derive macromodels for every specific application. Due to the uniform piecewise linear modelling approach, the simulator doesn't make any difference between analog and digital components.

1. INTRODUCTION

In the electronic industry, the use of simulation programs is widely accepted. Especially in the design of large scale integrated circuits these programs have become indispensable. Due to the high cost and long time delay imposed by prototype fabrication, a thorough design validation should precede prototyping. Computer simulation of the circuit behaviour has proven to be a reliable and fast method of checking the circuits proper operation before physical realization. At the moment several simulation programs for this purpose are available, developed by industries and universities. However a lot of problems still remain unsolved.
Traditional circuit simulators generate out of the circuit description a set of nonlinear implicit differential equations, consisting of both interconnection equations (according to Kirchhoff voltage and current law) and model equations which describe the physical behaviour of the components of the circuit. This set of differential equations is transformed to a set of algebraic equations for each timepoint, applying some stiffly stable integration method and stepsize control mechanism. For each timepoint, this set of nonlinear equations is solved, using some Newton Raphson type iteration algorithm. The main problems of this approach are the numerical stability of the Newton Raphson iteration and the excessive computational requirements. The difficulties with the Newton Raphson algorithm arise from the very strong nonlinearities typically found in device models, from the wide dynamic range of matrix coefficients, and from circuit characteristics like strong feedback or multiple solution curves. The large computation times are imposed by the recomputation of the whole set of circuit variables for each timepoint. Even for small circuits the matrix might be quite large and elaborate sparse matrix techniques are required. Furthermore the timesteps ought to be small to enable the Newton Raphson algorithm to solve the next operating point. All these conditions limit the practical use of these simulators to circuits not larger then about one hundred transistors, depending upon environment. With current state of the art integrated circuits approaching one million transistors, these simulators fall short in several orders of magnitude.

In order to cut the computational complexity, four basically different approaches were followed in the development of later simulators. Basically these changes are: Switching modelling and simulation to higher levels of abstraction, evading large-matrix processing by modularisation of the set of equations, exploiting latency effects in the circuit, and exploiting special hardware as for instance parallel processing machines. A combination of these techniques could be used in a particular simulation program. We will shortly discuss these approaches in the next few paragraphs.

Switching to higher levels of abstraction clearly reduces the required computational effort because both the modelling complexity as well as the amount of information to be generated, are reduced. Examples include rude transistor models, ultimately leading to the switch level approach [14], abstraction of subcircuits representing logic gates [97], and various logic and
register level simulators. The main problem with this approach is the inflexibility of current circuit level simulators in adopting new model equations, preventing the use of macromodels. Turning towards the switch level or logic simulators yields an enormous speedup, while losing a lot of information such as delay-times, spikes, and logic levels. Furthermore a mixing of a circuit and logic level simulation seems hard or impossible due to the basically different nature of the applied algorithms.

Evading large matrix processing by modularisation of the circuit (the set of equations) is only recently adopted in circuit level simulation. The obtained results are very promising and hence much work is under development at the moment. Basically the large set of equations is partitioned into a large number of small sets which are determined by some physical property. Typical choices are partitioning in sets describing logic cells, (automatically extracted) unidirectional subcircuits, or simply single nodes of the circuit. In the solution process, for each timepoint, all the sets are repeatedly solved individually, one after another, until convergence is obtained. In this "relaxation" process the next timepoint is solved for, only after the solution of the last timepoint is obtained. Alternatively the scheme can be modified to solve for a single set, over all the timepoints of the entire time interval, which is repeated over all sets. This scheme is denoted as "waveform relaxation", see figure 1.1.

Examples of a relaxation scheme can be found in [104], [99], of the waveform relaxation in [11], [18], [21], [59], [101]. These schemes tend to be efficient in simulating large digital circuits since these circuits are inherently modularized with a relatively low interconnection. Although very
powerful convergence proofs are derived for the simulation of MOS circuits. In practice, these algorithms sometimes suffer from a too low convergence rate. It appears that a good partitioning and a good order of evaluation of subcircuits are indispensable for obtaining an efficient simulator. However, the technique is promising, and especially for MOS digital circuits, efficient and accurate simulators might be built.

Exploiting latency effects in the circuit actually is a straightforward technique of skipping unnecessary operations during the simulation. Especially digital circuits tend to behave 'latent', which means that at each time instance only a small part of the circuit is really 'active' or 'changing'. For the simulation program, this means that a considerable speed-up could be obtained by only evaluating these active parts. Thus also some partitioning in the set of equations should have been done, in order to enable the program to choose at each time point, which subsets (subcircuits) are to be solved. In order to control these computations, some activity-list and/or some event-list should be maintained by the program, which induces some computational overhead. Furthermore, at the boundaries between boundary conditions are required to solve the active circuit. These values have to be available without recomputing the latent circuit, and are to be generated by some approximation. At the cost of an increased program complexity and the bookkeeping for the event determination, a considerable speedup can be obtained in simulating large modularized circuits. This exploitation of latency is commonly found in logic-level simulators and is easily incorporated in relaxation-based simulators.

Special or parallel hardware architectures have the ability for an enormous increase in computation speed, but the high costs have prevented a wide use. At the moment, special versions of conventional simulators like SPICE are available for vector-processing (Single Instruction Multiple Data) machines. In these programs, the parallelism is exploited at the processing of the matrix equations, for which these machines are optimized. However, in the near future we might benefit from Multiple Instruction Multiple Data machines in combination with some relaxation-based simulation program, in such a way that each processor can handle one or a few subcircuits, exchanging data with other processors for boundary conditions. Alternatively, special hardware has been developed for logic simulation as for instance the Yorktown Simulation Engine [1], [29], [63], [79], showing an unmatchable speed. At the time these hardware extensions become affordable to use for a wide public, its processing power might be needed in combination with sophisticated software in order to cope with future large-scale simulator tasks.
With the above described methods, the computational efficiency is remarkably increased, with some people believing that the sophisticated programs, currently under development, might reach the ultimate possible efficiency. Up to now nothing is told about the numerical stability problems of the Newton Raphson iteration process. In practice this appears to be a severe problem, because often no convergence and hence no simulation result can be obtained. For the waveform relaxation method powerful convergence proofs have been derived. The proof guarantees convergence for the simulation of all MOS circuits, if continuous static MOS transistor models are used and the circuit contains grounded capacitors at each node. However in practice some circuits with feedback show very low convergence rates, furthermore, because relaxation relies on a contraction property, for more general circuits the convergence properties are easily lost. Another way to avoid the Newton Raphson convergence problems is to adopt a piecewise linear modelling technique. It appears that the piecewise linear modelling together with specialized solution algorithms possess good global convergence properties. Furthermore fast model evaluation might be possible due to the removal of intrinsic nonlinear function evaluations. For these reasons, piecewise linear modelling has gained more attention in the circuit simulation field [5], [10], [25], [38], [41], [45], [52], [76], [90], [92], [99], [104]. These piecewise linear models are normally defined as a list of polytopes with an associated linear mapping on each polytope. For practical reasons, these mappings are normally restricted to one-dimensional functions (the polytopes are intervals) or two-dimensional functions on a rectangular grid (the polytopes are rectangles). Some time ago van Bokhoven studied the definition of piecewise linear mappings by means of a matrix-description, based upon the Linear Complementarity Problem [9]. It appeared that this type of definition was a powerful and flexible tool for describing simple as well as more general piecewise linear mappings.

Also piecewise linear modelling is ideally suited for the definition of macromodels. Because both low level circuit elements and higher level macros can be defined in the same type of description, a real mixed level simulator can easily be built. For these reasons a new simulator is developed based upon this piecewise linear modelling technique. The main properties of this simulator are pointwise noted below.

- This particular piecewise linear modelling has the ability for easy definition of efficient and compact macromodels. This feature is extensively supported by the simulation program, because models are not incorporated in the program code itself, but are part of the circuit
definition input. A large number of (parametrized) models are stored in a library, and each user can add his own new models if desirable.

- To solve the linear complementarity problem (the piecewise linear equations) one of the most powerful algorithms is implemented, which results in unique global convergence properties. Additional research has led to two new algorithms with optimized sparse matrix behaviour: a contraction algorithm and an integer labelling algorithm. These algorithms could be used in newer versions of the program.

- To be able to take advantage of latency effects, the equations are partitioned in a hierarchical structure. In this way the program has the ability to select only small parts of the equations for solving, while maintaining the ability to compute a global solution without an iterative relaxation. In the leafcells of the hierarchy the actual piecewise linear models reside, while the upper levels contain interconnection information only. By assigning individually optimized timesteps to each leafcell, latency effects are created in a natural way. An implicit stiffly stable integration method is applied, to ensure a stable analysis. The hierarchical structure of the equations matches the structure of the hierarchical circuit definition, as given by the user.

- The user interface of the simulator consists of an interactive schematics capture program for graphical circuit definition, an ndl-compiler for a textual circuit definition, an ndl-decompiler to obtain a textual description again from a previously (graphically) defined circuit, a linker to collect data from different files, together forming a complete circuit and an output-processor for a convenient handling, displaying and selection of the result of the simulation.

In section 2 the matrix notation for the description of leafcells is explained. In section 3 algorithms are given to solve the piecewise linear equations or, equivalently, the linear complementarity problem. In section 4 the hierarchical datastructure is explained, with an accordingly modified solution algorithm. In section 5 the structure of programs and files is presented, together forming the simulator with its environment. In section 6 some piecewise linear models are presented and finally in section 7 some conclusions and future extensions are given.
2. PIECEWISE LINEAR SYSTEMS

2.1 Static systems

A piecewise linear system can be treated as a mapping with input vector \( x \) and output vector \( y \) as depicted in figure 2.1.

![Figure 2.1. A piecewise linear system.](image)

A closed form matrix description for piecewise linear mappings is introduced by van Bokhoven [7], [9], which is capable of describing a large class of piecewise linear systems. This matrix description is denoted as given in equation (2.1):

\[
\begin{align*}
\mathbf{y} &= \mathbf{A} \mathbf{x} + \mathbf{B} \mathbf{i} + \mathbf{e} \\
\mathbf{v} &= \mathbf{C} \mathbf{x} + \mathbf{D} \mathbf{i} + \mathbf{f}
\end{align*}
\]  

(2.1)

in which \( \mathbf{x}, \mathbf{y}, \mathbf{i}, \mathbf{v}, \mathbf{e}, \mathbf{f} \) are real vectors, \( \mathbf{v} \) and \( \mathbf{i} \) of equal dimension say \( n \), and \( \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \) are real matrices. With \( \mathbf{v} \geq 0 \) we mean each element of \( \mathbf{v} \) to be positive and \( \cdot \cdot \cdot \) denotes the inner product. We will refer to this description (2.1) as the 'explicit form', contrary to the 'implicit form' which is slightly more general, is actually used in the simulation program, and is presented in section 4.1. In general (2.1) defines a piecewise linear mapping as \( 2^n \) different polytopes with associated linear mappings \( \mathbf{x} \rightarrow \mathbf{y} \). However, depending upon \( \mathbf{C} \) and \( \mathbf{D} \), some polytopes may overlap, some may not exist, or some \( \mathbf{x} \) may not belong to any polytope at all. The nature of (2.1) to define piecewise linear mappings can be shown as follows. Define the null-region \( V \) of the description (2.1) as:
\[ V = \{ x \mid v = Cx + f \geq 0 \} \]

Then \[ y = Ax + e, \ x \in V, \ i = 0 \] solves (2.1).

Due to the symmetry between \( i \) and \( v \) we can modify the description (2.1) to a new one (2.2) by the exchange of some elements of \( i \) and \( v \). Let \( I \subseteq \{ 1, \ldots, n \} \), \( I \) nonempty and \( D_{II} = (d_{ij}) \) with \( i, j \in I \). Now we exchange pairwise all \( i_k \) and \( v_k \), \( k \in I \), in (2.1) by simple linear programming methods (i.e. a partial inversion of \( D \)), assuming the determinant of \( D_{II} \) nonzero. After redefining the lefthand vector to be \( v' \) and the righthand vector to be \( i' \), we obtain (2.2):

\[
\begin{align*}
  y &= A'x + B'i' + e' \\
  v' &= C'x + D'i' + f' \\
  v', i' &\geq 0, \ v' \cdot i' = 0
\end{align*}
\]

(2.2)

This description is equivalent to (2.1) in the sense that it defines the same mapping \( x \rightarrow y \). However it has a different null-region \( V' \):

\[ V' = \{ x \mid v' = C'x + f' \geq 0 \} \]

So \[ y = A'x + e', \ x \in V', \ i' = 0 \]
solves (2.2) and hence (2.1).

Because generally, \( 2^n \) different sets \( I \) could have been used, (2.1) defines \( 2^n \) different polytopes \( V \) with associated linear mappings \( x \rightarrow y \).

We will refer to the above described exchange step as a "pivot operation". If the cardinality of \( I \) equals 1, we call \( D_{II} = d_{ii} \) the "pivot". If the cardinality of \( I \) is greater then 1, we call \( D_{II} \) a "block-pivot". Because in the above described operation the involved rows and columns always have the same indices, we can more strictly refer to \( D_{II} \) as a "diagonal (block-) pivot".
2.2 Example

As example the mapping of figure 2.2 is presented:

\[
\begin{align*}
\frac{t}{yl} & \quad - \quad \frac{x}{-} \\
\frac{y}{y_1} & \quad - \quad \frac{1}{x} \\
\end{align*}
\]

Figure 2.2. Hysteresis curve.

This mapping can be described by the relations (2.3):

\[
\begin{pmatrix}
  y \\
  v_1 \\
  v_2 \\
\end{pmatrix} =
\begin{pmatrix}
  -1 & -1 & 1 \\
  -1 & -1 & 1 \\
  1 & 1 & -1 \\
\end{pmatrix}
\begin{pmatrix}
  x \\
  i_1 \\
  i_2 \\
\end{pmatrix}
+ \begin{pmatrix}
  1 \\
  1 \\
  0 \\
\end{pmatrix}
\]

(2.3)

\[v, i \geq 0, \quad v \cdot i = 0\]

In order to check that equation (2.3) corresponds with figure 2.2 we first verify its null-region:

\[
\begin{align*}
V = \{x \mid -x+1 \geq 0 \land x \geq 0\} = \{x \mid 0 \leq x \leq 1\}
\end{align*}
\]

and thus \(y = -x + 1, \ 0 \leq x \leq 1\) solves (2.3) with \(i_1 = i_2 = 0\)

Pivoting on \(D_{11}\) yields:
\[
\begin{pmatrix}
  y \\
  i_1 \\
  v_2
\end{pmatrix}
= \begin{pmatrix}
  0 & 1 & 0 \\
  -1 & -1 & 1 \\
  0 & -1 & 0
\end{pmatrix}
\begin{pmatrix}
  x \\
  v_1 \\
  i_2
\end{pmatrix}
+ \begin{pmatrix}
  0 \\
  1 \\
  1
\end{pmatrix}
\text{(2.4)}
\]

with \( V' = \{ x | -x + 1 \geq 0 \wedge 1 \geq 0 \} \)
and thus \( y = 0, \ x \leq 1 \) solves \((2.3)\) with \( v_1 = i_2 = 0 \)

Equivalently, pivoting on \( D_{22} \) yields:

\[
i_1 = v_2 = 0, \ x \geq 0, \ y = 1
\]

Finally, pivoting on both \( D_{11} \) and \( D_{22} \) (inverting \( D \)) isn’t feasible because \(|D| = 0\), terminating the analysis of \((2.3)\).

For more examples the reader is referred to section 6.

2.3 Dynamic systems

The description of piecewise linear systems as given in \((2.1)\) can be extended to describe dynamic systems as well. This is accomplished by addition of the (equally sized) vectors \( u \) and \( n \), which are implicitly related to each other by integration over time:

\[
\dot{u} = \frac{\partial}{\partial t} u
\text{(2.5)}
\]

With these vectors the new piecewise linear system description is denoted as:

\[
\begin{pmatrix}
  y \\
  u \\
  v
\end{pmatrix}
= \begin{pmatrix}
  A_{11} & A_{12} & A_{13} \\
  A_{21} & A_{22} & A_{23} \\
  A_{31} & A_{32} & A_{33}
\end{pmatrix}
\begin{pmatrix}
  x \\
  u \\
  1
\end{pmatrix}
+ \begin{pmatrix}
  a_1 \\
  a_2 \\
  a_3
\end{pmatrix}
\text{(2.6)}
\]

\( v, \ i \geq 0, \ v \cdot i = 0 \)

An example of such a system is the square-wave generator \((2.7)\), which makes use of the same nonlinearities as found in \((2.4)\).
With the initial condition $u|_{t=0} = 0$, (2.7) can be solved to yield:

$$
egin{bmatrix}
  y \\
  \alpha \\
  v_1 \\
  v_2
\end{bmatrix} =
\begin{bmatrix}
  0 & 1 & 0 \\
  0 & -2 & 0 \\
  -1 & -1 & 1 \\
  0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
  u \\
  i_1 \\
  i_2
\end{bmatrix} +
\begin{bmatrix}
  0 \\
  1 \\
  1 \\
  1
\end{bmatrix}
$$

(2.7)

2.4 Analysis

One of the basic problems in piecewise linear analysis is stated as (2.9):

Determine, for each given vector $x$, all vectors $y$ satisfying (2.1)  

(2.9)

However for circuit simulation programs, it is often enough when the problem (2.10) can be solved:

Determine, for each given vector $x$ within some domain, one vector $y$ satisfying (2.1), assuming that at least one solution exists.  

(2.10)

By applying (2.1), one can easily see that these problems require the solution of the so-called Linear Complementarity Problem, stated as (2.11):
For given real square matrix $M$ and given real vector $q$, solve the vectors $v$ and $i$ satisfying:

$$
\begin{align*}
    v - M i &= q \\
    v, i &\geq 0, ~ v \cdot i &= 0
\end{align*}
$$

Actually, in the previous example (2.3), the solution was obtained by evaluating all $2^n$ possible zero-conditions for the vectors $v$ and $i$, for each condition checking whether the inequalities are satisfied. In practice this method isn't feasible due to the exponential growth in computation time as function of the problem size $n$. In literature, several algorithms are known that, within certain limitations, enable us to solve this linear complementarity problem in a more satisfactory way, although many of these algorithms show a worstcase nonpolynomial complexity. In section 3 we will continue with a more extensive treatment of these problems.

3. THE LINEAR COMPLEMENTARITY PROBLEM

As stated in section 2, the analysis of piecewise linear systems, applying the model-description (2.1), requires the solving of the Linear Complementarity Problem (LCP). During transient analysis (i.e. the simulation of time-response), the LCP has to be solved repeatedly for each time-point. The system to analyzed will often be very large, contain strong nonlinearities, and might show multiple solutions. For these reasons a powerful and efficient solution algorithm is required, which can make use of the solution of the previous time-point, to solve the next LCP. Due to the large size of the problem, an efficient sparsematrix implementation of the algorithm is required, for which the algorithm should be suitable. For these reasons, several different methods are evaluated, and some research is done to derive new algorithms. In this chapter, we will first present some characteristics of the LCP in general, to be able to compare different convergence properties. Afterwards a few different algorithms will be discussed.

In this chapter we will redefine the Linear Complementarity Problem (LCP) with new names for the involved vectors, which are more commonly used in literature, despite the previous use of these names for piecewise linear systems.
3.1 Classification

In this chapter we deal with solving the LCP as defined below in eq (3.1):

For given real square matrix $M$ and given real vector $q$, solve the vectors $x$ and $y$, satisfying:

$$\begin{cases} y = Mx + q \\ y, x \geq 0, \ y \cdot x = 0 \end{cases}$$

This LCP is known as a basic problem for quite some time already, and is mainly studied for applications in game theory and economics. An interesting application is the solution of the linear constrained quadratic optimization problem, which is equivalent to the LCP [61]. Below we will present some classes of matrices, suitable for the classification of the LCP, according to a paper of Karamardian [50].

- Positive definite matrices (PD)
  $M$ is of class PD if and only if
  $$\forall x \in \mathbb{R}^n, x \neq 0 : x^T M x > 0$$

- Positive semi-definite matrices (PSD)
  $M$ is of class PSD if and only if
  $$\forall x \in \mathbb{R}^n : x^T M x \geq 0$$

- P-matrices (P)
  $M$ is of class P if and only if
  $$\forall x \in \mathbb{R}^n, x \neq 0 \exists k : x_k \cdot (Mx)_k > 0$$

- PO-matrices (PO)
  $M$ is of class PO if and only if
  $$\forall x \in \mathbb{R}^n, x \neq 0 \exists k : x_k \cdot (Mx)_k \geq 0$$

- Strict copositive matrices (SCP)
  $M$ is of class SCP if and only if
  $$\forall x \in \mathbb{R}_+^n, x \neq 0 : x^T M x > 0$$

- Copositive Plus matrices (CPP)
  $M$ is of class CPP if and only if
  $$\forall x \in \mathbb{R}_+^n : x^T M x \geq 0$$
  and also
  $$\forall x \in \mathbb{R}_+^n, x \neq 0, x^T M x = 0 : (M + M^t) x = 0$$
- **Strict semimonotone matrices (SSM)**

  $M$ is of class SSM if and only if
  
  \[ \forall x \in \mathbb{R}^n, x \neq 0 \exists_k : x_k \cdot (Mx)_k > 0 \]

- **L-matrices (L)**

  $M$ is of class L if and only if
  
  \[ \forall x \in \mathbb{R}^n, x \neq 0 \exists_k : x_k > 0 \wedge (Mx)_k \geq 0 \]

  and also
  
  \[ \exists \Lambda, \Omega \geq 0 \forall x \in \mathbb{R}^n, x \neq 0, Mx \geq 0, x^T Mx = 0 : \]

  \[ \Omega x = 0 \wedge (AM + M^T \Omega)x = 0 \]

  in which $\Lambda$ and $\Omega$ denote diagonal matrices.

The relations between these classes can be depicted as in figure 3.1 where an upward link means "is a subset from".

![Figure 3.1. The relation between the matrix classes](image)

In relation to the LCP it is known that there exists a unique solution if and only if $M$ belongs to class $P$ [50]. This class is closed under diagonal pivot-transformations, and can also be characterized by the property that all principal minors of $M$ (i.e. $|D_{II}|$ as defined in section 2.1) are positive; which implies that all diagonal elements of $M$ are positive.

Furthermore there exists an odd number of solutions (at least one) if $M$ belongs to class SSM. This class isn't closed under diagonal pivot-transformations, but also has positive diagonal elements.
None of the above mentioned classes allow matrices to have negative diagonal elements.

The complexity of the solution algorithms generally appears to be of nonpolynomial complexity with respect to the dimension n of the problem. Only for the restricted case M in PD, van Bokhoven has presented a polynomial algorithm [9].

3.2 The Katzenelson algorithm

One of the oldest algorithms to solve a set of piecewise linear equations was presented by Katzenelson in 1965 [51]. The method is based upon the generation of a continuous path through the space of polytopes. The method is still extensively used in piecewise linear simulation programs (probably with extensions as defined by Chien [22]), mainly because it is easily programmed on piecewise linear mappings defined as lists of linear mappings and polytopes. However it can be applied to solve the LCP as well, and is described by the steps below:

1. Assume an initial solution $M_0$, $q_0$ is known with $x = 0$ and $y \geq 0$. The solution for some $q_*^\lambda$ is to be found. This solution is generated by tracking a continuous path, defined by the solution of the LCP for $q_\lambda^\lambda$, with $q_\lambda = \lambda \cdot q_0 + (1-\lambda) \cdot q_*^\lambda$, $0 \leq \lambda \leq 1$

   Set $k = 0$, $\lambda = 1$, $x = 0$, $y_\lambda = q_\lambda$.

   Goto step 2.

2. Decrease $\lambda$ until either some component of $y_\lambda = q_\lambda$ reaches zero, or $\lambda$ reaches zero.

   If $\lambda$ has become zero, $(M_k, y_\lambda, q_\lambda)$ solves the problem.

   Otherwise set $m$ at the index of the detected new zero component of $y_\lambda$, the so-called blocking row.

   Goto step 3.

3. set $k = k + 1$ and assume $M_{mm} > 0$. Perform a pivot-operation on $M_{mm}$ (exchange the position of the variables $y_m$ and $x_m$ in the description), define the new matrix to be $M_k$,

   and rename the new lefthand vector to be $y_\lambda$ again.

   Now $(y_\lambda)_m = 0$ (remind $x$ was zero), and

   $\delta(y_\lambda)_m/\delta\lambda < 0$ due to the positive pivot value.

   Thus a further decreasing of $\lambda$ will increase $y_m$ again.
Assuming the m to be determined in step 2 is unique, the convergence proof for class P is quite easy and based upon the monotonic decrease of $\lambda$ [60]. The problem involved by a non-unique m can be solved either by an infinite small displacement of $q_0$ [22], or by an ordering mechanism [62].

The limitation to positive pivots is quite restrictive, and is necessary to guarantee the monotonic decrease of $\lambda$. M in class P is a sufficient condition for this, since class P implies all possible pivots in all states to be positive. Due to the class P property, the LCP will always have a unique solution and the method is therefore guaranteed for (certain) piecewise linear functions only.

Later extensions of the algorithm allow for negative pivots to be performed, after which the direction of change of $\lambda$ inverts, and even zero diagonal elements can be handled, by traveling a line-piece with fixed value of $\lambda$ [22]. These extensions are not treated here, mainly because they resemble in some way the methods adopted in the Lemke and the van de Panne algorithms, to be found in the next sections.

The problem of finding an initial solution to the LCP as is assumed in step 1 is easily solved by taking $q_0 \geq 0$.

This is contrary to the problem when the piecewise linear mapping would not be defined by means of the LCP, but as a list of linear mappings on polytopes, as used by most other researchers in this field. In that case the initial solution should be given as a value for the inputs and outputs of the mapping. This problem is generally quite severe. Of course during transient analysis at each time-point the solution from the previous time-point can be used. However the problem remains to be solved the first time. In [22] some heuristics can be found to try and guess this initial solution. However the problem generally remains, because finding an initial solution is more or less the same as the actual problem to be solved by the algorithm.

3.3 The Lemke algorithm

An algorithm especially derived to solve the LCP was published by Lemke in 1965 [60]. This algorithm resembled the above mentioned Katzenelson algorithm in the sense that the solution is obtained by tracing a continuous path, controlled by a monotonic decreasing parameter ($\lambda$), starting at an artificially generated initial solution, and ending at the solution of the problem. In 1968 Lemke
published a powerful extension to his original algorithm. Now not only the λ-parameter but also components of the x- and y-vectors are manipulated in order to trace the path and reach the solution. Below the description of the algorithm is given.

1. Assume the problem to be solved is given by \( M_0, q_0 \).
   • If \( q_0 \geq 0 \), the solution is given by \( y = q_0, x = 0 \).
   • Otherwise choose a vector \( e \) and parameter \( \lambda \) such that \( \lambda > 0 \), \( y = \lambda \cdot e + q_0 > 0 \).
      Now decrease \( \lambda \) down to the point where the first component of \( y \), say \( y_k \), reaches zero.
      Treat \( \lambda \) as \( x_0 \), an extra component of the vector \( x \), and \( e \) as an additional column of \( M \), corresponding with \( x_0 \).
      Set \( j = 0 \), indicating the only positive component of \( x \). Proceed with step 2.

2. Perform a pivot-operation on the matrix-element \( M_{kj} \), guaranteed to be nonzero (exchange the pair \( y_k, x_j \)).
   Rename the lefthand-vector to be \( y \), the righthand \( x \).
   proceed with step 3.

3. Set \( j \) equal to \( k \).
   • Try to increase \( x_j \), until some component of \( y \) becomes zero. If no component of \( y \) reaches zero, \( x_j \) could be increased unlimited, and the algorithm terminates without a solution. Otherwise some component of \( y \), say \( y_k \), becomes zero.
      If \( k = 0 \), \( \lambda \) became zero, and a solution is obtained.
      Otherwise \( k \neq 0 \) and proceed with step 2.

We would like the reader to note that the algorithm might perform off-diagonal pivots, and that the originally defined complementary pairs \((x_k, y_k)\) get mixed. However the complementarity condition \( x_k \cdot y_k \geq 0 \), \( x_k \cdot y_k = 0 \) remains satisfied.

The algorithm can be applied to all matrices. However only for certain classes of matrices it is proved that a termination of the algorithm without finding a solution, implies that the problem \((M, q)\) is infeasible. Lemke himself proved this for class CPP [61], Dantzig and Cottle proved this for class SSM [26], and finally Eaves extended these results to class L matrices [31].

The convergence proof for the above mentioned algorithm relies on the fact that all visited states are complementary (i.e. \( x_j \cdot y_j = 0, 1 \leq j \leq n \)), and all states
have a unique successor (another complementary state) and a unique predecessor. Only for the initial state a predecessor cannot exist, and the final state doesn't have a successor. Therefore cycling cannot occur, and the algorithm must terminate in a finite number of steps, due to the finite number of different states (i.e. at most $2^n$ with $n$ the dimension of the problem). Despite this exponential worst-case complexity, the algorithm appears to behave quite efficient in practice.

3.4 The van de Panne algorithm

In 1974 C. van de Panne published an algorithm to solve the LCP, equivalent to the Lemke algorithm. At the cost of a more complex algorithm, he obtained two advantages: a powerful feasibility check can be done during the run of the algorithm (for which its author proposes this algorithm), and only diagonal (block-) pivots are performed (this is the reason for which we selected this algorithm).

For the description of the algorithm we will adopt the term "transparent stack" describing a last-in-first-out memory structure containing records all being visible (i.e. can be checked). The term "active variable" refers to the single variable ($\lambda$ or component of $x$) which is manipulated (forced) by the algorithm and can be seen as the current parameter tracing the path. The term "blocking row" refers to the row of the matrix (or, equivalently, to the single component of $y$) bounding the movement of the current active variable, by becoming zero. These blocking rows are pushed on the stack, until an "unblocking" occurs, i.e. the row becomes nonzero again.

The algorithm is defined by the six steps given below:

1. Assume the problem to be defined by $\langle M,q \rangle$.
   Choose a $\lambda$ and vector $e$ such that:
   $\lambda > 0, \ y = \lambda \cdot e + q \geq 0$.
   Let $j$ denote the active variable $x_j$, with $j=0$ denoting $x_0=\lambda$. Set $j=0$ and the direction of the active variable at 'decreasing'. Define the records of the transparent stack to contain the number ($j$) of a previously active variable, with associated direction, and initialize the stack to contain a single record, with values $j=0$ and 'decreasing'.
   Proceed with step 2.

2. Check for the unblocking of blocked rows:
   Set $d = \text{the number of unblocked rows determined by}$:
- d denotes a number of records in the stack, measured from the top.
  0 ≤ d ≤ current size of the stack.
- if d > 0, let k be the column-number stored in the d'th record,
  and M_{kj} - \frac{\partial y_k}{\partial x_j} ≠ 0
- There is no larger d for which the above conditions hold.

If d = 0, proceed with step 3.
Otherwise, keep the sign of unblocking = sign of \frac{\partial y_k}{\partial x_j} for later use, and proceed with step 6.

3. No unblocking occurs, perform a step indicated by the current active variable x_j and its direction:
   Move x_j in the current direction until a blocking is detected either by some component of y (say y_k) becoming zero or by x_j itself becoming zero.
   If no blocking can be detected at all, the algorithm terminates without finding a solution.
   If x_j became zero and j = 0, \lambda became zero, the algorithm terminates and a solution is obtained.
   If x_j became zero and j ≠ 0, proceed with step 4.
   If y_k became zero, proceed with step 5.

4. The algorithm succeeded in reaching zero for an active variable x_j, not being \lambda.
   Remove the top-of-stack record, which described row j as blocking row, from the stack by popping up the stack one level.
   Determine a new j (active variable x_j) and associated direction according to the new top-of-stack.
   Proceed with step 2.

5. A new blocking row, say k, is detected.
   Choose a new active variable x_j with j = k.
   Set the new direction to 'increase'. Push this new j (blocking row k) and direction on the stack.
   Proceed with step 2.

6. An unblocking of d rows has been detected. (d > 0)
   Perform a diagonal block-pivot upon the rows and
columns, corresponding with the variable-numbers stored in the top \(d\) records on the stack. Remove these \(d\) records from the stack.

If the unblocking occurred with a negative sign, change the direction of move in the new top-of-stack record.

Choose a new \(j\) (active variable \(x_j\)) and associated direction according to the new top-of-stack. Proceed with step 2.

It can easily be proved that this algorithm proceeds along the same path as the Lemke-algorithm would follow, that the same convergence-properties are obtained (processing of all class \(L\) matrices), and that the block-pivot in step 6 always has a nonzero determinant (see [77]).

The property that only diagonal block-pivots (see section 2.1) are performed, is indispensible in our application on a hierarchical datastructure, as pointed out in section 4.

The infeasibility-tests that can be incorporated in the algorithm, are removed from the description above. We always assume in our application (the transient analysis of physical systems) the existence of a solution.

4. **Hierarchical Analysis**

In the previous section several algorithms were presented to solve the LCP. For a first implementation of the piecewise linear simulator, we have chosen for a pivoting-type of algorithm. This choice is mainly based upon the superior global convergence properties and upon the encouraging results of small test-programs, showing both fast analysis and good convergence properties, at our university as well as at other sites [8],[104]. However, the pivoting-type algorithms require a complex sparse-matrix structure, and sparsity might even be lost during the run of the program. The solution we adopted, is partitioning the large set of equations in a hierarchical way. In this way large sets of equations don’t occur anymore, and also latency effects could be explored (see introduction). In this chapter we will present the adopted hierarchical datastructure, together with the accordingly modified algorithms. In the last section of this chapter we will treat the integration-method, with the implicitly related latency evaluation.

In the hierarchical partitioning, we distinguish between ‘leafcells’, containing the piecewise linear matrix definition of (small) components, and ‘compounds’, containing interconnection information only. Leafcells and
compounds form together the hierarchical structure, as denoted in figure 4.1.

Figure 4.1. The hierarchical structure.

The particular hierarchical structure for each simulation task is extracted from (corresponds with) the system definition, as given by the user, either by means of a textual description, or by means of the interactive graphics schematics capture program.

4.1 The leafcell description

The leafcells of the hierarchy are given by the implicit matrix definition (contrary to the explicit definition (2.1)), given in equation 4.1:

\[
\begin{pmatrix}
0 \\
u \\
v
\end{pmatrix} -
\begin{pmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{pmatrix}
\begin{pmatrix}
x \\
u \\
i
\end{pmatrix}
= 
\begin{pmatrix}
a_1 \\
a_2 \\
a_3
\end{pmatrix}
\]  

\(v, i \geq 0, \quad v^T \cdot i = 0\)

\(u = \frac{\partial}{\partial t} u\)

In this definition \(x\) denotes the vector of (concatenated) input and output variables, to be treated identically. Clearly (4.1) is more general than (2.1): (2.1) assumes that \(y\) can be determined from a given \(x\), in each segment of the piecewise linear mapping, thus ignoring a degeneracy which might occur in some segments. More general, (4.1) assumes that a corresponding number of
boundary conditions are imposed, completing (4.1) to a square matrix.

As an example the x-y-relation as depicted in figure 4.2 is presented:

This relation between x and y can be described by a matrix in the explicit form as in equation (4.2), as well as in the implicit form as in equation (4.3).

\[
\begin{bmatrix}
y \\
v
\end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4.2}
\]

\[
\begin{bmatrix}
y \\
v
\end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4.3}
\]

Whereas both (4.2) and (4.3) describe the mapping of figure 4.2, only (4.3) can be pivoted, and thus solved by some pivoting algorithm.

We notify the reader that in the explicit matrix definition (2.1), a submatrix $A_{33}$ always implies the mapping $x \rightarrow y$ to be a function, because in this case the LCP has a unique solution for each $x$. However, in the implicit leafcell description (4.1), this statement doesn’t hold anymore. Where the mapping of figure 4.2 clearly doesn’t represent a function, the corresponding $A_{33}$ submatrix in equation (4.3) is a (one dimensional) unit matrix, thus belonging to class P. It appears that we have been able to define also models with the unit matrix, for instance hysterese curves, flipflops and ideal switches.
4.2 The compound description

The compound cells of the hierarchy contain information about the interconnection of its subsystems with the outside world (i.e. one level up in the hierarchy). These subsystems themselves can either be leafcells or compounds again. It is assumed that the jacobians \( J \) of all its subsystems are determined, \( J \) denoting the linear relationship between the variables \( x \), imposed by the current segment of the piecewise linear modelling. With these jacobians the equations of a compound cell are denoted as:

\[
\begin{pmatrix}
0 \\
0 \\
0 \\
0 \\
0
\end{pmatrix}
\begin{pmatrix}
S_0 & S_1 & S_2 & \cdots & S_t
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_t
\end{pmatrix}
= \begin{pmatrix}
0 \\
h_1 \\
h_2 \\
\vdots \\
h_t
\end{pmatrix}
\]

with:
- \( x_0 \) denoting the vector of input-output variables to the outside world.
- \( S = (S_0 \ldots S_t) \) denoting the interconnection matrix. Generally a sparse matrix with element values +1, -1 and 0.
- \( x_i \ (1 \leq i \leq t) \) denoting the vector of input- and output variables of the \( i \)'th subsystem of this compound.
- \( 0 = J_i x_i + b_i \ (1 \leq i \leq t) \) denoting the linear constraint imposed upon \( x_i \) by the \( i \)'th subsystem in its currently valid segment of the piecewise linear mapping.

If we assume in the leafcells the vectors \( u \) and \( a \) absent (they will be treated at the end of the chapter), and the vector \( i \) to be zero (imposed by the pivoting algorithm), then the constraints \( (J,b) \) imposed upon \( x \) by the leafcell, are easily determined as \( (A_{11},a_1) \), given in (4.1).

The compound equations (4.4) are linear equations only, and therefore the constraints \( (J,b) \) of a compound are easily obtained by eliminating the
variables $x_1 \ldots x_t$ from (4.4). In this way the jacobians $(J,b)$ can be derived bottom-up in the hierarchy. In the equations (4.1) as well as in (4.4), the number of variables is larger then or equal to the number of equations. We assume the compound-cell at the root of the hierarchy, not to have any external variables $(x_o)$ anymore. Thus the matrix in the root (4.4) is square, and its variables $x$ can be solved. Now all variables $x$ in the tree can be solved in top-down order, because in each compound cell, the external variables $(x_o)$ are known, and enough equations are provided to solve the remaining variables (internal variables $x_1 \ldots x_t$).

4.3 Basic operations

In this section we will describe some basic operations on the hierarchical datastructure, to be used by the van de Panne algorithm, and/or by the time-integration.

4.3.1 Jacobian determination

The determination of jacobians will be described in some more detail, being one of the most important operations performed by the algorithm.

It is assumed that in each phase of the solution algorithm all jacobians have the proper value, corresponding with the current valid segment of the piecewise linear system. If a pivoting is done by the algorithm, a number of jacobians should be recomputed. More precisely, if a pivoting is performed in a single leafcell, its jacobian has to be recomputed, as well as the jacobians of the compounds along a path upwards, starting at the leafcell, and ending at the root. If no changes are detected anymore, the update process upwards can terminate before reaching the root. In this case the pivot-operation isn't 'visible' at the terminal representation of higher level compounds. By exploiting this feature, a lot of computational effort can be saved, while simulating practical examples.

In order to determine the jacobian of a compound, a relative simple sparse $L/U$-decomposition is performed, restricted to the matrix-part corresponding with the variables $x_1 \ldots x_t$ (generally having more rows than columns). Before determining the $L/U$-decomposition, a row and column normalization is performed, to increase the numerical stability. In order to maintain sparsity, a rowcount is maintained, holding an estimation for the number of nonzero entries in each row. In each (pivot-)step of the $L/U$-decomposition a row is chosen with the lowest nonzero count and of at least
value one, and in this row the element with the largest absolute value is chosen as pivot. By examining the pivot-row and -column an estimation is done for the number of fill-ins in the remaining matrix part, and the row-count is updated accordingly.

If we denote the matrix part of (4.4) consisting of the intersection of the pivot-rows and the pivot-columns by $A_{22}$, we have found:

$$
\begin{bmatrix}
S_0 & S_1 & S_2 & S_3 & 0 \\
0 & J_1 & 0 & 0 & J_1 \\
0 & 0 & J_2 & 0 & J_2 \\
0 & 0 & 0 & J_3 & J_3
\end{bmatrix}
$$

maps to:

$$
\begin{bmatrix}
A_{11} & A_{12} & a_1 \\
A_{21} & A_{22} & a_2
\end{bmatrix}
$$

In which the mapping is defined by the appropriate row and column permutation and normalization. For the (square) matrix $A_{22}$ an L/U-decomposition is determined, so that $L$ is a lower triangular matrix, $U$ an upper triangular matrix and:

$$
A_{22} = L \cdot U
$$

Now the jacobian of (4.5) can be expressed as:

$$(J, b) \leftarrow (A_{11}, a_1) - A_{12} \cdot (L \cdot U)^{-1} \cdot (A_{21}, a_2)$$

Which is easily solved by fore- and back-substitution, and where '+' takes care of a de-normalization.

The L/U-decomposition created in this way, is lateron used (intentionally many times) to solve the internal x-vectors ($x_1, \ldots, x_t$), once the external x-vector ($x_0$) is known.

Besides the simplicity of the pivot-determination, it has the property, that if $A_{22}$ could be permuted to a lower triagonal form, this is recognized, yielding a stricktly diagonal $U$-matrix, a situation often occuring in for instance a higher-level system.
4.3.2 derivative determination

The pivot-type of algorithms reach the solution of the LCP by tracing a continuous path. Basically these paths are determined by computing the derivative of all the variables in the system, relative to a specified parameter. After these derivatives have been obtained, the stepsize is to be determined, for one straight linepiece of the path. To compute these derivatives we use the L/U decompositions as found in the jacobian generation, only new jacobian-source-vectors are to be determined.

Assume one particular leafcell to be described by:

\[
\begin{bmatrix}
0 \\
\nu
\end{bmatrix} = \begin{bmatrix}
A_{11} & A_{13} \\
A_{31} & A_{33}
\end{bmatrix} \begin{bmatrix}
x \\
i
\end{bmatrix} + \begin{bmatrix}
a_1 \\
a_3
\end{bmatrix} + \lambda \begin{bmatrix}
e_1 \\
e_3
\end{bmatrix}
\] 

(4.6)

We can write the derivative of this system relative to \( \lambda \) as:

\[
\begin{bmatrix}
0 \\
\nu
\end{bmatrix} = \begin{bmatrix}
A_{11} \\
A_{31}
\end{bmatrix} \begin{bmatrix}
x
\end{bmatrix} + \begin{bmatrix}
e_1 \\
e_3
\end{bmatrix}
\] 

(4.7)

with \( \bar{x} = \frac{\partial \nu}{\partial x} \) and \( \bar{v} = \frac{\partial \nu}{\partial \lambda} \)

Here we have made use of the property of pivoting algorithms, that \( i \) remains zero, or at least constant, and independent of \( x \).

In order to solve for the vector \( \bar{v} \) in this and other leafcells, we derive a jacobian \((J,\bar{v})\) of this leafcell as:

\( (J,\bar{v}) = (A_{11},e_1) \)

While computing the terminal representations \((J,\bar{v})\) upwards in the tree, the jacobians \( J \) remain unchanged, need not to be recomputed, and thus also the L/U-decompositions remain valid.

In all the other leafcells, \( \lambda \) is assumed to be inactive, and their direction-source-vector \( \bar{v} \) remains zero. By implicitly assuming \( \bar{v} \) to be zero, we need only recompute \((J,\bar{v})\) along a single path upwards, starting at the active
leafcell.
The computations along the path upwards will terminate, either by reaching the root of the hierarchy, or by finding a zero $\mathbf{x}$. In both cases, the direction-vectors $\bar{\mathbf{x}}$ can be solved in the last compound, and further downwards in its (sub-)tree. Again, if in some compound the terminal-variables $x_k$ of its $k$'th leafcell are determined as zero, the subtree corresponding with that leafcell need not be evaluated. In this way, the computations might be restricted to only a small number of cells of the hierarchy, which is indeed found in nearly all practical examples currently processed by the simulation program.

In all the leafcells reached in this way with a nonzero $\mathbf{x}$, the vector $\mathbf{v}$ can be determined by:

$$\begin{cases} 
\mathbf{v} = A_3 \bar{\mathbf{x}} + \mathbf{e}_2 \text{ in the active leafcell} \\
\mathbf{v} = A_3 \mathbf{x} \text{ in the other leafcells}
\end{cases} \quad (4.8)$$

After the all $\mathbf{v}$ are known, we can determine the stepsize $\lambda$ such that all $\mathbf{v}$ remain positive. We note that the inactive leafcells with $\mathbf{x}$ equal zero are not visited by the algorithm to compute $\mathbf{v}$, and are also not needed for evaluation for the stepsize.

If, as in the van de Panne and Lemke algorithms, also a variable $i_k$ of some leafcell can act as the current parameter, the same computations can be done, by replacing the vector $\mathbf{e}$ in (4.7) and (4.8) by the corresponding column of the matrix $A$ of the active leafcell.

4.4 The van de Panne algorithm

In this section we will present the van de Panne algorithm, as it is modified to operate on the hierarchical datastructure, mentioned in the previous sections.

One of the reasons for the modification of the algorithm, is induced by the difficulty that the element values of the matrix $M$ in equation (3.1) are not explicitly available. These values are needed by the algorithm to determine the nature of a pivot (i.e. positive, zero or negative). Theoretically, the matrix $M$ could be determined by concatenating all equations and variables of the hierarchy in a single large matrix, and then removing all variables $\mathbf{x}$ by linear programming operations. However, in this way a huge matrix is obtained, in which sparsity might be lost as a result of the pivoting operations, which is just the problem we are trying to avoid. This theoretical experiment is
useful to make clear that the element values of $M$, needed by the pivoting algorithms, are really different from the values found in the $A_{33}$ matrices of the leafcells. 

So if we want to investigate a matrix element say $M_{kk}$ as a candidate pivot, we determine the value of $\delta v_k/\delta i_k$, in the same way as described in section 4.3.2. If this value happens to be nonzero, as well as the corresponding element of $A_{33}$ in the proper leafcell, this element might be used as next pivot. If the value of $\delta v_k/\delta i_k$ equals zero, and the corresponding element in $A_{33}$ is nonzero, the van de Panne algorithm refuses this element as pivot. If a pivot operation on that element would be allowed, a degeneracy in a jacobian higher up in the hierarchy would appear, thus leading to an unsolvable set of equations. If the value of $\delta v_k/\delta i_k$ is nonzero, and the corresponding element in $A_{33}$ equals zero, the van de Panne algorithm will select this element as pivot. However we cannot perform this pivot-operation in our hierarchical datastructure. This appears to be a basic problem, and the program will terminate with an error message, if this situation occurs. (We were able to force the program into this situation, by applying a specifically designed leafcell.) In general this situation can be avoided by applying only leafcells which can be pivoted to all $2^n$ different states, for which $A_{33}$ in class $P$ is a sufficient condition. In practice this limitation isn’t too severe. Nearly all the cells currently in use have the identity-matrix as $A_{33}$, and more complicated mappings can be made by the interconnection of such cells.

Below the hierarchical form of the van de Panne algorithm is given. For more information about the algorithm we refer to section 3.4 and to [77].

1. Determine bottom-up the terminal-representation $(J,b)$ for each cell in the tree.
Solve top-down all $x$- and $v$-vectors, assuming $i$ equal zero. If in a leafcell $v$ happens to be $\geq 0$, mark this cell 'proper', otherwise 'improper'.
Proceed with step 2.

2. Choose a leafcell, say $K$, marked 'improper'.
If no such leafcell exists, the algorithm terminates, the solution is found.
Otherwise determine for this leafcell a real number $\lambda$ and vector $e$ such that:
$\lambda > 0$, $v = A_{31}x + \lambda e + a_3 \geq 0$.
Let $j$ denote the active variable $i_j$,
with $j=0$ denoting $i_0 = \lambda$. Set $j=0$ and
the direction of the active column at 'decreasing'.
Define the records of the transparent stack to contain
the identification of the active leafcell,
the number (j) of an active column, with associated
direction, and initialize the stack to contain a
single record with the current leafcell, active
variable and direction. Proceed with step 3.

3. Check for the unblocking of blocked rows:
Set \( d = 0 \) and \( \theta = \infty \).
Determine \( \partial v / \partial i_j \) by computing new
direction-source-vectors \( \delta \) along a path
upwards in the tree, and computing new \( \hat{x} \) in a
subtree downwards, as explained in section 4.3.2.
During these computations through the hierarchy,
maintain the values of \( d \) and \( \theta \) such that:
\( d \) - the number of unblocked rows determined by:
- \( d \) denotes a number of records in the stack,
  measured from the top.
  \( 0 \leq d \leq \) current size of the stack.
- if \( d > 0 \) and \( k \) is the column-number stored in the
  \( d \)'th record, then \( \partial v_k / \partial i_j \neq 0 \)
- There is no larger \( d \) for which the above
  conditions hold.
\( \theta \) - the maximum allowed stepsize, such that:
\( \theta = \max \{ \nu + \lambda \cdot (\partial v / \partial i_j) \geq 0 \} \)

in all 'proper' leafcells.
and 'active variable' + \( \theta \cdot 'direction' \geq 0 \).
If \( d \) remains zero, proceed with step 4. Otherwise,
keep the sign of unblocking - sign of \( \partial v_k / \partial i_j \)
for later use, and proceed with step 7.

4. No unblocking occurs, perform a step \( \theta \) indicated by
the current active variable \( i_j \) and its direction:
If no blocking was detected at all (\( \theta \) became \( \infty \)),
the algorithm terminates without finding a solution.
If a blocking was detected (\( \theta \) has a finite value),
perform the update:
\( x := x + \theta \cdot \hat{x} \)
\( v := v + \theta \cdot \hat{v} \)

4.29
\[ i_j := i_j + \delta \cdot \text{'direction'} \]

in all compounds with nonzero \( \bar{x} \),
in all 'proper' leafcells with nonzero \( \bar{\nu} \),
and in the active leafcell \( K \).

If \( i_j \) became zero and \( j=0 \), \( \lambda \) became zero,
this leafcell is solved, mark the cell 'proper',
and proceed with step 2.
If \( i_j \) became zero and \( j=0 \), proceed with step 5.
If \( v_k \) became zero for some \( k \), proceed with step 6.

5. The algorithm succeeded in reaching zero for an
active variable \( i_j \), not being \( \lambda \).
Remove the top-of-stack record, which described row \( j \)
as blocking row, from the stack by popping up
the stack one level.
Determine a new \( j \) (active variable \( i_j \)) with
the leafcell number \( K \) and associated direction
according to the new top-of-stack.
Proceed with step 3.

6. A new blocking row \( k \) is detected.
Choose a new \( j, K \) (active variable \( i_j \) in leafcell \( K \)),
with \( j - k \) and \( K = \) leafcell containing row \( k \).
Set the new direction to 'increase'.
Push a new record containing this \( j, K \) and direction
on the stack. Proceed with step 3.

7. An unblocking of \( d \) rows has been detected. \((d > 0)\)
Perform a diagonal block-pivot upon the rows and
columns, corresponding with the variable-numbers and
leafcells stored in the top \( d \) records on the stack.
Compute new jacobians along a path upwards from
the leafcells, as indicated in section 4.2.1.
Remove these \( d \) records from the stack.
If the unblocking occurred with a negative sign, change
the direction of move in the new top-of-stack record.
Choose a new \( j, K \) (active variable \( x_j \) in leafcell \( K \))
with associated direction according to the new
top-of-stack. Proceed with step 3.
4.5 Time integration and latency

Upto now nothing has been told about the integration over time, necessary to solve the transient-response of dynamic systems. The modifications needed to perform the integration happen to fit neatly in the previously described algorithms and datastructures, enabling us to postpone the description of the adopted integration-method.

The main goal of the integration method is to exploit latency-effects in the circuit, that is not to spend computational effort on those subcircuits that change very slowly. Thus fast changing subcircuits should not impose their small stepsizes on other 'sleeping' subcircuits. In this way, a large speed-up of the simulation of large structured systems can be obtained.

This goal is reached by assigning individually optimized stepsizes to all leafcells of the hierarchy. These stepsizes result in a modified terminal representation \((J,\delta)\), for computing the time-derivatives of all variables in the system. The actual \((J,\delta)\) depend upon the value of the stepsize and the adopted integration method. In order to ensure numerical stability, an implicit integration method is required. In our program (at the moment of writing this), the trapezoidal integration rule is adopted, whereas other implicit methods could be implemented. With the terminal representations \((J,\delta)\) known for all leafcells, all variables \(x\) in the system can be solved for, providing for an approximation of the time-derivative of \(x\), in the current time-interval. With these approximations for the time-derivatives of all the variables known, the actual allowed time-step can be determined in each leafcell, by an explicit determination of the timepoint (the so-called "nearest event") at which some boundary hyperplane of the piecewise linear mapping is reached. For solving the 'nearest event' (i.e. performing the necessary operations to proceed in time again), only a local update in a small subtree, activated by a single leafcell might suffice. By repeating this over time, parts of the circuit might not be 'visited' by the algorithm at all, thus implementing the latency exploitation. In the next sections this will be worked out in more detail.

4.5.1 The leafcell terminal representation

The first problem to be solved, is to determine a suitable terminal description \((J,\delta)\) which realizes updates (direction-vectors times stepsize) according to the trapezoidal integration rule.

Assume the set of vectors \((x_0, u_0, a_0, i_0, v_0)\) to satisfy the equations of a leafcell as defined by equation (4.1), at time-instance \(t=0\).

Assume the set of vectors \((x_1, u_1, a_1, i_1, v_1)\) to satisfy these equations at
Furthermore assume the sets of equations have been solved by a pivoting algorithm, leading to $i_0 - i_0 = 0$, and both solutions to be in the same segment of the piecewise linear mapping.

Now both sets of equations can be subtracted from each other to yield equation (4.9):

\[
\begin{align*}
0 &= A_{11} \dot{x} + A_{12} \dot{u} \\
\dot{u} &= A_{21} \dot{x} + A_{22} \dot{u} \\
\dot{v} &= A_{31} \dot{x} + A_{32} \dot{u}
\end{align*}
\]

with:

\[
\begin{align*}
\dot{x} &= (x_1 - x_0)/\Delta \\
\dot{u} &= (u_1 - u_0)/\Delta \\
\dot{v} &= (v_1 - v_0)/\Delta
\end{align*}
\]

The trapezoidal integration rule can be given by equation (4.10):

\[
\dot{u} = \frac{1}{2}(u_1 + u_0)
\]

(4.10)

By substituting (4.10) in (4.9b) and rephrasing the left-hand side, we obtain (4.11):

\[
(q_1 + q_0) - 2q_0 = \Delta A_{21} \dot{x} + \frac{1}{2} \Delta A_{22} (q_1 + q_0)
\]

(4.11)

Or equivalently (4.12):

\[
\frac{1}{2}(q_1 + q_0) - \frac{1}{2} \Delta (I - \frac{1}{2} \Delta A_{22})^{-1} A_{21} \dot{x} + (I - \frac{1}{2} \Delta A_{22})^{-1} q_0
\]

(4.12)

Where we have assumed $\Delta$ to have a value such that the inversion is feasible.

By substituting (4.12) and (4.10) in (4.9a) we obtain the terminal-description of the leafcell as equation (4.13):
\[
0 = J \dot{x} + b
\]
with:
\[
J = A_{11} + \frac{h}{2} \Delta A_{12} (I - \frac{1}{2} \Delta A_{22})^{-1} A_{21}
\]
\[
b = A_{12} (I - \frac{1}{2} \Delta A_{22})^{-1} \mathbf{0}
\]

If we determine the jacobians/sourcevectors \((J, b)\) in this way for all the leafcells, we can recompute bottom-up the jacobians and source-vectors for all the compounds. Afterwards we can determine all \(\dot{x}\) vectors in the hierarchy top-down as indicated in section 4.3.2.

Now we can find the directions, in which the other vectors in the leafcell propagate, once \(\dot{x}\) is known:

\[
\begin{align*}
\ddot{u} &= \frac{1}{2} \Delta (I - \frac{1}{2} \Delta A_{22})^{-1} A_{21} \dot{x} + (I - \frac{1}{2} \Delta A_{22})^{-1} \mathbf{0} \\
\ddot{a} &= A_{21} \dot{x} + A_{22} \ddot{u} \\
\ddot{v} &= A_{31} \dot{x} + A_{32} \ddot{u}
\end{align*}
\]

4.5.2 size of time steps

In the previous section we computed the directions (time derivatives) in which all variables in the system propagate. We now have to determine the maximum timestep, for which this propagation will be valid.

First of all the variable \(\Delta\) has to be determined, separately for each leafcell. At the moment this is done by explicitly determining (an approximation for) the second derivative of \(u\) versus time, in the previous section found as \(\ddot{u}\). The norm of this vector is easily obtained, and can be used to determine a timestep \(\Delta\), given an allowed absolute integration error \(\epsilon_a\) and/or an allowed relative integration error \(\epsilon_r\), according to equation (4.15):

\[
\Delta = \left( \frac{2 \text{err}}{||\ddot{u}||} \right)^{\frac{1}{2}} \text{ with } \text{err} = \epsilon_a + \epsilon_r \cdot ||u||
\]

The \(\Delta\) derived in this way, is used in the computation of the terminal description \((J, b)\) and has the interpretation of representing the maximum time-
interval in which \((J, b)\) is assumed to be an approximation of the external behaviour of the leafcell within the required accuracy. Of course this value determines the maximum allowed time-step for this leafcell, without an internal update. However, several different 'events' might occur, restricting the actual timestep to a smaller value.

Basically for each leafcell an event is set at the time-point \(t + \Delta\), with \(t\) the current time and \(\Delta\) the with respect to this leafcell allowed timestep. However, this event might be replaced by an earlier one, imposed by the restriction that \(v\) is to remain positive. With the known value of \(v\) at the current time-point, and the derived \(\dot{v}\) this event is easily determined. In this way the 'nearest event' is computed for each leafcell, and in each compound the 'nearest event' is set to the minimum value of the 'nearest events' of its subsystems. Furthermore in each compound a mark is placed, indicating which of its subsystems imposed this minimum value.

The integration algorithm has the task of repeatedly solving the 'nearest event' until finally the required time-interval, asked for by the user, has been solved. In solving this 'nearest event' two different cases are recognized:

1. The nearest event is imposed by reaching the end of the allowed integration-interval \(\Delta\).
   Assume \(K\) is the leafcell, causing this event. Within this leafcell, determine \(\overline{u}\) at the current time-point and choose a new stepsize \(\Delta\). Compute bottom-up along a path new source-vectors \(b\), and new jacobians \(J\) if \(\Delta\) changed. Starting in the cell, where the path ended, compute top-down in the subtree new direction-vectors \(x\), only in those parts where changes are detected. Update the \(x\)-vectors up to the current time-point with the old \(x\), before \(x\) is replaced by the new \(x\). In all the leafcells reached, update \(v\) and \(u\) up to the current time-point and determine the new \(\dot{v}\) and \(\Delta\). Determine the new events caused by the new \(\dot{v}\), and the new \(||\Delta||\) in these leafcells. Trace these events bottom-up in the hierarchy up to the root, in order to find the new nearest event.

2. The nearest event is imposed by a zero-crossing of some component of a \(v\)-vector.
   Assume \(K\) is the leafcell causing this event. Start the van de Panne algorithm from this leafcell, with an initialization according to:
And initialize the stack of blocking rows with two records: The top-of-stack record containing the leafcell K, the blocking row corresponding with the zero-crossing component of \( \nu \), and direction increasing; the record below containing the leafcell K, row-indication zero (for \( \lambda \)), and direction decreasing.

Initialized this way, the algorithm will terminate when \( \lambda \) reaches zero, having performed the necessary pivot operations to prevent \( \nu \) becoming zero at the current time-point.

Where necessary, the van de Panne algorithm must update the vectors \( x \) and \( \nu \) upto the current time-point. If pivot-operations are performed, the new jacobians must be computed in accordance with the implicit integration method as in equation (4.13), and must be accompanied with the new \( \delta \) vectors. Also new direction-vectors and events have to be determined top-down and bottom-up in the same way as described above. If a step is made by the algorithm in the direction of some \( i_k \) or \( \lambda \) of discrete (nonzero) size, extra precaution is to be taken. In this case the variables \( x \), \( \nu \) and \( u \) may change discontinuously with respect to time, and the integration is to be restarted in these leafcells again, which is done by forcing a new event in these leafcells at the current time-point.

As can be seen from the above description, no operations are involved that must affect the whole tree. So depending upon the behaviour of the particular circuit to be simulated, all the involved operations, might appear to be restricted to a small subset of the whole system. So a subcircuit in a quiescent state is possibly not affected (visited) by the algorithm at all, while other parts of the system are changing rapidly. Of course this type of behaviour heavily depends upon the particular system being analyzed, but is expected in the simulation of large digital circuits, which can be seen as a collection of relatively independent subcircuits (gates) with a relatively low interconnection rate. As found by others [82], the simulation of these circuits is enormously speeded up by exploiting this behaviour.
The traveling through the hierarchy is mainly controlled by the determination of 'change' during the computation of new jacobians bottom-up and the computation of new direction-vectors top-down. The use of floating point representations of all the numbers throughout the program, more or less enforces the use of some discrete error-criterion for the decision whether a change is detected or not. This allowed discretization error is a program parameter, to be supplied by the user.

For the stepsize control, the value of $||\dot{u}||$ is used, according to equation (4.15). However, for each iteration, the current value of $||\dot{u}||$ is evaluated to estimate the allowed time-step of the new integration-step, actually leading to a validity-interval for the new terminal-description. If other events occur during this interval, the actual performed time-step might turn out to be significantly shorter. However, we do not recompute the jacobians according this new (shorter) time-step and recompute the elapsed interval, but simply state that the adopted jacobian was a valid representation of the leaf cell, within the error-bound of the integration as allowed by equation (4.15).

In order to start-up the integration, the program starts with computing an initial solution of the circuit, by adopting fixed values for the vectors $u$, and adopting the van de Panne algorithm precisely as described in section 3.4. Actually the derived jacobians correspond with an explicit integration step, starting at time zero. If the nonlinearities of the circuit are solved in this way, the time-derivatives are obtained by computing bottom-up the direction-source-vectors $\delta$, according to an explicit method, actually corresponding with equation (4.13) with inserted value zero for the stepsize. After the values for $x$, $\dot{x}$ and $\ddot{x}$ are obtained, a decision can be made for a stepsize $\Delta$, and the above described integration method can start.

4.5.3 Numerical stability

As described in section 4.5.1 we adopt the implicit trapezium rule as integration method. However this method has been modified in that we initially only determine a direction $\hat{x}$ in which the time update is to be performed, and afterwards determine the actual size of the timestep in this direction. Thus in formulas:

Trapezium rule: $x_i = x_{i-1} + h \hat{x}$

with $\hat{x}$ depending upon $x_{i-1}$ and $h$
Contrary we actually perform:

\[ x_i = x_{i-1} + \frac{1}{k} \Delta t \]

where \( k \) represents some integer to indicate a shorter time step.

Due to this modification, we should check whether the stability properties of the trapezium rule still apply here.

In order to enable us to perform an analysis of the effects of this time step truncation, we restrict ourselves to a simple linear differential equation:

\[ \dot{x} = A x, \quad t \geq 0, \quad x_0 \text{ known} \]

For the determination of the discretization error it is sufficient to restrict ourselves to the one-dimensional case:

\[ \dot{x} = a x, \quad x_0 \text{ known} \]

The exact solution of \( 4.19 \) is easily determined as \( 4.20 \):

\[ x_t = x_0 e^{at} \]

or:

\[ x_{t+h} = x_t e^{ah} \]

Now \( 4.21 \) can be approximated by:

\[ x_{t+h} = x_t \left( 1 + ah + \frac{1}{2} a^2 h^2 + \frac{1}{6} a^3 h^3 + ... \right) \]

Equation \( 4.22 \) is to be compared with the approximate \( x_{t+h} \) as found with our integration method.
First derive \( \dot{x} \) as:

\[
\dot{x} = \frac{1}{h} (x_t + x_{t+h})
\]

\[
x_{t+h} = x_t + h \dot{x}
\]

Applying (4.19), this can be solved to yield:

\[
\dot{x} = \frac{a}{1-\frac{1}{2}ah} x_t
\]  \hspace{1cm} (4.23)

now

\[
x_{t+h/k} = x_t + \frac{1}{k} \dot{x}
\]

\[
= \frac{1+(2/(k-1))ah}{1-\frac{1}{2}ah} x_t
\]

For simplicity we will now assume that the algorithm performs \( k \) of these steps to finally find \( x_{t+h} \).

Thus:

\[
x_{t+h} = c^k x_t
\]

\[
c^k = \left( \frac{1+(2/(k-1))ah}{1-\frac{1}{2}ah} \right)^k
\]  \hspace{1cm} (4.24)

In series expansion this is written as:

\[
c^k = \begin{cases} 
1 + ah + \frac{1}{2} a^2 h^2 + \frac{1}{3} a^3 h^3 + \ldots & \text{for } k=1 \\
1 + ah + \frac{a^2 h^2}{2} + \ldots & \text{for } k \geq 2
\end{cases}
\]  \hspace{1cm} (4.25)

After comparison of (4.25) with (4.22) we find the discretization error:
Thus we can conclude that the second order convergence property of the trappzeum rule is lost by the truncation of timesteps, and a first order method (like the forward and backward Euler method) is obtained.

For the numerical stability of the method, we observe that the exact solution of (4.18) converges if and only if all the (complex) eigenvalues $\lambda$ of $A$ have a negative real part. The simulation leads to a converging sequence of $x_1$'s if and only if all the eigenvalues $\gamma$ of $C$ have a norm less than 1.

Now we can state (compare with 4.24):

\[ \gamma \text{ is eigenvalue of } C \Rightarrow \gamma^k \text{ is eigenvalue of } C^k \Rightarrow \gamma = \frac{1+(2/k-1)^k h\lambda}{1-h\lambda} \]

with $\lambda$ eigenvalue of $A$.

From (4.26) it can easily be proved that $\text{Re}(\lambda)<0 \Rightarrow ||\gamma||<1$, for all $h$. Thus we can conclude that the truncation of time-steps in the trappezium integration method, results in a stiffly stable integration.

To illustrate equation (4.26) figure 4.3 is added, showing the sets $\Lambda$ and $\Lambda^k$ defined as:

\[ \Lambda = \{ \gamma \mid \text{Re}(\lambda) = 0 \} \]
\[ \Lambda^k = \{ \gamma^k \mid \text{Re}(\lambda) = 0 \} \]

for $k = 1, 2, 3$ and 4.

All the values of $\gamma$ within the closed curves of figure 4.3 correspond with values of $\lambda$ for which $\text{Re}(\lambda)<0$. 

4.39
Figure 4.3. Stability regions of the integration method.
5. IMPLEMENTATION ASPECTS

After the description of the currently applied algorithms in the simulation program in the previous chapters, we will now focus on a few aspects of implementation. We will briefly denote the structure of different programs and datafiles, implemented as part of the simulation project. In a later section we will deal with the circuit definition.

5.1 Programs and datafiles

The simulation project is actually partitioned in a few different programs, each of them developed to solve a dedicated task. In this way, the programs retain a manageable size, and the concurrent development of the subtasks is easier, as well as the error recovery. By connecting these programs in a command-file at the operating system level, the user doesn't have to worry about the flow of the data through the different programs. The structure of the different programs and files is depicted in figure 5.1. Some of the programs are still in the development stage.

![Figure 5.1. The structure of programs and datafiles.](image)

The functions of the different programs are summarized in the list below:

- **NDL-compiler**
  - The NDL-compiler reads a user-supplied text-file containing the definition of the system to be analyzed, and generates a datafile (System-Data-File), containing a coded form of the circuit description, suitable for fast
handling and access. The system is to be described in a newly developed language (Network Description Language) featuring a powerful pascal-like syntax, extensive capabilities for hierarchical circuit definition, mixed-level system definition, and parameterization through multiple levels of the hierarchy. Furthermore not only a parameterization and interconnection of components can be described, but also new (parameterized) leafcells might be defined, containing (the matrices of) the actual piecewise linear models. While implementing these language capabilities, the compiler also features an extensive, user-friendly syntax error recovery. In a later section we will deal with the language in some more detail.

- Schematics capture program
The schematics capture program yields an alternative for the circuit definition. The program is operated interactively at a (color) graphic terminal, and allows the user to define his circuit in a graphical way. A hierarchical system is defined by placing components on the screen, drawing 'wires' for the interconnection of 'contacts' of components between each other, or with the outside world. In this way new systems are defined and stored in the library, which can be called for as components in a higher level of the hierarchy. The user can add a parameterization with the same capabilities as found in the compiler, and might define graphical symbols to represent new systems as components at a higher level.

- NDL-decompiler
This program is able to generate a text-file containing the network description language of a specified (set of) systems from the System-Data-File. Its main purpose is to allow for the portability of system-descriptions from one computer to another, which store the coded data-files in a different way, for instance if these systems were defined in a graphical way. Thus systems defined in a local workstation, can be ported to a bigger computer for the simulation of a large system. Furthermore this feature appears to be a good tool for documentation and library purposes. (Parts of the program still under development.)

- Linker
The linker is used to collect modules from different System-Data-Files, together to be stored in a new file. This feature is used in the handling and incorporation of libraries containing components and systems, called for by several users, and for the exploiting of the compiler feature of
Equation generation

The equation generation program, reads a System-Data-File, and builds the hierarchical datastructure, containing the datastructure used by the simulator program (Equations-File). For this task, the interconnection equations (S-matrix in eq.(4.4)) are generated, according to the Kirchhoff current and voltage law for electrical interconnections, and the actual matrices for the leafcells (as in eq.(4.1)) are derived, according to the current values of the parameters, which are evaluated in a top-down order.

The hierarchical structure is expanded to a physical present module (leafcell or compound) for each instance in the hierarchy, and a worst-case approximation is made for the storage-requirement for the sparse L/U-decompositions within each compound. Besides this Equations-File, also a textfile is generated, containing the relationship between the variable-numbers in this datastructure, and the user supplied names of modules and contacts (Varinfo-File), needed by the output-processor.

Simulator

The simulation program solves the piecewise linear equations and performs the transient-analysis according to the algorithms as described in chapter 4. The Equations-File is read by the simulator, and contains the datastructure describing the system to be analyzed. During the run of the program, this file is used for a virtual memory-implementation, backing up the excess storage, not fitting in the array-space declared in the simulation-program. The computed waveforms of all the terminal variables of a specified set of modules in the hierarchy are written to the so-called Signal-File.

Output processor

The output processor is used to obtain a graphical representation of the simulation results. The user can select a time interval and a set of signals (or linear combination of signals) to be drawn. Furthermore the user can ask for a time-interval relative to a triggerpoint, determined by the program, but defined by the user as a combination of conditions upon several signals. The program reads from both the Signal-File and the Varinfo-File in order to be able to present the signals, asked for by the user as pathnames through the hierarchy of (named) compounds, terminated with contact- or signal-names. (Still under development.)
6. MODEL EXAMPLES

In this chapter we will present a few piecewise linear models, in order to show the capabilities of the Linear Complementarity based description method. Most models in use are defined with an Identity-matrix as the LCP-matrix $A_{33}$, leading to a unique solution for these models, and thus always represent continuous functions. For a few more complicated mappings different matrices can be used, as can be seen in the examples of the logic-delay element, the analog-to-digital converter and the compact-square element.

6.1 Functions of one variable

Piecewise linear continuous functions of one variable are easily described with the matrix notation, given as equation (2.1). We will present a general example with the unit-matrix as D-submatrix in (2.1), and with one variable pair $v_k$, $i_k$ for each breaking point in the piecewise linear function.

Let

$$y = f(x) = a_0 x + b_0 + \sum_{k=1}^{n} a_k [x - b_k]_+$$

with $a_k$, $b_k$ ($0 \leq k \leq n$) given real numbers and:

$$[z]_+ = \begin{cases} 
0 & \text{if } z \leq 0 \\
 z & \text{if } z > 0 
\end{cases}$$

The matrix notation (2.1) of this function can be given as:

$$\begin{bmatrix}
y \\
v_1 \\
v_2 \\
\vdots \\
v_n 
\end{bmatrix} = \begin{bmatrix}
a_0 & a_1 & a_2 & \ldots & a_n \\
-1 & 1 & 0 & \ldots & 0 \\
-1 & 0 & 1 & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
-1 & 0 & \ldots & \ldots & 1 
\end{bmatrix} \begin{bmatrix}
x \\
i_1 \\
i_2 \\
\vdots \\
i_n 
\end{bmatrix} + \begin{bmatrix}
b_0 \\
b_1 \\
b_2 \\
\vdots \\
b_n 
\end{bmatrix}$$
6.2 A simple mosfet

An extreme simple mosfet model is derived according to the $I_{ds}(V_{ds})$ curves as depicted in figure 6.1:

These characteristics can be defined as a combination of four different linear mappings, each to be defined on its own domain. These mappings are listed below ($\beta$ and $V_t$ denoting parameters of the model):

The subthreshold region:

$I_{ds} = 0$ if $V_{g}-V_{t}-V_{s} \leq 0$ and $V_{g}-V_{t}-V_{d} \leq 0$

The forward saturation region:

$I_{ds} = \beta (V_{g}-V_{t}-V_{s})$ if $V_{g}-V_{t}-V_{s} \geq 0$ and $V_{g}-V_{t}-V_{d} \leq 0$

The backward saturation region:

$I_{ds} = -\beta (V_{g}-V_{t}-V_{d})$ if $V_{g}-V_{t}-V_{s} \leq 0$ and $V_{g}-V_{t}-V_{d} \geq 0$

The linear region:

$I_{ds} = \beta (V_{d}-V_{s})$ if $V_{g}-V_{t}-V_{s} \geq 0$ and $V_{g}-V_{t}-V_{d} \geq 0$

These equations are modelled in a matrix-description as given in equation 6.1:
For a practical relevant model, the above description can be improved by accounting for resistances in series with the mosfet terminals, and for a linear dependance of \( V_t \) as function of \( V_s \) and \( V_d \). Furthermore, by adding a single row \( \frac{d}{dt} \) and column \( u \), a gate-capacitance can be added. A more elaborate mosfet model, suitable for compact and efficient piecewise linear modelling, is also developed \[36\]. This model allows for accurate analysis of mos circuits, with a basic modelling accuracy of a few percent. The derivation \[36\] of this model is inserted as appendix 1 in this thesis. Both models are available in the simulation program, from a single leafcell-description by means of an additional parameter ("level") selecting between different matrix-definitions.

### 6.3 Compact square function

In this section a leafcell is presented which performs a mapping \( x \rightarrow y \), representing a piecewise linear approximation of the relation \( y = x \cdot x \), in a bounded domain. For this example we will use the regularity of this mapping for a compact modelling, in which an \( n \)-dimensional vector \( v \) and \( i \) return \( 2^n \) different segments for the piecewise linear mapping.

Below the equations are given for the mapping of \( x \rightarrow y \), in which the domain \( 0 \leq x \leq 16 \) is partitioned in 16 segments, by applying four-dimensional vectors \( v \) and \( i \).

\[
\begin{align*}
\begin{pmatrix}
1 & 2 & 6 & 14 & 30
\end{pmatrix}
\begin{pmatrix}
x
\end{pmatrix}
\end{align*}
\]

\[
\begin{pmatrix}
\begin{bmatrix} 1 & 2 & 2 & 2 \end{bmatrix} & \begin{bmatrix} 1 \end{bmatrix} & \begin{bmatrix} 2 \end{bmatrix}
\end{pmatrix}
\begin{pmatrix}
\begin{bmatrix} 1 & 2 & 2 & 2 \end{bmatrix} & \begin{bmatrix} 2 \end{bmatrix}
\end{pmatrix}
\]

We would like to note that the D-submatrix as defined in equation (2.1) in this
example belongs to the classes P, PSD and SCP but doesn’t belong to class PD.
(For the definition of these classes see section 3.1.)

6.4 Logic delay element

Next we will treat a small piecewise linear dynamic system, which is used to
delay the pass-through of logic signals. Thus the input-variable x is assumed
to has values of either 0 or 1, dependent upon time. For correct delay-
operation the value of the delay is to be smaller then the time between two
consecutive changes in x. With fast changing inputsignals x, the output y will
appear as a slow response, with about the same mean value over time. In the
equations below 6 represents the value of the time-delay.

\[
\begin{bmatrix}
y \\
u \\
v_1 \\
v_2
\end{bmatrix} = \begin{bmatrix}
-1 & 0 & 0 & 1 & 0 \\
0 & 1/6 & 0 & -1/6 & 0 \\
0 & 0 & -1 & -1 & 1 \\
0 & 0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
u \\
i_1 \\
i_2
\end{bmatrix} + \begin{bmatrix}
1 \\
0 \\
1 \\
1
\end{bmatrix}
\]

6.5 Analog to digital converter

Here we will present the model for a single-bit, static analog to digital
converter, which can be cascaded to form a converter for arbitrary length. The
input-signal x is assumed to have a value in the range from 0 to 1. The output
y is zero if \( x \leq 0.5 \), or y is one if \( x \geq 0.5 \). The output xout (carry out)
obtains the value \( 2 \cdot x - y \). By connecting this output xout to the input of another
module, the next, less significant, bit is generated.

\[
\begin{bmatrix}
y \\
xout \\
v_1 \\
v_2
\end{bmatrix} = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & -0.5 & 0 & i_1 \\
-1 & 0 & 1 & i_2 \\
0 & -1 & 0 & 0.5
\end{bmatrix}
\begin{bmatrix}
x \\
i_1 \\
i_2 \\
1
\end{bmatrix} + \begin{bmatrix}
0 \\
0 \\
0.5 \\
1
\end{bmatrix}
\]

6.6 And-or-invert gate

Finally we will treat the modelling of an and-or-invert gate. In MOS
technology this kind of gates is heavily used, leading towards compact logical
function realization. In general these functions can be realized as an
arbitrary network of MOS-transistors, of which all inputs are connected to
gates in the network, one node is connected to ground, and one node is connected to the source-side of a depletion-transistor of which the drain-side is connected to the supply-voltage. The (single) output of the gate is the node of the network connected to the depletion-transistor. Of course equivalent realizations can be made in dynamic NMOS or (dynamic) CMOS technologies. One example of a logic function will be given here.

\[ y = x_1 \cdot x_3 \cdot x_4 + x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_5 \]

Now this function can be modelled by a single matrix-description, in which the dimension of the \( i \) and \( v \) vectors is one plus the number of min-terms in the logical function (the number of min-terms is always less than or equal to the number of input-variables \( x \)).

For the above mentioned function the matrix is defined as:

\[
\begin{bmatrix}
y \\
v_1 \\
v_2 \\
v_3 \\
v_4
\end{bmatrix} =
\begin{bmatrix}
0 & 0 & 0 & 0 & -1 & -1 & -1 & 1 \\
1 & -1 & 0 & -1 & 0 & 1 & 0 & 0 \\
-1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\
-1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & -1 & -1 & 1
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
i_1 \\
i_2 \\
i_3 \\
i_4
\end{bmatrix}
\]

7. CONCLUSION

In the previous chapters theoretical backgrounds as well as implementation aspects were treated covering a newly developed simulation program, being totally different from any other simulator currently on the market or under construction. The basic concepts making this simulator a unique program are:

- Piecewise linear models are used to implement all nonlinear components.

- An extremely powerful algorithm, with respect to global convergence properties, is implemented to solve the piecewise linear equations.
- The (functional) hierarchical decomposition of the system, as given by the user, is maintained throughout the program, by applying hierarchical data structures and accordingly modified algorithms. By explicitly detecting changes during the analysis, the hierarchical structure is exploited for skipping of unnecessary operations, thus implementing latency. This feature is strongly effectuated by applying an integration scheme with individually optimized timesteps for all leaf cells.

- Due to the implementation of special features in the system definition phase, (i.e., different types of interconnect), not only a new field of problems in electrical engineering can be analyzed (mixed level simulation), but also different (non-electrical) systems are analyzed as easily, such as for instance control- and feedback- systems, economics, biomedical systems or network flow problems.

- Whereas the actual piecewise linear models are not a part of the simulation program, these models are added separately (by the linker program), allowing for maximum simulation flexibility in a wide range of applications.

- Due to the piecewise linear modelling and a carefully implemented implicit integration method, an explicit stepsize control mechanism is obtained. In conventional Newton-Raphson based circuit simulation programs, the Newton iteration often doesn't show convergence, mainly because strong nonlinearities are crossed with a too large stepsize. A (repeated) recomputation with smaller timesteps is required in those situations. Contrary in our piecewise linear simulator the allowed timestep up to the boundary of the nearest nonlinearity is easily and explicitly obtained.

Upto the time of writing this, we have had about one year to test and evaluate the simulation program. So far the program showed a trouble-free simulation of analog, digital and mixed-level circuits, either with or without delay modelled inside the components. Upto now only couple of very special circuits are encountered, which couldn't be solved by the simulator (assumed no errors are made in the modelling of the components). Whereas the simulator couldn't solve the circuit, this was caused by flaws in the implementation of the algorithm, to be corrected in the version currently under development.

In order to be able to test the applied algorithms and their interaction in an early phase, we tried to have a running program as soon as possible, operating
without additional difficulties. For these reasons in several parts of the program simple, robust and straightforward methods were applied, without emphasizing on computational efficiency. So besides optimizing the algorithms itself, a considerable speedup can still be obtained by replacing small parts of the program with more elaborate substitutes. As example I would indicate determining an optimal order of evaluation of subsystems (at the moment not done at all), or applying some incremental update or partial recomputation scheme of the (sparse) L/U-decomposition in each cell of the hierarchy (at the moment the L/U-decomposition and normalization are recomputed as a whole after each minor change). As a consequence the current version (december '85) doesn't show the speed, as could be expected. Whereas the program maintains statistics about the computational effort spent in several parts, and the event-activity as exploited by the algorithms, valuable information is obtained to evaluate the different choices of algorithms and implementation aspects.

Based upon the experience with this program, a new version is under development. Many features of the old program remained, whereas they appeared to be highly successful. Among them are latency evaluation guided by the hierarchical structuring, individually optimized timesteps for each leafcell, matrix definitions for the piecewise linear models, and the Van De Panne algorithm to solve the LCP.

However several important modifications are made, leading to a highly improved efficiency:

- The datastructure to store the equations is changed, in order to reduce (eliminate) the overhead imposed by the hierarchy. In the new version the hierarchical structuring becomes less explicit, and jacobians are no longer needed (derived) to solve the set of equations. Furthermore the size of this set is reduced by applying a node-voltage based equation generation, instead of the full tableau approach currently in use.

- A rank one update strategy for the L/U-decomposition is implemented, resulting in a very fast update scheme, perfectly matching the pivot operations in the leafcells.

- A minimum time-delay between two successive events is imposed, leading to a concurrent update for multiple changes, and considerable less overhead when many events are generated in a small time interval. Especially for analog circuits with elaborate component models, the efficiency can be raised this way.
J.T.J. van Eijndhoven was born March 1, 1957 in Roosendaal (NB), The Netherlands. He studied electrical engineering at the Eindhoven University of Technology, Eindhoven, The Netherlands, where he obtained his masters degree cum laude in May 1981. From May 1981 up to September 1983 he worked as PhD student at the Eindhoven University of Technology, department of Electrical Engineering, in a CAD research group. Based on this thesis, he hopes to obtain his PhD degree in December 1984. Besides his work on simulation methods, he was interested in IC layout design and computer hardware. Since September 1983 he is working as senior research member of the scientific staff in the same group.
REFERENCES


4.52


[34] Eijndhoven, J.T.J. van; Jess, J.A.G.; "The solution of large piecewise linear systems"; Proc. Int. Symp. on Circuits And Systems, Rome, Italy, pp.597-600, 10-12 may 1982


[38] Eijndhoven, J.T.J. van; Jess, J.A.G.; "Mixed mode mixed level analysis with PWL systems"; proc Int. Symp. on Circuits And Systems, Montreal Canada, pp.1377-1380, 7-10 may 1984


[57] Laan, G. Van der; Talman, A.J.J.; "Convergence and properties of recent variable dimension algorithms"; in: num. sol. of highly nonlin. problems, w.forster(ed), pp.3-36, 1980

[58] Laan, G. van der; "Simplicial fixed point algorithms"; Amsterdam: Mathematisch Centrum, Mathematical Centre Tracts 129, 1980


4.55
[60] Lemke, C.E.; "Bimatrix equilibrium points and mathematical programming"; management science, vol. 11, no. 7, pp. 681-689, mar. 1965


[63] Lineback, J.R.; "Logic simulation speeded with new hardware"; Electronics, vol. 55, no. 12, pp. 45-46 june 1982


[73] Newton, A.R.; "Techniques for the simulation of large scale integrated circuits"; IEEE trans. on circ. and syst., vol. cas-26, no. 9, pp 741-749, sep. 1979

4.56


4.57
[88] Saigal, R.; "On the convergence rate of algorithms for solving equations that are based on methods of complementary pivoting"; math. of operations research, vol.2, no.2, pp.108-124, may 1977


[93] Tai, H.M.; Saeks, R.; "Parallel processing for nonlinear circuit simulation"; proceedings Int. Symp. on Circuits And Systems, Newport Beach, California, pp.674-676, 2-4 may 1983

[94] Talman, A.J.J.; "Variable dimension fixed point algorithms and triangulations"; Mathematical Centre Tracts 128, Mathematical Centre, Amsterdam, The Netherlands, 1980


[98] Vlach, M.; "L/U decomposition and forward-backward substitution of recursive bordered block diagonal matrices"; proceedings Int. Symp. on Circuits And Systems, Newport Beach, California, pp.427-430, 2-4 may 1983

[99] Vlach, M.; "Block relaxation in the Kaczanelson algorithm"; proc. Int. Symp. on Circuits And Systems, Montreal, Canada, pp.551-554, 7-10 may 1984


4.58


[104] Yu, Q.; Wing, O.; "PLMAP, a piecewise linear MOS circuit analysis program"; proc Int. Symp. on Circuits And Systems, Montreal, Canada, pp.530-533, 7-10 may 1984
This is the reference manual for NDML.

NDML is the network description language for a Piecewise Linear Simulator. The network description part of the language provides general constructs for expressing hierarchical network structure. The modelling part of the language allows the circuit primitives to be described by means of a set of piecewise-linear equations and ordinary first order differential equations. Both ways of description, compounds and leafcells, share a common interface construct. In this way the implementation information of a circuit template is hidden.

December 1985

1. INTRODUCTION

This report describes the Network Description and Modelling Language NDML.

The network description part of the language provides general constructs for expressing hierarchical network structure, i.e. system types called templates may be defined by interconnecting a number of more simpler circuits. The language provides a simple mechanism for parameterising circuit behaviour. A structural description of a circuit is called a compound template or compound for short.

The modelling part of the language allows the circuit primitives to be described by means of a set of piecewise-linear equations and ordinary first order differential equations. Circuit primitives thus modelled are called leafcell templates or leafcells for short.

Both ways of description, compounds and leafcells, share a common interface construct. In this way the implementation information of a circuit template is hidden.

The kind of circuit that can be described solely depends on the descriptive power of the leafcells (structural description can not introduce new behaviour).

This paper was produced as part of Task IV of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
1.1 Design Goals

The primary goal for NDML is to provide the Eindhoven Piecewise-linear simulator with a simple and small user-interface for describing network input data. Simple in the sense that the language should both be easy to learn and easy to implement. This is achieved by carefully designing the syntax, including suggestive reserved words and using similar constructs to indicate similar semantics.

Clearly the second aspect, "smallness", also directly contributes to ease of learning and implementation. The language deliberately excludes many of the fancy constructs found in comparative circuit and hardware description languages. We feel that most of these fancy features are scarcely used and their exclusion is justified as long as other more primitive constructs are present that can be used to achieve the same result.

The step-by-step design process of NDML is described in [Ruehli]. Similar ideas on a structure description language can be found in [BDL].

1.2 Language Summary

A NDML circuit description is composed of one or more source text files that can be separately compiled. Each file consists of one or more system definitions. System definitions can either be compound definitions or leafcell definitions. The latter kind of system definition is normally not supplied by a user. A leafcell definition describes the behaviour of a piece of hardware by means of a piecewise-linear model. These models will normally be prepared by a qualified person(s) and collected in libraries. Compound definitions basically allow for net-list specifications.

A compound definition specifies a template for a hardware module by specifying the interconnection between the sub-systems it is built of. Compound systems can then be used as sub-system in other compound definitions. The use of a compound inside another is called instantiation. A sub-system is said to be an instance derived from a template.

A compound system definition starts with an interface specification. Here a name for the template is introduced and its terminal ports are named and typed. Optionally, parameters can be associated with this definition.

Terminal ports represent the physical contacts of a hardware component. Often they are called pins.
Parameters are convenient to model certain varying quantities of the system. Parameters can only take numeric values.

To describe the 'contents' of a compound, instance declarations are required that state the number and kind of sub-systems. At this point parameter values for the instances must be given, either explicitly by means of assigning a value obtained by evaluating an expression, or implicitly by the default mechanism.

The connections between the pins of the instance (sub-) systems is achieved by means of nets. A net is said to connect a number of terminal ports. Nets are denoted by names and declared just prior to the connection statements. A connection statement associates the terminal ports of an instance with nets.

1.3 Notation

This manual presents the language concepts in a bottom-up fashion, i.e. first the basic character set (alphabet) is defined, then the lexical rules that guide the formation of elementary language symbols (words) are discussed and finally the syntax and semantics is presented per basic concept. To define the syntax an extended Backus Naur Form is used. The EBNF we use closely resembles the proposal in [Wirth] also known as Wirth's Syntax Notation. However our extensions are specially made to allow for automatic processing, e.g. to determine the type of grammar and to generate nice syntax diagrams. The Extended BNF meta-language we use to describe the syntax can be defined in itself:

```
<syntax> ::= { <production-rule> } .
<production-rule> ::= <nonterminal-symbol> " ::= " <expression> "." .
<expression> ::= <term> { " | " <term> } .
<term> ::= ( <factor> )+ .
.factor ::= <nonterminal-symbol> <term> ( " | " <expression> ) .
<factor ::= ( " <expression> " ) ) " | " [ " + " ] .
```

The meta-symbols have the following meaning:

- $T_1 \mid T_2$ denotes a choice between either $T_1$ or $T_2$,
- $[ \ E \ ]$ denotes an optional construct $E$,
- $( \ E \ )$ denotes repetition of $E$ zero or more times,
- $( \ E \ )^+$ is equivalent to $E \{ \ E \}$ and thus denotes a repetition of one or more times,
(E) parenthesis can be used to override the operator precedence, concatenation of factors is denoted by juxtaposition.

To unambiguously denote a literal which usually is called a terminal symbol, it is written between double-quote characters. For clarity the syntactic variables or non-terminal symbols stand out by enclosing them in angular brackets ("<" and ">").

1.4 Classification of Errors

The language recognises three categories of errors.

1. Errors that must be detected at compilation time. These errors correspond to any violation of a rule of the language, other than stated below. All lexical and syntactic errors belong to this class. In this manual this category is implied by the use of terms such as "legal", "must", "allowed", or "may only".

2. Errors that must be detected at evaluation time. These errors result from violations against language rules that are outside the scope of a compiler, e.g. errors during expression evaluation, cross-checking separately compiled modules etc.

3. Errors that result from violations against the pragmatics of the language. The language allows for certain circuit descriptions that have no physical counterpart, like a circuit with only one electrical contact. Also violations against certain connection rules belong to this class. These kinds of errors need not necessarily be detected by a compiler, but if so the error should be reported as a warning. This manual uses the word erroneous to refer to this category of errors. If a description that contains erroneous statements is evaluated and/or simulated, its effect is undefined and unpredictable unless stated otherwise.

4.63
2. LEXICAL ELEMENTS

This chapter defines the lexical elements of the language. Rules will be presented that determine the creation of the symbols of the syntactic level from individual characters. Furthermore the use of text formatting characters like space, TAB and new-lines and the use of comments will be explained.

2.1 Character Set

All language text may be represented with a basic graphic character set, which is subdivided as follows:

(a) upper case letters

\[
\]

(b) digits

\[
\text{<digit> ::= '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'.}
\]

(c) special characters

\[
\text{<special-character> ::= '!', '\&', '(', ',', '.', '!', '/', '}', '&', '@', '-', '#', '$', '~', '@', '\', '^', '_'.}
\]

(d) the space character

\[
\text{<space-character> ::= ' ', ' ', '_', ',', '!', '/'.}
\]

The character set may be extended to include further characters from the 95 character ASCII graphics set. These are:

(e) lower case letters

\[
\text{<lowercase-letter> ::= 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'.}
\]

(f) other special characters

\[
\text{<other-special-character> ::= '!', '\&', '(', ',', '.', '!', '/', '}', '&', '@', '-', '#', '$', '~', '@', '\', '^', '_'.}
\]

Any lower case letter is equivalent to the corresponding upper case letter, apart from their occurrence in so-called escaped identifiers (see section 2.3.1). Other special characters can only appear in comments, in the extension construct and in escaped identifiers.
2.2 Lexical Units and Spacing Conventions

A program is a sequence of lexical units; the partitioning of the sequence into lines and the spacing between lexical units does not affect the meaning of the program. The lexical units are identifiers (including reserved words and predefined attribute keywords), numeric literals, delimiters and comments. A delimiter is either one of the following special characters:

```
<one-character-symbol> ::= 
  "!" "&" "('" "('" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "('<" "(<
Note that identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

Examples:

<table>
<thead>
<tr>
<th>names</th>
<th>reserved words</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>system</td>
<td>input</td>
</tr>
<tr>
<td>bool_and2</td>
<td>net</td>
<td>unused</td>
</tr>
<tr>
<td>NDML</td>
<td>begin</td>
<td>inout</td>
</tr>
</tbody>
</table>

### 2.3.1 Escaped Identifiers

To allow for ease of transcription from a certain foreign network language into NDML and to support automatic generation of NDML texts, the notion of escaped identifier is introduced. An escaped identifier is an identifier written with special lexical syntax in order to allow for the inclusion of characters not allowed in ordinary identifiers. Escaping also overrides the interpretation of the identifier as a reserved word. An escaped identifier has an appearance much like string literals have in languages such as C and Pascal. Ordinary users are not recommended to use this feature.

\[
<\text{escaped-identifier}> ::= "\" ( <\text{escaped-character}> )+ "\" .
\]

\[
<\text{escaped-character}> ::= <\text{double-quote-image}> | <\text{letter}> | <\text{special-character-except-double-quote-and-blank}> .
\]

\[
<\text{double-quote-image}> ::= <\text{double-quote-character}> <\text{double-quote-character}> .
\]

\[
<\text{double-quote-character}> ::= "\" .
\]

All letters appearing in an escaped identifier are recorded case-distinct, i.e. "NAND_2" is different from "nand_2". When using both escaped and normal identifiers in one text, it is undefined whether some of them will denote the same object. As is specified for all lexical units, escaped identifiers must fit on a line. To include a double-quote character in an escaped identifier it must be written twice. Formatting characters, e.g. a horizontal tab, are disallowed.

Examples:

"bool_and2"
"3-input$NAND"
"don't-care"
"NET-123"
"begin"

### 2.4 Reserved Words

The identifiers listed below are called reserved words and are reserved for special significance in the language. Declared identifiers may not be reserved.
BEGIN ELECTR IF PL SYSTEM
CASE ELSE INSTANCE REFERENCE TAN*
CONNECT* END LOG* REMOVE THEN
COS* EXP* NET SIGNAL U
DEFAULT EXTERNAL OTHERWISE SIN* VAR
DU FI PIVOT SQRT* ZERO

of these the following can only appear in compound system definitions:

CONNECT* INSTANCE NET

whereas,

CASE FI IF PL PIVOT REMOVar NETWORK U
DU IF PL PIVOT NETWORK U

can only appear in leafcell system definitions.

The reserved words marked with * are likely to disappear in the future. The arithmetic function names are for now considered reserved words for ease of implementation.

Note that upper and lower case spellings of a reserved word are not distinct.

2.5 Numeric Literals

There are two classes of numeric literals: integer literals and real literals.

<nrumeric-literal> ::= <unsigned-integer-number-literal>  
| <unsigned-real-number-literal>  

<unsigned-integer-number-literal> ::= <digit-sequence>  
| <digit-sequence> <scale-factor>  

<digit-sequence> ::= <digit> { <digit> }  

<unsigned-real-number-literal> ::=<digit-sequence> <scale-factor>  
| <digit-sequence> <scientific-scale-factor> <digit-sequence>  
| <digit-sequence> <fractional-part> <scale-factor> <digit-sequence>  

<fractional-part> ::= "." <digit-sequence>  

<scale-factor> ::= ( "E" | "e" ) <exponent>  
| <scientific-scale-factor>  

<exponent> ::= [ "+" | "-" ] <digit-sequence>  

<scientific-scale-factor> ::=<Giga> | <Mega> | <kilo> | <milli>  
| <micro> | <nano> | <pico> | <femto>  

<Giga> ::= "G" | "g"  
<Mega> ::= "M" | "m"  
<kilo> ::= "k" | "K"  
<milli> ::= ( "M" | "m" ) ( "L" | "I" )  
| <nano> ::= "n" | "N"  
| <pico> ::= "p" | "P"  
| <femto> ::= "F" | "f"  

(* 10E+9 *)  
(* 10E+6 *)  
(* 10E+3 *)  
(* 10E-3 *)  
(* 10E-6 *)  
(* 10E-9 *)  
(* 10E-12 *)  
(* 10E-15 *)
Examples:

0 123  32767  (* integer *)
0.0 0.0123 12345.0  (* real *)
123E-1 0.0123e12 0.3E+6  (* real with exponent *)
123K 0.1ml 1.23u  (* real with scale factor *)
1k3 1ul 100M23  (* real *)

2.6 Comments

Comments may appear before and after any lexical unit. They have the same
delimiting effect as a space. Comments have no effect on the meaning of the
program.

<comment> ::= "(*" <comment-string> ")"
<comment-string> ::= ( <a-printable-ASCII-character> )
<a-printable-ASCII-character> ::= <letter> | <digit> | <TAB> | <EOL>
| <special-character> | <other-special-character>

Comments may be nested !!!

Examples:

(* This is a comment *)

(* Comments may extent passed
the end of a line *)

(* This is an example of a (* nested comment *) *)

2.7 Extension Mechanism

The extension construct provides a well-defined escape possibility out of the
language, i.e. any text within curly braces is not considered part of the
language and will typically be treated just like comment. It allows different
language processors to share a common program description where some processor
may define and use the text in the extension construct.

<extension-construct> ::= "{" ( <a-printable-ASCII-character-
except-right-curly-brace> ) "}"

Extension constructs may not be nested. Where extension constructs may or
should appear in the syntax is not defined here. Also the syntax and semantics
of the contents of an extension is not defined.

To date, NDML defines one extension construct namely the attribute list in a
formal contact section.

4.68
2.8 Attributes

Attributes are identifiers that when they appear in a language defined extension construct get a special meaning. Attributes are not reserved. See the section on terminal ports for the attributes defined for them.

2.9 Compiler Directives

Compiler directives are used to control the compilation process. A compiler may define options that can be altered by a user. Compiler directives can only appear in an extension construct and their syntax and effect is implementation defined.

Examples (implementation defined):

```
{ directive list off }          (\$L-)
{ directive xref }             (\$L-)
```
3. OBJECTS AND DECLARATIONS

This chapter describes the classes of objects in the language, how they must be declared and what types can be associated with them.

3.1 Objects in Circuit Description

Objects are abstract entities that can be identified and manipulated in some way. A class defines general characteristics shared by all objects of that class. The language defines five classes of objects for circuit description:

1. system templates or definitions
2. system instances, derived from templates
3. terminal ports
4. interconnection nets
5. parameters

These classes are related. This will become clear in later sections.

3.2 Declarations

A declaration associates an identifier (name) with an object. Each object must be explicitly declared before it can be used. There is an exception to this rule where some objects are declared implicitly solely for user convenience, see the section on implied nets below.

The process by which a declaration achieves its effect is called the elaboration of the declaration. In general the following successive actions are involved:

1. First, the identifier for a certain new object is introduced at a point called the defining point or first occurrence of the named object. The new identifier may hide previously declared identifiers from then on. (The rules for scope and visibility are defined in section 5).

2. Then the particular class of the object and possible special characteristics for this object are determined. The declared identifier may only be used to name the object after this elaboration has completed. This rule excludes recursive object declarations.

The region of text over which the declaration has an effect is called the scope.
of the declaration; the region starts at the point of first occurrence of the associated identifier and if not sooner, ends at the end of the program text.

A typical declaration is of the form:

<identifier> " : " <particular-characteristics> .

Terminal port, parameter, instance and net declaration are described here. Treatment of system definitions is deferred to section 6.

3.3 Terminal Ports

A terminal port or terminal for short can be visualised as a contact at the perimeter of a system. Terminals constitute part of the interface of this circuit with its environment (neighbouring circuits). Signal data is visualised to pass through a terminal. A terminal has as its value a function of time. The range of this function is called the type of the terminal port. Clearly, signal types and terminal types must agree.

Terminals are declared in the heading of a system definition and are thus said to be formal terminal ports.


A terminal declaration introduces one or more terminal ports of a given type and optionally with a given attribute list and optional default value.

Let A be an instance of an attribute list, I1, I2, ... In an instance of the identifier list, T an instance of the terminal type and D an instance of a default value then,

A I1, I2, ... In : T D

is semantically equivalent with

A I1 : T D ;
A I2 : T D ;
A In : T D

3.3.1 Terminal Attributes

appear in a language defined extension construct. Attributes are not essential from a simulator point of view. Therefore they are considered not to belong to the core of the description language. Attributes are a means to enforce particular connection rules that assist the user in verifying the correctness of his description.
The meaning and use of terminal attributes will be described in a separate section.

3.3.2 Terminal Default Values

specify a connection of the terminal to a certain constant source system. Thus the terminal value becomes a constant function in time. Terminal default values are effectuated at compile-time. When this effectuation takes place is yet unspecified, but is related with the connection rules associated with the use of certain terminal attributes.

3.3.3 Implied Nets

Each terminal port declaration implicitly implies a net declaration. See section 3.6.

Examples of terminal declarations:

(input) in, data : signal
(output) output : signal
v_ref : electr
node1 : electr

3.4 Parameters

System definitions may be parameterised. Parameters allow the behaviour of a system to be altered upon instantiation. Formal parameter declarations are an optional part of the interface of a system definition. At evaluation time, parameter values are obtained by evaluation of expressions. At simulation time, parameter values can be considered as constants.
A value-range-specification states that the value of the parameter must always lie within or be equal to the specified boundary numbers. The bounds must be numeric literals optionally proceeded by a sign, where the lower-bound must be smaller or equal to the upper-bound. A no-bound-denoter specifies a one-sided open interval of values.

If no value-range-specification is present in a parameter declaration then it is assumed that the parameter may take any real number as its value. This is equivalent to ": " .. " .

The default-value-specification specifies a default value for the parameter. The default value must be a legal value for the parameter.

Let \( I_1, I_2, \ldots I_n \) be an instance of the identifier list and \( TD \) an instance of the type and default specification, then

\[
I_1, I_2, \ldots I_n \text{ in } TD
\]

is semantically equivalent with

\[
I_1 \text{ in } TD ; \\
I_2 \text{ in } TD ; \\
I_n \text{ in } TD
\]

Examples of parameter declarations:

- \( \text{amplitude : 0.0 .. * default 1} \)
- \( \text{v_init : default 0.0} \)
- \( \text{required\_parameter} \)

3.5 System Instances

System instance declarations appear in system definitions. They constitute the structural components of the system, i.e. they specify the subsystems of which
the system being defined is built of. A system instance is said to be derived
of a system definition or template. A system instance cannot be derived from
the system containing the declaration, i.e. recursive instantiation is not
allowed.

Part of the declaration forms the association of values to the formal
parameters of the template system. These values are called the actual
parameters of the instance. Association of actual with formal parameters is
done by means of parameter assignment statements. This approach is often
referred to as named parameter association.

The identifier appearing on the left-hand side of a parameter assignment must
be the identifier of a formal parameter of the template system, or a unique
abbreviation thereof. It is an error otherwise. An abbreviation is a declared
identifier with some characters omitted from the right. Unique means that the
abbreviation does not match more than 1 formal parameter identifier of the
corresponding template. Abbreviation is introduced for user convenience; it
suggests that library systems are defined using self-explanatory and therefore
probably lengthy formal parameter names.

The expression must evaluate to a numeric value, i.e. it must be either of type
integer or real. An integer value is always automatically coerced to a real
before it is assigned to the formal parameter. The value obtained must be
within the range specified for the formal parameter.

If a system template specifies a default value for a parameter, then the
parameter may be omitted from the actual parameter part, i.e. there need not be
an assignment statement with the identifier for that parameter appearing on the
left-hand side. Otherwise it will be an error if the number of actual
parameters does not agree with the number of formal parameters.

Naming rules (see section 5) allow instances to be identified with the same
identifier as the template system is identified. Therefore the identifier list in a instance declaration is optional.

Let \( I_1, I_2, \ldots \) stand for some identifier list, \( T \) for some template name and \( AP \) is the actual parameter part (possibly empty), then

\[
I_1, I_2, \ldots : T \ AP
\]

is semantically equivalent with

\[
I_1 : T \ AP ;
\]

\[
I_2 : T \ AP ;
\]

\[
I_n : T \ AP
\]

When one chooses to leave out the identifier list and the subsequent colon, the following declaration,

\[
T \ AP
\]

is semantically equivalent with

\[
T : T \ AP
\]

Examples of instance declarations:

and

\[
gen \begin{aligned}
&: \text{sine_generator} \\
&\left<\text{ampl} := 10, \text{frequency} := 2k\right> \\
&\text{or1, or2, or3 : bool_or2}
\end{aligned}
\]

3.6 Nets

Nets constitute the medium through which signals flow. They are a means by which connections of terminals can be specified.

Nets are explicitly declared in the net declaration part of a system definition:

\[
\text{<net-declaration> ::= <identifier-list> "::" <signal-type> .}
\]

An implicit net declaration is implied by each terminal port declaration. Such a net inherits both the identifier and the type of the declared terminal.

Let \( I_1, I_2, \ldots \) be an instance of the identifier list and \( T \) an instance of the net type, then

\[
I_1, I_2, \ldots : T
\]

is semantically equivalent with

\[
I_1 : T ;
\]

\[
I_2 : T ;
\]

\[
I_n : T
\]
Examples of net declarations:

\begin{verbatim}
and1_out, and2_out : signal
node1, node2 : elect
\end{verbatim}
4. NAMES AND EXPRESSIONS

This section defines the rules for constructing names and expressions.

4.1 Names

Names denote declared objects. A name constitutes a way of accessing the value of the object, i.e. it specifies an access path to the object.

\[ \text{name} ::= \text{identifier} \]

As yet there are no other forms of names defined. The distinction between name and identifier is explicitly made for congruence of terminology with for instance programming languages and to anticipate future language extensions that require more extended forms of names.

A name will always be constructed from an identifier. The identifier must be a so called applied occurrence of an identifier. See the section on scope and visibility.

Future extensions will include qualified identification of the terminals of a system template, and terminal ports that are structured as a one-dimensional array.

4.2 Expressions

An expression is a formula that specifies the computation of a value. An expression is constructed by applying operators to operands. The type of the operand(s) determines the allowable operators. Operands can themselves be expressions.

In compound system templates the only place where an expression may occur is as the actual parameter in a parameter assignment. The only non-constant primitive operands allowed are the names of formal parameters. Actual parameters must be either of type integer or of type real. The following only presents that part of the expression syntax as applicable to the case stated above.
<numeric-expression> ::- <simple-expression> .
<simple-expression> ::-

<unary-operator> 1 <term>
( <aading-operator> <term> ) .

<unary-operator> ::<unary-minus-operator>
I <unary-plus-operator>
<unary-minus-operator> .. <unary-plus-operator> ::= "+"
<adding-operator> ::<plus-operator>
I <minus-operator>
<plus -operator> :: = "+"
<minus-operator> ::- "-"
<term> ::- <factor> ( <multiplying-operator> <factor> ) .
<multiplying-operator> :: <times-operator>
I <di vis ion-operator> .
<times-operator>
. . - "*"
<division-operator> :: - "I"
<factor> ::- <primary>
[ <exponentiating-operator> <primary> 1 •
<exponentiating-operator> ::- "**"
<primarv> ::,,(1. <expression> ")"
<formal-parameter-name>
<unsigned-integer-number-literal>
<unsigned-real-number-literal>
<predefined-function-name> "(" <expression> ")" .

I

I

<predefined-function-name> ::<EXP-keyword>
<LOG-keyword>
<SQRT-keyword>
<SIN - keyword>
<COS-keyword>
<TAN-keyword>

I

<EXP-keyword>
<LOG-keyword>
<SQRT-keyword>
<SIN-keyword>
<COS-keyword>
<TAN-keyword>

· .· .· .·· ..· .-

"EXP"
"LOG"
"SQRT"
"SIN"
"COS"
"TAN"

Examples of numeric expressions :
primary :

factor
term

( amplitude + 10
frequency
123456
3 . 14159
sine 2 * 3.14 * freq )
10 ** 2
cos (omega)

**

2

(10 + amplitude) * 5
omega I {2 * 3.1416)

simple-expression :

-11

+3.14
-(amplitude + 10) - 100
a + b

4.78


5. SCOPE AND VISIBILITY

Scope and visibility are terms used in connection with certain rules that control the use of identifiers. Identifiers are used for naming objects; the naming is effectuated in a declaration.

5.1 Definitions

The scope of a declaration, sometimes also referred to as the scope of a declared identifier or object, is defined as the region of text over which the declaration has an effect. The scope starts at the point the identifier is introduced which is called the defining point of the declared object. When not defined otherwise, the scope ends at the end of the text.

This definition allows for the same identifier to be introduced by different declarations and may even be associated with different objects. The scope of several declarations with the same identifier may thus overlap.

The declaration of an object with a certain identifier is said to visible at (or from) a given point in the text when an occurrence of the identifier at this point can refer to the object, i.e. in the region in which an object is visible the occurrence of the associated identifier indeed unambiguously refers to that object.

Clearly, visibility of an object at a given point requires that this point lies within the scope of the declaration of that object. However this condition need not be a sufficient one because of overlap of scope of other declarations. Overlap of scope can make a declaration invisible or hide it.

The use of an identifier to denote an object other than at the point of introduction is called an applied occurrence of the identifier. For an applied occurrence to refer to the correct object it is mandatory that the applied occurrence is within the visibility region of the object’s declaration. Furthermore the point where an applied occurrence (mostly within a name) may occur depends on the syntax.

An identifier or object is said to be visible whenever the declaration of this object with this identifier is visible.

5.2 Scope of Declarations

The declarations of the objects involved in circuit description in NDML have the following scope:
For system templates the scope starts at the point of introduction of the identifier for the template and extends to the end of text.

The scope of declarations of objects that belong to a subclass of the class of template objects is always enclosed within the region of text that forms the template definition. In other words, the scope of terminal ports, formal parameters, subsystem instances and nets belonging to a certain template all start at their respective points of declaration and end at the end of the template definition, with the exception that the scope of formal parameters is extended to the end of the text. The latter exception is necessary because named parameter association is used, which allows formal parameters of a template to appear in the actual parameter part of instance declarations.

The scope of an implied net declaration coincides with the scope of the corresponding terminal port declaration.

It is an error, when within the scope of a declaration of a certain identifier with a certain class another declaration of the same identifier and for the same class of objects appears. This states that the language does not support the notion of nesting of scopes.

NOTE: a declaration of a terminal port implies a declaration of a net with the same identifier. This is allowed because terminals and nets are considered different objects. However, the above rule forbids that a net with the same identifier as a terminal port is explicitly declared in a net declaration. All this of course within the context of a template definition.

5.3 Visibility of Identifiers

Within the scope of declaration there are certain subregions where the declared object can be named simply by its identifier; the object, the declaration, and the associated identifier are then said to be directly visible from the subset. Where it is not directly visible, the region will be determined by a suitable context. The regions of visibility within the scope of declaration are determined by the language, so a user must obey some visibility rules in order to correctly use an identifier as an applied occurrence.

- An identifier for a given template is directly visible in instance declarations contained in template definitions following the definition of this template. A template definition can be viewed as a declaration; the template identifier scope starts as soon as the complete declaration is elaborated.
- Terminal identifiers are never visible; they merely serve as placeholders in the heading of a template definition. Do not confuse a terminal port with its associated implied net (see below). The association of actual nets with formal terminals in instance connection statements is done by position (place).

(Note however that a preliminary version of the language supports so-called terminal interconnect statements in which terminal port identifiers are directly visible.)

- An identifier for a net is directly visible in the actual net list of instance connection statements in the compound template definition where the nets are declared (either implicit or explicit).

- Formal parameter identifiers of a given template are directly visible within the expression on the right hand side of a parameter assignment in instance declaration appearing within the definition of this template. They are also visible in parameter assignments in instance declarations concerning this template contained in template definitions following the definition of this template but then on the left hand side.

Example:

The following rather absurd compound system definition shows the flexibility of the naming rules. Use of identifiers in this way is not recommended.

```
system name(name : signal);
   << name : default 1 >>;
instance
   name : other_name << name := 2 * name >>;
net
   other_name : signal;
begin
   name(other_name, name);
end; (* name*)
```

The identifiers "name" and "other_name" stand for (in order of appearance):

<table>
<thead>
<tr>
<th>1</th>
<th>name</th>
<th>introduction point of system template</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>name</td>
<td>formal terminal port</td>
</tr>
<tr>
<td>3</td>
<td>name</td>
<td>formal parameter</td>
</tr>
<tr>
<td>4</td>
<td>name</td>
<td>introduction point of instance</td>
</tr>
<tr>
<td>5</td>
<td>other_name</td>
<td>applied occurrence of system definition</td>
</tr>
<tr>
<td>6</td>
<td>name</td>
<td>applied occurrence of formal parameter of 1</td>
</tr>
<tr>
<td>7</td>
<td>name</td>
<td>applied occurrence of formal parameter of 5</td>
</tr>
<tr>
<td>8</td>
<td>other_name</td>
<td>introduction point of net</td>
</tr>
<tr>
<td>9</td>
<td>name</td>
<td>applied occurrence of instance</td>
</tr>
<tr>
<td>10</td>
<td>other_name</td>
<td>applied occurrence of net</td>
</tr>
<tr>
<td>11</td>
<td>name</td>
<td>applied occurrence of formal terminal</td>
</tr>
<tr>
<td>12</td>
<td>name</td>
<td>nothing to do with it, just comment</td>
</tr>
</tbody>
</table>
A system is the language term for a simulation unit. A simulation unit is a set of related data both constant and variable that constitute a certain behaviour as seen at its terminals within the simulator environment.

In fact, a system in NDML is the description of the constant data of a simulation unit. In conventional terms, a system consists of an interface specification together with a description of the inner behaviour or structure. The language does not allow mixed behaviour and structure description within one system definition.

<system-definition> ::=<SYSTEM-keyword> <system-interface-part>
<system-implementation-part> ";
<system-implementation-part> ::= <system-implementation> | <external-directive>
<system-implementation> ::= <leafcell-implementation> | <compound-implementation>
<external-directive> ::= <EXTERNAL-keyword>

Systems that describe structure are called compound systems, or compounds for short. Compounds offer a means to exploit hierarchical description. The structure of a compound consists of interconnected subsystems. A subsystem is an instance derived from either a leafcell (see below) or a compound.

Behavioural description is contained in systems definitions that are called leafcells. A leafcell directly reflects the type of underlying simulator program: in our case the body of a leafcell consists of piecewise-linear equations and ordinary first-order differential equations written in a canonical matrix form. Leafcell definitions are discussed in section 6 of this manual.

The subdivision of a system definition in interface and implementation part allows for a separate compilation scheme. Libraries of precompiled systems can be made; to access such a library system the user need only include the interface part of the system, thereby establishing appropriate defining points for the template. The absence of the implementation part is then indicated by the reserved word EXTERNAL. The semantics of separate compilation are not defined by NDML.

From a formal point of view, systems form a class of objects. Values of system objects are the so called instances. The class of system objects encapsulates all other classes of objects, i.e. all other classes of objects are contained
herein as a subclass. The subclasses in question are terminal ports, nets and parameters. Moreover we note that the class of system objects is recursive in nature: a (compound) system can be defined by reference to other system definitions.

6.1 System Interface Part

The system interface part of a system definition describes those aspects of the system that need to be known to the 'outside world'. The interface description is the same for both compound and leafcell systems. The interface part introduces a name for the system template and declares the formal terminal ports and formal parameters.

\[
\text{<system-interface-part> ::= <identifier> <formal-terminal-part> \\
\text{[ <formal-parameter-part> ]} .}
\]

\[
\text{<formal-terminal-part> ::= "(" <terminal-declaration> \\
\text{[ <terminal-declaration> ] \\n\text{")} .}
\]

\[
\text{<formal-parameter-part> ::= "" "(<parameter-declaration> \\
\text{[ <parameter-declaration> ] \\n\text{"") .}
\]

6.2 Compound System Definitions

A compound system defines a template for the structure of a (sub-)network. It can be understood as a sort of macro definition for a certain interconnection pattern between certain components. The compound template then can be used to create instances (macro calls) in order to build a more complex network. Like with macros of an assembler language, compounds (also leafcells) may be parameterised to achieve more flexibility in using them.

For the interface part of a compound system definition see the previous section. The compound implementation part consist of:

1. A subsystem declarations part,
2. an optional net declarations part, and
3. a body containing the interconnection statements.

\[
\text{<compound-implementation> ::= <instance-declaration-part> \\
\text{[ <net-declaration-part> ] \\
\text{<compound-body> .}
\]

4.83
6.2.1 The instance declaration part

consist of the instance declarations:

\[
<\text{instance-declaration-part}> ::= \\
\text{<INSTANCE-keyword>}
\text{<instance-declaration> ";"}
\{ \text{<instance-declaration> ";"} \}.
\]

6.2.2 The net declaration part

which is optional contains the local net declarations.

\[
<\text{net-declaration-part}> ::= \\
\text{<NET-keyword>}
\text{<net-declaration> ";"}
\{ \text{<net-declaration> ";"} \}.
\]

6.2.3 Interconnection

statements describe in what way the terminal ports of the declared instances and the formal terminal ports of the system being defined are interconnected. All connections are accomplished by nets. Here we have the following legal possibilities:

( in all examples the system "tl" interface part is:

```plaintext
system tl(pl, p2, p3 : signal);
```

- Connect a formal terminal of the system being defined to a terminal of an instance. The connection is done implicitly via the implied net.

Example:

```plaintext
system w(a, b, c : signal); 
instance
  i1 : tl;
begin
  i1 (c, b, a);
end;
```

a, b and c are the nets of w; a is connected to terminal p3 of the instance i1 of template tl, likewise b is connected to p2 and c to pl.

- Connect two or more terminals of possibly distinct instances.
the net nl connects terminal p1 of instance il of tl with terminal p3 of the same instance and p2 of instance i2. The net n2 connects terminals p2 of il and p1 of i2.

Leave a terminal of an instance not-connected, i.e. no net is associated with that terminal. This can be accomplished in two ways: either a declared net is associated with that and only that terminal or the special language symbol "-" is used. The latter approach is recommended.

the terminal p3 of instance il of tl is left open.

Connect two or more formal terminals. This possibility is obsolete and will be replaced by a different construct in the future.

the terminals p1 and p3 of the system z are internally short-circuited.

Summarising, we have the following syntax for the body of a compound system definition:
<compound-body> ::= <BEGIN-keyword> 
  { <terminal-interconnect-statement> } 
  { <system-instance-invocation> } ; 
  <END-keyword> .

<terminal-interconnect-statement> ::= <CONNECT-keyword> <formal-terminal-name-list> ;

<formal-terminal-name-list> ::= <formal-terminal-name> |
  <formal-terminal-name-list> , <formal-terminal-name> |
  "" <formal-terminal-name-list> .

<system-instance-invocation> ::= <system-instance-name> ( <actual-net-list> ) .

<actual-net-list> ::= <actual-net> |
  <actual-net-list> , <actual-net> |
  "" <actual-net-list> .

<actual-net> ::= <actual-net-name> |
  <not-used-denoter> .

<not-used-denoter> ::= "-" .

4.86
A leafcell system or leafcell for short models the behaviour of a certain hardware component by means of piecewise-linear equations augmented with first-order differential equations. The latter type of equations allow for dynamic, i.e. time-dependent behaviour description. Ordinary linear equations can be seen as a special case of piecewise-linear equations, however in NDML this distinction is made explicit, so in general a leafcell contains these three types of equations.

More conventional circuit simulators ([SPICE], [ASTAP]) typically start out from a set of linear, non-linear and differential equations, which they solve using well-known numerical methods. It is clear that compared with a piecewise-linear simulator the difference lies in the treatment of non-linearity.

As stated earlier leafcell systems cannot be distinguished from compound systems when viewed from the 'outside'. Both share the same interfacing constructs. The difference in syntax between both kind of systems becomes apparent when we look at the body of a leafcell definition. The syntax for the body of a leafcell directly reflects the mathematical notation commonly in use to describe piecewise-linear equations ([Bokhoven], [Eijndhoven]). All equations can conveniently be described in matrix-form. Hence, in principle a leafcell body is a matrix, be it that we have added some syntactic sugar for user convenience.

For completeness we repeat here the syntax of a system definition. The system interface part is described in section 6.1.

\[
\text{<system-definition>} ::= \\
\quad \text{<SYSTEM-keyword>} \text{ <system-interface-part>} \text{<system-implementation-part> ";" .}
\]

\[
\text{<system-implementation-part>} ::= \text{<system-implementation>} \\
\quad | \text{<external-directive> .}
\]

\[
\text{<system-implementation>} ::= \text{<leafcell-implementation>} \\
\quad | \text{<compound-implementation> .}
\]

\[
\text{<external-directive>} ::= \text{<EXTERNAL-keyword> .}
\]

With this in mind we now fill in the details concerning the leafcell implementation syntax:

\[
\text{<leafcell-implementation>} ::= \text{<matrix-definition>} \\
\quad | \text{<select-statement> .}
\]
7.1 The Voltage Reference

Two models for signals are currently defined by NDML. This is reflected by the two different types that are defined for nets and terminal ports, indicated with the reserved words "SIGNAL" and "ELECTR". In order to properly model electrical signals that are subject to the familiar Kirchoff's laws, the type "ELECTR" must be used. The type "SIGNAL" can be used to model e.g. logical signals.

Signals of type "ELECTR" can be viewed to consists of a voltage component and a current component. When they appear as variables in leafcell equations this distinction must be explicitly made, moreover we must also specify a common 'ground' terminal, with respect to which all voltages are expressed. For that purpose the reference definition is introduced:

```
<reference-definition> ::= <REFERENCE-keyword> "=" <formal-terminal-name> ";"
```

The formal-terminal-name must be of type "ELECTR", it is an error otherwise. The electrical terminal port declared to be the reference may not appear as variable in the variable-list.

7.2 The Matrix Definition

A fixed matrix definition describes the equations in matrix form whereby each element can be a numerical expression containing formal parameter names. The matrix rows are labelled by a keyword indicating the type of equation, the columns are labelled with their corresponding variable.

When many variables are involved, the equation matrix is likely to become rather large. To facilitate a nice lay-out when typing the matrix elements it is allowed to split the matrix in a number of parts each representing one or more complete columns, i.e. the splitting is only allowed in a vertical direction. Splitting up a large matrix in this way has no semantic effect.

```
<fixed-matrix-definition> ::= <variable-declaration> <row-list>
   ( <variable-declaration> <row-list> ) .
<variable-declaration> ::= <VAR-keyword> "(" <variable-list> ")" ";" .
```
The variable-list enumerates the variables that appear in the equations. Their position corresponds to the column of the matrix where their coefficients appear. The last entry in the variable-list must always be the numeric literal "1" (the digit 1) which is used to symbolically label the last matrix column which denotes a constant vector.

Two groups of variables are distinguished:

1. Variables denoting the signal value at one of the terminal ports of the leafcell system. This group is called the external variables. When this signal is of type "ELECTR" both its voltage component and its current component must appear explicitly as separate variables and therefore must appear with a proper extension to its name. No formal terminal port may be left unused, i.e. each must appear as a variable in the variable-list.

2. Variables introduced by the piecewise-linear modelling technique and variables used in the differential equations: the internal variables group. These are denoted by the reserved word "PL", c.q. "U", suffixed with a non-negative integer number, in order to distinguish each of them.

7.3 The Matrix Rows

Each row in the matrix definition is an equation of a certain kind expressing a relation between some of the variables appearing in the variable-list. The kind of equation is determined by the row-identification, which starts with a reserved word. Each row element can be an expression resulting in a coefficient to be multiplied with the variable corresponding to the column position of the element. Elements may be left empty in which case the coefficient is taken to be zero.
The three kinds of equations are:

1. Linear equations, indicated with the reserved word "ZERO" and followed by a suffix. In conventional mathematical terms such an equation reads:
   \[ 0 = a_{11} \cdot v_1 + a_{12} \cdot v_2 \ldots \]

2. Piecewise-linear equations, starting with the reserved word "PL" again followed by a suffix to properly distinguish between more of them. These equations relate the so-called complementary state-variables, often denoted with the letters "v" and "i" (see the literature about piecewise-linear modelling for more theoretical details). In PWL terms such an equation reads:
   \[ v_i = \ldots + d_{ij} \cdot i_j + d_{ij+1} \cdot i_{j+1} + \ldots \]

3. First-order ordinary differential equations, indicated with "DU" followed by a suffix. Such an equation expresses the derivative of an "U" variable with respect to time ( \( \frac{dU}{dt} \)) as a linear combination of some variables in the variable-list.

7.4 The Select Statement

In general, a select-statement will be used to bring related sets of equations in one and the same leafcell definition. Here 'related' means that the matrices describe basically the same kind of behavioural model but with different accuracy. As an example, consider a leafcell used to describe the exponent function \( e^X \). The precision with which the function value is calculated depends on the number of piecewise-linear segments used. This directly determines the size of the matrix. With a select-statement it is now possible to collect a number of matrices describing the function with varying precision into one leafcell. Upon instantiation a parameter can be set to choose the required precision.
A select-statement offers a number of at least 1 cases, each consisting of a matrix-definition preceded by a guard. The otherwise-clause is optional but when it appears it must be the last case. The guard is introduced by the reserved word "CASE" and states a condition by means of a boolean expression. The select-statement states that the applicable matrix-definition is the first one with a guard that evaluates to true. (Note the difference in semantics compared to the Pascal case statement.) When none of the guards is true and an otherwise clause is present then its matrix-definition applies. It is an error when none of the guards become true at evaluation time and there is no otherwise clause specified. Note that select-statements and also all statements described below are effectuated at evaluation time, i.e. just prior to the start of simulation.

7.5 Statements

Statements can follow the fixed-matrix-definition. Their purpose is to allow for certain relatively small changes to be made to the matrix probably in a conditional way depending on the values of certain parameters.

In many practical cases one need not use statements in order to properly model a certain behaviour. A situation where statements are very useful is when one wants to define a correct behaviour for certain boundary or degenerate cases. Take for instance a leafcell to model a capacitor having a parameter indicating its actual capacitance when instantiated. What if the required capacitance is zero? Another example are transistor models. It is common practice that a certain transistor is modelled in more than one way perhaps even with several different levels of accuracy within a model. This will in general result in matrices of different size.

The sole purpose of the empty-statement is to have more freedom in placing
semicolons. An empty-statement has void meaning.

7.5.1 Assignment-statement

An assignment statement assigns a value to a matrix element. Any previously assigned value is lost. The matrix-element-identification specifies the element to which a new value is assigned. The value must be of numeric type.

\[
<\text{assignment-statement}> ::= \; <\text{matrix-element-identification}> \; "\; := \; " \; <\text{expression}> .
\]

\[
<\text{matrix-element-identification}> ::= \; "[" \; <\text{row-identification}> \; "," \; <\text{variable-identification}> \; "]" .
\]

7.5.2 If-statement

If statements allow for conditional execution of sequences of other statements depending on the result of evaluating an expression.

\[
<\text{if-statement}> ::= \; <\text{IF-keyword}> \; <\text{expression}> \; <\text{THEN-keyword}> \; <\text{statement-list}> \; <\text{ELSE-keyword}> \; <\text{statement-list}> \; <\text{FI-keyword}> .
\]

The expression must evaluate to a boolean value. If this is the value true then control is passed to the statements following the "THEN", if the value of the expression is false then the statements in the "ELSE" clause if present are executed.

7.5.3 Remove-statement

The action of a remove statement is to eliminate a variable from the matrix-definition. Only internal variables may be removed, it is an error otherwise. The matrix-element-identification specifies the column corresponding to the variable to be removed. The row-identification can be chosen arbitrarily.

\[
<\text{remove-statement}> ::= \; <\text{REMOVE-keyword}> \; <\text{matrix-element-identification}> .
\]

7.5.4 Pivot-statement

The pivot statement is intimately bound to the algorithm used by the simulator. Its effect is to pivot a complementary PWL variable pair. This causes the simulator to start in a different linear segment. The matrix-element-identification must specify a "PL" type of row and "PL" type of variable (column).

\[
<\text{pivot-statement}> ::= \; <\text{PIVOT-keyword}> \; <\text{matrix-element-identification}> .
\]
7.6 Expressions in Leafcell Definitions

The expression constructs defined in section 4.2 must be extended to include logical expressions. Here we present the complete expression syntax:

\[
\text{<expression>} ::= \text{<simple-expression>} \ [ \text{<relational-part>} ] .
\]

\[
\text{<relational-part>} ::= \text{<relational-operator>} \text{<simple-expression}> .
\]

\[
\text{<relational-operator>} ::= \text{<less-than-operator>} | \text{<greater-than-operator>} | \text{<less-or-equal-operator>} | \text{<greater-or-equal-operator>} | \text{<equality-operator>} | \text{<not-equal-operator>} .
\]

\[
\text{<simple-expression>} ::= \text{<unary-operator>} \ [ \text{<multiplying-operator>} \text{<factor>} ] .
\]

\[
\text{<unary-operator>} ::= \text{<unary-minus-operator>} | \text{<unary-plus-operator>} .
\]

\[
\text{<unary-minus-operator>} ::= - ; \text{<unary-plus-operator>} ::= + .
\]

\[
\text{<multiplying-operator>} ::= \text{<times-operator>} | \text{<division-operator>} | \text{<AND-operator>} .
\]

\[
\text{<times-operator>} ::= \ast ; \text{<division-operator>} ::= \div ; \text{<AND-operator>} ::= \& .
\]

\[
\text{<factor>} ::= \text{<primary>} \ [ \text{<exponentiating-operator>} \text{<primary>} ] .
\]

\[
\text{<exponentiating-operator>} ::= \text{<expression>} "\ast" .
\]

\[
\text{<primary>} ::= \text{<formal-parameter-name>} | \text{<unsigned-integer-number-literal>} | \text{<unsigned-real-number-literal>} | \text{<predefined-function-name>} "(" \text{<expression>} ")" | \text{<NOT-operator>} \text{<primary>} .
\]
<formal-parameter-name> ::= <identifier>.

<predefined-function-name> ::=<br>  <EXP-keyword> | <LOG-keyword> | <SQRT-keyword> | <SIN-keyword> | <COS-keyword> | <TAN-keyword>.

<EXP-keyword> ::= "EXP".
<LOG-keyword> ::= "LOG".
<SQRT-keyword> ::= "SQRT".
<SIN-keyword> ::= "SIN".
<COS-keyword> ::= "COS".
<TAN-keyword> ::= "TAN".

<NOT-operator> ::= ".".
REFERENCES

[ASTAP]
Advanced Statistical Analysis Program (ASTAP),
Program Reference Manual, Pub. no. SH20-1118-0,
IBM Corp. Data Proc. Div., White Plains, NY 10604

[BDL]
Slutz, E., Okita, G., Wiseman, J.,
"Block Description Language (BDL): A Structural Description Language",
ACM IEEE 21-th Design Automation Conference Proceedings,
pp. 81-85, June 1984

[Bokhoven]
Bokhoven, W.M.G. van,
"Piecewise-Linear Modelling and Analysis",

[Eijndhoven]
Eijndhoven, J.T.J van,
"A Piecewise Linear Simulator for Large Scale Integrated Circuits",
Ph.D. dissertation, Eindhoven University of Technology, 1984

[Janssen]
Janssen, G.L.J.M.,
"PWL-Simulator User's Guide, First Edition" (in Dutch),
"Network Description & Modelling Language - User Manual",
Department of Electrical Engineering,
Eindhoven University of Technology, October 1984 (Not published)

[Ruehli]
Chapter 1 "Circuit Description" in
"Circuit Analysis, Simulation and Design - Part I",
A. E. Ruehli ed.
to be published by North-Holland

[SPICE]
Vladimirescu, A., Zhang, K., Newton, A.R., Pederson, D.O.,
Sangiovanni-Vincentelli, A.,
Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley, Augustus 1981

[Wirth]
Wirth, N.,
"What Can We Do about the Unnecessary Diversity of Notation
for Syntactic Definitions",
Comm. ACM, Vol. 20, Nr. 11, pp. 822-823, November 1977

4.95
Appendix I: SYNTAX DIAGRAMS

**system-definition**:  
```
SYSTEM → system-interface-part  
  system-implementation-part  
```

**system-interface-part**:  
```
identifier → formal-terminal-part  
  formal-parameter-part  
```

**system-implementation-part**:  
```
  system-implementation  
  EXTERNAL  
```

**formal-terminal-part**:  
```
{  
  terminal-declaration  
  }  
```

**terminal-declaration**:  
```
attribute-list  
  identifier-list → signal-type  
```

**signal-type**:  
```
ELECTR  
SIGNAL  
```

**attribute-list**:  
```
{  
  attribute  
  }  
```

**attribute**:  
```
INPUT  
OUTPUT  
INOUT  
UNUSED  
```

4.96
formal-parameter-part :
\[ << \text{parameter-declaration} \rightarrow \text{type-and-default-specification} \rightarrow >> \]

parameter-declaration :
\[ \text{identifier-list} \rightarrow \text{type-and-default-specification} \]

type-and-default-specification :
\[ : \rightarrow \text{value-range-specification} \rightarrow \text{default-value-specification} \rightarrow \text{default-value-specification} \]

value-range-specification :
\[ \rightarrow \text{lower-bound} \rightarrow \ldots \rightarrow \text{upper-bound} \]

lower-bound :
\[ \rightarrow \text{no-bound-denoter} \rightarrow \text{lower-bound-number} \]

no-bound-denoter :
\[ \rightarrow \star \]

upper-bound :
\[ \rightarrow \text{no-bound-denoter} \rightarrow \text{upper-bound-number} \]

default-value-specification :
\[ \rightarrow \text{DEFAULT} \rightarrow \text{number-literal} \]

compound-implementation :
\[ \rightarrow \text{instance-declaration-part} \rightarrow \text{net-declaration-part} \rightarrow \text{compound-body} \]

instance-declaration-part :
\[ \text{INSTANCE} \rightarrow \text{instance-declaration} \rightarrow ; \]
instance-declaration :

- identifier-list → template-name
- actual-parameter-part

actual-parameter-part :

- << parameter-association >>

parameter-association :

- parameter-assignment

parameter-assignment :

- formal-parameter-name := actual-parameter

actual-parameter :

- expression

net-declaration-part :

- NET net-declaration : identifier-list : signal-type

compound-body :

- BEGIN
- terminal-interconnect-statement
- system-instance-invocation :
- END

terminal-interconnect-statement :

- CONNECT formal-terminal-name-list :
formal-terminal-name-list :

system-instance-invocation :

actual-net-list :

actual-net :

leafcell-implementation :

matrix-definition :

REFERENCE = formal-terminal-name ;

fixed-matrix-definition :

variable-declaration :

variable-list :
matrix-element-identification:

if-statement:

remove-statement:

pivot-statement:

expression:

relational-operator:
simple-expression :

adding-operator :

term :

multiplying-operator :

factor :

primary :

predefined-function-name :
Appendix II: LEAFCELL EXAMPLES

**capacitor.m Page 1**

```literate
leafcell capacitor(ref_node, node : electr);
<< capacitance;
V_init : default 0;
>>;
(* electrical capacitor *)
($L$-)
begin
  reference = ref_node;
  var ( node.v, node.i, u.l, 1);
  du.l = (*1/c*), , ,
  if capacitance <> 0 then
    [du.l, node.i] : -1/capacitance
  else
    remove[zero.l, u.l]
  fi;
end; (*capacitor*)
($L+$)
```

**bool_vco.m Page 1**

```literate
leafcell bool_vco((output) out : signal;
  (input) freq : signal
) ;
<< max_frequency : 0..* (* required*) >>;
(* signal controlled boolean block generator;
  frequency of output signal equals value of input signal "freq"
  condition : 0 <= freq < max_frequency
  *)
($L$-)
(* PL-Model:
  build from bool block; additional switch (pl.3) allows frequency
  of output signal (pl.1) to be determined by a signal input.
  NOTE:
  pl.1 = 0 --> pl.3 = 0
  pl.1 = 1 --> pl.3 = freq ( < max)
  *)
begin
  var ( out, freq, u.l, 1); pl.1, pl.2, pl.3, 1);
  zero.l = -1, , , 1, , ;
  du.l = , 2, , , -4, ,
  pl.1 = , , -1, , 1, , 1;
  pl.1 = , , -1, , 1, , 1;
  pl.3 = , -1, , -max, , 1, max ;
end; (*vco*)
($L+$)
```

4.104
s_to_v.m Page 1

leafcell s_to_v(signal_in : signal;
    ref_node, out_node : electr
); (* signal to voltage source conversion *)

(*

\[
\begin{array}{c}
\text{signal_in} \\
\text{out_node.i} \\
\text{out_node.v} \\
\end{array}
\]

*)

begin
    reference = ref_node;
    var (signal_in, out_node.v, out_node.i, 1);
    zero.1 = -1, 1, 1, ;
end; (*s_to_v*)

($L+$)

v_to_s.m Page 1

leafcell v_to_s(ref_node, in_node : electr;
    signal_out : signal
); (* voltage to signal conversion *)

(*

\[
\begin{array}{c}
\text{in_node} \\
\text{signal_out} \\
\end{array}
\]

*)

begin
    reference = ref_node;
    var (in_node.v, in_node.i, signal_out, 1);
    zero.1 = -1, 1, 1, ;
    zero.2 = -1, 1, 1, ;
end; (*v_to_s*)

($L+$)
s_to_i.m Page 1

1 leafcell s_to_i (signal_in : signal;
  ref_node, out_node : electr
);

4 (* signal to current source conversion *)
5 (*

9 signal_in |____|<o + out_node
10 o----|____|--o -
12 *)
13 begin
14  reference = ref_node;
15 var (signal_in, out_node.i, out_node.v, 1);
16 zero.1 = -1, 1, 1;
17 end; (*s_to_i*)
18 ($L+$)

sine_gen.m Page 1

1 leafcell sine_gen(erator) (output) out : signal);
2  << freq : 0..* default 1;
3  ampl : default 1;
4  phase : 0..6.283184 default 0;
5  >>;
6 (* sine generator *)
7 ($L-)
8 begin
9  var (out, u.1, u.2, 1);
10  zero.1 = -1, 1, ampl * sin(phase);
11  du.1 = , 6.283184 * freq, 6.283184 * freq * ampl * cos(phase);
12  du.2 = -6.283184 * freq, , , ;
13 end; (*sine_gen*)
14 ($L+$)
leafcell nmost(B(*ulk*), D(*rain*), G(*ate*), S(*source*) : electr);
<< level: 0 .. 3 default 0 (* accuracy indication, integer [1] *);
Vfb -20 .. 20 default 1 (* Flatband Voltage [V] *);
W : 1 .. 1000 default 4 (* channel width, [µM] *);
L : 1 .. 1000 default 4 (* channel length, [µM] *);
theta: 0.01 .. 0.1 default 0.025 (* kT/q [V] *);
phif: 0.2 .. 1.0 default 0.58 (* Fermi level [V] *);
Cox : 5e-5 .. 1e-2 default 5e-4 (* oxide cap. per unit area [pC/VµM] *);
mobil: 10 .. 500 default 70 (* eff. electron mobility [µM/VµS] *);
msub: 0.1 .. 10 default 1 (* subthreshold current multipl.[1] *);
gamma: 0.01 .. 1.0 default 0.36 (* gate-bulk capacitance multipl.[V%] *);
lov1: 0 .. 100 default 50 (* gate source (drain) overlap [µM] *);
Rb : 0 .. 10 default 1 (* gate source (drain) overlap [µM] *);
Rg: 0 .. 10 default 0.5 (* gate series resist. [kΩ] *);
Rd : 0 .. 10 default 0.1 (* drain series resis. [kΩ] *);
Rs : 0 .. 10 default 0.1 (* source series resis. [kΩ] *);
Vrange : 1 .. 100 default 5 (* voltage range of nonlin.func. [V] *);
init_Vg: 0 .. 25 default 2.5 (* initial gate-bulk cap. volt. [V] *)
begin
($L-$)
reference - B(*ulk*)
var( D.v, G.v, S.v, D.i, G.i, S.i, u.l, pl.1, pl.2, l )
zero.1= 1, , -1, , (??*), , , (??*), , 1, , -1, , ;
zero.2= -1, , , , , , , , , , , , , , , , ;
du; = -1, , , , , , , , , , , , , , , , ;
pl.1 =(*??*), 1, , , , , , , , , , , , ;
pl.2 =(*??*), 1, , , , , , , , , , , , ;
[zero.1.D.i]=$\pm$ -Rd -L/(mobil*Cox*W*Vrange);
[zero.1.S.i]=$\pm$ Rs +L/(mobil*Cox*W*Vrange);
[du.1 .G.i]=- 1/(Cox*W*(L+2*Lov1));
[pl.1 .D.v]=$\pm$ -0.5*gamma/SQRT(phif + 0.5*Vrange);

(* uM :: micro-meter, kΩ :: kilo-ohm, nS :: nano-seconds, V :: volt, pC :: pico-coulomb, pF :: pico-farad, V% :: square-root( Volt )
22 *)

case level=0
(* The simplest piecewise linear MOS model, containing only two diodes. The following equations are implemented:
In*Ron - Vd - Vs + <Vg - Vt - Vs> - <Vg - Vt - Vd>
with: <x> = -x if x < 0; <x> = 0 if x > 0.
Vt = Vfb + e(Vs + Vd)
W = Vs = Vext - Is*Rs
D = Vs = Vdext - Id*Rd
Vg = Vg = Vgext - Ig*RG
Vg = u + (Is+Id*2*Vext)/L = 0
In = 0.5*(Id - Is)
Ron = 2L/(mobil*Cox*W*Vrange)

$e = 0.5*gamma / SQRT(phif + 0.5*Vrange)$

*)

begin
($L-$)
reference - B(*ulk*)
var( D.v, G.v, S.v, D.i, G.i, S.i, u.l, pl.1, pl.2, l )
zero.1= 1, , -1, , (??*), , , (??*), , 1, , -1, , ;
zero.2= -1, , , , , , , , , , , , , , , , ;
du; = -1, , , , , , , , , , , , , , , , ;
pl.1 =(*??*), 1, , , , , , , , , , , , ;
pl.2 =(*??*), 1, , , , , , , , , , , , ;
[zero.1.D.i]=$\pm$ -Rd -L/(mobil*Cox*W*Vrange);
[zero.1.S.i]=$\pm$ Rs +L/(mobil*Cox*W*Vrange);
[du.1 .G.i]=- 1/(Cox*W*(L+2*Lov1));
[pl.1 .D.v]=$\pm$ -0.5*gamma/SQRT(phif + 0.5*Vrange);
if \text{init}\_Vg < V_{fb} \text{ then pivot}[p1.1, p1.1]; pivot[p1.2, p1.2] fi;
end;

bool_latch.m Page 1

leafcell bool_latch((input) data, clock : signal;
       (input) preset, clear : signal;
       (output) q, q\_bar : signal);
   \langle q\_init : 0..1 default 0 >>;
(* level clocked D-type flipflop with independent preset and clear
   (all inputs active HIGH)
   *)
($L+$)
begin
var ( q, q\_bar, data, clock, preset, clear, pl.1, pl.2, pl.3, 1);
zero.1 = -1, , , , , 1, -1, ;
(* inverted output : *)
zero.2 = -1, -1, , , , , , , 1;
pl.1 = , , 1, -1, , , 1, , , ;
pl.2 = -1, , , -1, -2, 2, 2, 1, , ;
pl.3 = -1, , , -1, -2, 2, 2, , 1, 1;
if q\_init = 1 then
   pivot[pl.2, pl.2];
   pivot[pl.3, pl.3];
fi;
end; (*bool_latch*)
($L+$)
Chapter 5

The ASTRA System
THE ASTRA SYSTEM : SYMBOLIC DESIGN OF CUSTOM VLSI

M C Revett
British Telecom Research Laboratories

ABSTRACT

The ASTRA system is a comprehensive CAD system which has been developed for custom chip design. It supports a design method that uses two distinct forms of symbolic design data, hierarchical floor plans and cell layouts. From this design data, which is mainly entered and edited via interactive graphical editors, both the information required to check and simulate the design and the final geometrical mask data used to fabricate the chip is generated automatically. The system provides a complete chip design environment, and is currently in production use for the design of several custom CMOS chips.

1. Introduction

A comprehensive CAD system for custom chip design, known as ASTRA - A STRuctured Approach, has been developed at British Telecom Research Labs (BTRL). It is based on the use of a design method that utilises two distinct forms of symbolic design data, hierarchical floor plans and cell layouts, to describe and capture the design. From this design data both the information required to check and simulate the design and the geometrical mask data used to fabricate the chip is generated automatically. The ASTRA system provides a complete design environment, apart from the use of industry standard circuit and logic simulation programs, to which direct interfaces are provided, and the use of post-processing programs to produce mask data in specific formats required by silicon foundries.

The system has been developed through two main versions. The first experimental version [1] was written in Fortran 77 to run on VAX computers running the VMS operating system. Based on the comprehensive evaluation of this experimental system, including its use in the initial stages of the design of a 25k device CMOS chip [2], a new and enhanced version of the

This paper was produced as part of Task VI of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
system has been developed. This version was written in C and runs on a network of machines running versions of the UC Berkeley UNIX operating system. The network includes a number of high performance graphics workstations, each with 1024x768x16 colour graphics displays, keyboard and mouse for user input, 2Mb of main memory and at least 100Mb of local disc storage. Also on the network is a VAX computer running ULTRIX, on which most of the ASTRA software can be run. This is used to off-load heavy computational tasks, to provide central storage of design data, and to provide access to a range of plotting and printing peripherals. In addition, other VAX machines running the VMS operating system are accessible via the network for use in running simulation and post-processing programs.

2. System Overview

The basic structure of the ASTRA system is shown in figure 1. In the figure circles represent programs that are run by users to perform the various design operations, and the boxes represent (sets of) files of design data. The ASTRA system is accessed by the user predominantly via the graphics workstations, and there is a 'graphical shell' program [3] through which the user controls the execution of the various programs, using multiple 'windows' on the graphics screen.

The programs in the ASTRA system were developed to be as far as possible process independent. The information that characterises specific processes is stored in text files, which are accessed for process details as required by individual programs. The specific process to be used during a design session is set by the user. The current set of processes supported by the system are all single level metal CMOS, representing a number of different silicon foundries. Other types of processes, eg NMOS or two level metal CMOS, could be supported by creating suitable process information files.

The way in which a chip designer makes use of the programs shown in figure 1, and the forms of data that are entered and generated, will be outlined in the rest of this document, with references being given to the other documents in this volume that give detailed descriptions of specific programs and their associated data files.

3. Design Approach

The conventional approach to chip design, as currently used by many design houses, splits the design process into two distinct phases, functional (logic) and physical (layout),
The ASTRA System

figure 1
with two, often rather separate, sets of design tools to support the two phases. The functional design phase uses some form of schematics capture, from which net-lists are generated, and various forms of simulation are performed. Following completion of the functional design, the layout phase is undertaken, typically using an interactive graphics system to create geometrical mask drawings, followed by various checking processes, including design rule and electrical rule checks. In addition, some further simulation is usually necessary to check that timing approximations made during the original simulation are valid for the actual circuit layout. The final step typically involves a complete extraction of the circuit from the full mask set, followed by a network comparison with the original net-list generated from the schematic, to check that the layout implements the logic designed during the functional design phase.

The approach supported by the ASTRA system, based on a combination of two ideas introduced in recent years, floor planning and symbolic layout, provides a number of advantages over the conventional approach. The first main advantage is that the split between functional and physical design is removed, and the two phases of design are encouraged to proceed in parallel. The splitting of the design process into separate stages may have been appropriate for older technologies, eg printed circuit boards, but is certainly at odds with more recent ideas on structured chip design [4], in which algorithms and chip architectures are chosen which map effectively into silicon.

A second main advantage, which follows from the first, is that the design data is captured only once, and from it is generated automatically both the net-list data for simulation and the final mask layout data. This means that design data can be checked and validated at an early stage, leading to earlier and easier correction of errors, and ensures that simulation is performed on realistic data, including layout related effects.

A third main advantage is that the complete set of geometric mask data is generated automatically from more abstract, symbolic design representations, which means firstly that the layout is design rule correct by construction, and secondly that a chip design can be to some extent process 'portable', enabling different versions to be generated for different foundries, or enabling a design to track process improvements. This capability was utilised in the design of the first ASTRA designed chip, where a significant change in the design rules was introduced by the foundry at a late stage, and the change was accommodated with a small amount of effort by the designer.
Finally, the design approach provides a more abstract, less cluttered view of chip design, not least because the designer is freed from the need to have detailed knowledge of a full set of process rules. This enables the human to concentrate on the important, global design issues, leaving the computer to deal with many of the details.

4. Floor Planning

The term floor planning has sometimes been used to describe a variety of loosely defined aspects of chip design, but in the ASTRA system floor plans have a precise format and function in the design process. The overall floor plan of a chip consists of a hierarchically defined set of floor plans. At the top level in the hierarchy the floor plan of a complete chip consists of a partitioning of the total chip area into a set of named abutting rectangular blocks, with defined sets of connections between them. The constituent blocks themselves have floor plans of an equivalent form, and so on down through the hierarchy.

The set of hierarchically defined floor plans are entered and edited by means of a comprehensive graphical floor plan editor [5]. The basic editing operation consists of subdividing rectangular areas of silicon into smaller rectangular areas by means of horizontal or vertical cut lines. Once a rectangular area has been given a name it becomes indivisible at that level, and must be "descended into" to be further sub-divided. In addition, all connections between blocks are entered as connectors along common edges of adjacent blocks. All connectors have default or specified names, and optionally defined attributes such as material (eg polysilicon, p-diffusion etc) and direction (in, out, inout). As connectors are added they are propagated up or down the hierarchy, appearing where appropriate as external connectors on higher or lower level blocks.

The floor plans capture two types of information. Firstly, the structural interconnection information, conventionally represented in a block diagram, which is used for generating net-list data for simulation purposes, and, secondly, the estimated sizes and topological relationships of the blocks, the latter of which is used in generating the final layout during the automatic chip assembly stage. The floor plan editor also enables information to be entered which will control the chip assembly process, including the definition of a block as 'rigid', and the alignment of connectors or edges on different blocks.

At the lowest level in the floor plan hierarchy there can be placements of three types of objects. Symbolic cells are the
usual form in which the detailed circuitry is defined, as will be described in the next section. A lowest level block can contain a placement, or an array of placements, of a symbolic cell with one of the eight possible orientations, and, optionally, with some of the pins on the symbolic cell left unconnected. Layout cells are pieces of geometrical mask layout, which are mainly used for such features as pads, registration marks and logos, but can be used to incorporate special purpose pieces of circuitry defined directly at the geometrical level. They have a fixed size, and have to be handled as 'rigid' blocks during chip assembly. Floor plan modules are effectively further pieces of (hierarchical) floor plan, which enable multiple instances of larger pieces of circuitry to be placed, with defined orientation and disconnected pins if required.

5. Cell Design

Symbolic cells represent rectangular areas of circuitry, with defined connection points (pins) around their edges. The circuitry consists of tracks on the various process layers, which are connected through contacts, and create active devices where polysilicon crosses diffusion. The tracks have defined width, and the active devices have defined width and length, in each case specified as a multiplier of a default value.

The design of the symbolic cells is normally performed using a graphical editor [6], which has a comprehensive range of editing functions to facilitate experimentation and iterative refinement of cell designs. These include multiple screen windows, which enables concurrent editing of different cells or different active views of the same cell, various selective move and delete options, and the ability to edit in a conventional 'sticks' form or in a sizes.

In addition to the graphical image of the cell, the editor continuously maintains an electrical representation. This enables various forms of electrical rule checking to be performed in the background during editing [7], providing the designer on request with a full set of circuit errors and warnings, which if corrected before completion of the editing session will guarantee a valid simulation net-list of the cell is immediately available. The editor can also highlight electrically connected nodes in a cell on request.

Simulation data is generated automatically on request, for user specified areas of the design, and in the required form and format for the relevant simulation program [8]. The simulation net-list of a symbolic cell initially consists of a switch-level representation, in the relevant format for specific switch-level simulators. By combining structural
information from the floor plans and individual cell net-lists simulation can be performed on larger areas of circuitry. To perform more detailed circuit-level simulation, with realistic capacitive and resistive parasitics, the actual sizes of all elements in the individual cells must be generated using the spacing program [9], and using this information the full set of parasitic data can be generated.

The interface between floor planning and cell design is provided by cell 'outlines'. If used in a pure top-down fashion, floor planning will produce an outline as the physical specification for a cell. This outline defines the relative positions and layer for the pins around the edges, together with an estimate of required size and aspect ratio. This outline can be imported from the floor plan description into the cell editor as a starting point for detailed design. In practice, previously designed cells will also need to be incorporated into floor plans (bottom-up), and in this case the outline of a cell can be created and exported into the floor plan editor.

Various forms of automatic block generation are also available in the ASTRA system, including PLA, ROM modules and switch-box routing cells. These are generated in symbolic form, and are processed in the same way as manually designed blocks.

6. Geometrical Mask Data Generation

Generation of geometrical mask data proceeds in two steps. Firstly, maximally compacted versions of each different instance of each cell are produced by the spacing program [9], using the required process design rules as defined in a relevant technology file. The spacing program uses a combination of virtual grid and graph based methods, and provides a range of features, including spacing about a specified interior or boundary point, and the selective pruning of redundant elements from a cell with pins left unconnected. This latter feature has been found to be useful in producing different functional instances of a single symbolic cell. In addition, information on which design rule has produced a specific constraint is stored, and can be accessed from within the symbolic cell editor to guide the designer in making suitable symbolic layout changes to reduce the final area of the cell.

The second step in the mask layout generation process is called chip assembly [10]. This involves assembling all the constituent cells into the configuration defined in the floor plans, with connection between adjacent blocks being achieved either by stretching cells where necessary to align
connection points and edges, or, optionally, by river-routing between cell edges. This chip assembly process includes comprehensive checking, based on comparing the number, relative position, layer, width and type of connecting pins on abutting cells. In addition, chip assembly also ensures no design rules violations occur at the abutting edges of cells, based on 'significant edge' information passed to it by the cell spacing program.

The input data for the chip assembly process is the relative positioning and connection information from the floor plans, and the size, pin positions and 'significant edge' information for the maximally compacted cells produced by the spacing program. The chip assembler sets up and solves a directed graph of constraints between all pins and edges in the horizontal and vertical directions in sequence. The result of the chip assembly process is the 'stretched' floor plan, which contains the complete set of final sizes and pin positions for all the cells. A graphical plot of the stretched floor plan can be produced which shows the places where stretching has occurred, to enable the designer to identify fruitful areas for further design optimisation. In practice, chip assembly normally forms part of an iterative design process, with critical blocks being designed and assembled first, to enable the floor plan to be refined before further blocks are designed.

When a satisfactory assembly of the complete chip has been achieved, the mask generation program [11] is run, which produces for cells which have been stretched the geometrical mask data for the new size and pin positions. The complete set of mask data for all cells and the positions of their placements within the floor plan are used to produce the data for mask making, bloated or shrunk if required.

Experience with chip assembly has led to considerable refinement of the process to enable good quality final layouts to be produced. These refinements include the ability to define rigid boundaries within the floor plan hierarchy, which enables lower level blocks to be kept at their assembled size as higher level assembly takes place, with connections being made to their neighbouring blocks by automatic river routing if necessary. This same process is also used to incorporate fixed size layout cells. The chip assembly process can also be controlled by using defined alignments between specified edges or connectors.

7. Conclusions

The first version of ASTRA running under UNIX, which consisted of a subset of the total system, was released for production use in late 1984, and a number of subsequent
releases brought it to a full system by late 1985. Two chips designed on the system, both of around 25k devices in 2.5 micron CMOS, have so far been taken through to fabrication, and several further designs are now at various stages of development. All design work is currently for single metal CMOS, for which at present several different 2-3 micron processes are supported by the system.

Acknowledgements

I would like to thank my colleague Dr. Peter Ivey for all his efforts in the origination and co-management of the ASTRA work, and the many individuals whose efforts and enthusiasm have contributed to its success, including Paul Benyon, John Cheshire, David Myers, John Oldfield, Dave Orrey, Brian Roe, Richard Scarfe, Richard Weeks and Nick Weiner.

The work described in this document comprised Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme.

References


THE ASTRA SHELL CONTROL ENVIRONMENT

B J Roe

British Telecom Research Laboratories

ABSTRACT

This paper gives some insight into the requirements and implementation of the ASTRA shell, which is the program that controls the user interface for the ASTRA VLSI design system. Its limitations are discussed and further work suggested.

1. Introduction

The ASTRA suite of software [1] has been designed as a unified system for the design of custom VLSI. As an aid to this unification early discussions between the software designers and the potential users of the system produced a list of requirements for the system. Some of the main requirements were:

- Users wanted the ability to run several processes at the same time.

- Users wanted "Short Commands", that is single keystroke commands, for the editors to be controlled outside the editors themselves, so that all instances of a particular editor in a session would have the same short commands.

- The designers of the symbolic cell editor wanted to provide a system where coordinate information could be obtained either from the mouse, or typed in by the user at the keyboard.

As the workstation used did not provide the necessary environment to allow such facilities to be provided from within the existing UNIX+ "sh" and "csh" utilities it was decided to write a user interface program to handle some of these requirements. The following sections describe the major features of the user interface and its interaction with the other programs of the ASTRA suite [2,3,4,5,6].

† UNIX is a trademark of Bell Laboratories.

This paper was produced as part of Task VI of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
2. Display Area and Zones

When ASTRA is started the screen is split into 2 areas, the system zone and the user zone. Textual output from the shell and all its sub-processes appears in the system zone. When the user creates a sub-process, an area of the user zone, known as a process zone, is defined for graphical output. A separate area of the screen, the message zone, is reserved for static system messages. This area is used for displaying the function of the mouse buttons. In ASTRA, process zones are allowed to overlap. Graphical output to overlapping process zones will be mixed owing to limitations in the hardware and operating system.

3. Event Queue

The shell handles all input from both the keyboard and the mouse. The editors require coordinate input from either source, hence all input is formed into a single homogeneous chronologically ordered event queue. Input from both sources is handled in an asynchronous manner, using the interrupt facilities of UNIX. The event queue is a single linked list of data items. Each data item holds the following:

- The position of the mouse at the time of the event.
- An indicator of the event type.
- A pointer to the next event.
- The data from a keyboard event.

When events are processed by the shell the data space is saved on an internal free list. When events are added to the event queue, space is used from the free list in preference to allocating new space from the system. This approach minimises the calls to the system memory allocator.

Events fall into 5 classes, 1 class for mouse events and 4 classes for keyboard events. These classes are:

- Mouse events.
- User zone short command events.
- Shell (system zone) short command events.
- User long command events.
- Shell long command events.

The mouse is set to signal when movement is detected or when a mouse button changes state. Long button depressions are used to display pop up menus. When a button is depressed a timer is fired which is cancelled when the button is released. If the timer expires and signals the shell process a pop-up menu is generated. Events are queued for mouse movements and button transitions.

The keyboard is set in CBREAK mode and is set to signal on
each character. Certain character codes are reserved to indicate the start of a "long command", the character "!" is used for shell long commands and the characters ";" or ":" are used for user long commands. When one of the starting codes is received from the keyboard a flag is set and all keys up to the next new line are saved in a character array. The event is queued, and the mouse position noted, when the new line is entered. Command line editing, similar to that provided by the tty driver, is available when a long command is being entered.

If the user is not entering a long command and the mouse is pointing to the system zone a shell short command event is queued. If the mouse is not pointing to the system zone a user area short command event is queued.

4. Processing events

The shell waits for input events to process and keeps track of the sub-processes, removing dead process entries from its internal process tables. When input is available, the shell gets each event from the event queue and acts on it according to the event type. All user long commands and user short commands are sent to the current process. All shell long commands and shell short commands are processed by the shell itself.

Mouse movement events are ignored when the mouse is over the system zone, or outside the zone of the current process. Mouse events over the current process zone are compared with the types of events required by the current process. Mouse movement events are used only by the Floor Plan Editor [2]. Mouse button release events are used both by the Floor Plan Editor and the Symbolic Leaf Cell Editor [3].

A mouse button release event over a process zone, other than the current process zone, is treated as a signal to switch the current process to the new process zone.

5. Process Creation and Control

The shell handles process creation and all input to the system. It maintains information about all sub-processes created and keeps a pointer to the "current process". When the shell receives a command, a check is made to see if it is one of the small number of built-in commands which are executed directly. If this is not the case check is made to ensure that the command exists in the directory associated with the ASTRA commands and that the file is executable and not a directory.

Information regarding the process requirements is obtained
from internal tables. This specifies the following attributes of the command:–

produces graphical or textual output.
requires coordinates with each message text.
requires mouse movement messages.
requires button depression messages.
requires button release messages.
the default zone coordinates.

A set of command arguments are created from the command line entered. If the command uses graphical output the coordinates of the zone to be used are appended to the argument list.

The shell creates a pipe for process communication and forks a copy of itself. The output pipe in the child process is attached to the standard input stream of the process and the signal handlers used by the shell to handle the mouse, keyboard and timer interrupts are disabled. The new command is executed from the child process using the "exec" system call.

6. Communication with Sub-processes

The input end of the pipe created for a sub-process is noted in the sub-process data structure. The general format of a message to a sub-process is:

\[
\text{XCOORD YCOORD TEXT_OF_MESSAGE}
\]

If the sub-process does not require the coordinate information, that part of the message is not sent. All coordinates are sent as ASCII strings. This method simplifies interprocess communication and allows simple testing of the individual programs.

7. Short Commands

During the initialisation of the shell, a path, defined in an environment variable, is searched for files containing definitions of short command substitutions. These definitions are read in for various processes defined in the shell, notably the two graphics editors and the shell itself.

When a short command event is received a check is made that short commands have been defined for the current process type or the shell. If short command definitions exist, the data character from the short command event is used as an index to the substitution string. The resultant string, if defined, is sent to either the current process or the shell.
command parser as appropriate.

8. Menus

Pop-up menus are displayed when a mouse button has been held down for a pre-determined, user adjustable, time. Pop-up menu data is read in during the shell initialisation and are stored in a logical name data structure [7]. For each process, the shell maintains data containing the last displayed menu.

Each of the 3 buttons of the mouse can be used to display a menu. The centre button displays the shell menu, the right hand button displays the last selected menu for the current process and The left hand button displays the "quick" menu for the current process. Thus it is possible to have more than one instance of a program running yet the shell knows which menu to display for each process. This approach contrasts with the method for expanding short commands.

When a menu is displayed, the mouse is then controlled by the menu software and the button definitions are displayed in the message zone. Pop-up menus use facilities in the workstation to generate a pop-up text zone. The size of the pop-up zone is defined by the number of menu entries. The displayed zone is centred about the current cursor position, modified to restrict the displayed zone to be within the user zone and aligned to a 16 pixel boundary for speed of display. The display text for the menu is written at known points in the zone. The position of the mouse is compared with the known position of the displayed text and the appropriate menu item selected.

A graphics zone is opened over the top of the pop-up zone and this is used to highlight the currently selected menu item. As menu items are selected, a command string is built up for processing by the shell or the current process. Some menu entries can be used to change to another menu display. A modal command string can also be added to the generated command string, when changing from one menu display to another.

Some of the menu entries have an on-line help frame associated with the menu entry. When help frames are displayed the aspect ratio of the menu is changed to enhance the format of the text. When the exit button is released, a check is made on the position of the mouse. If the mouse is over the displayed menu, the command string is sent to be processed, otherwise the command string is discarded. After the menu has been removed the shell signals the current process to update the message region of the main display.
9. Conclusions

The shell provides a common interface for all the other programs in the ASTRA suite, overcoming the limitations of the workstation operating system and hardware. There are, however, difficulties for the writers of the other programs. All programs requiring graphics input and output must respond to the standard messages generated by the shell. Programs can only get input through the shell and this could lead to slow response times in some applications. All textual output from a program is displayed in the system zone, leading to intermixed text from many processes which could cause confusion.

Further work is suggested in the following areas:-

A single common graphics manager, run as a daemon process to coordinate the graphics from multiple processes.

Generation of text zones for the output from user processes.

Modifications to the menu handler to allow text input from the keyboard to be appended to the command string before execution by the shell or sub-process.

Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme.

References


The floor plan editor is an interactive graphical editor for the ASTRA VLSI design system. The screen of the terminal acts as a window into the file currently being edited. Changes made to the file are reflected in what is seen. The editor can be used to create and modify hierarchical floor plans consisting of abutting rectangular blocks with defined connectors along block boundaries.

1. Introduction

The design of a chip starts with some specification of the required behaviour or function of the chip. Using the design method supported by ASTRA [1] the first stage in the design process is to define the basic architecture of the chip and its associated floor plan. This involves partitioning the chip into an interconnected set of blocks, each with a defined function and an estimated rectangular size. This partitioning is done by the designer based on his expertise and experience, and is entered into the design system using the graphical floor plan editor.

Once a floor plan has been entered, it can be used to automatically generate input data for simulation programs. The structural information it contains is combined with suitable functional specifications of the blocks and relevant waveform data [2].

Partitioning can be applied in a similar way to the main functional blocks of the chip, again using the graphical editor. This partitioning can proceed down the hierarchy until the resulting functional blocks are sufficiently simple to be implemented directly, or until a block is specified for which a design already exists. At each stage of the partitioning process, simulation and testability analysis can be applied to verify the quality of the design, and the resulting input and output data can be accumulated.

This paper was produced as part of Task VI of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
for later use in testing.

The lowest level blocks or cells are defined symbolically using the leaf cell editor [3].

When partitioning is complete and all cells have been designed the complete chip is assembled using the chip assembler program [4]. For each of the constituent cells the spacing program [5] will have produced a file defining the minimum size and associated pin positions of each cell. The assembler works through the hierarchy defined by the floor plan, stretching cells and blocks as it goes to provide pitch matching between pins. No provision is made for "spaghetti" interconnections as all connections between blocks are made by abutment. However blocks containing only routing are supported and can be created automatically using a switch box router.

At the end of the chip assembly process the exact size and pin positions of all constituent cells are known. The spacer has produced mask geometry data which, when combined with the hierarchical information extracted from floor plans, forms the mask geometry data files used for post processing. These are processed in the conventional way to produce the masks.

This document describes the graphical floor plan editor, which forms part of the ASTRA VLSI system. Details are given of the basic editing functions, the commands used to perform these functions and the data structures used to store and manipulate floor plan data.
2. The Primary Editing Functions

An example floor plan of abutting rectangular blocks is shown in figure 1.

Figure 1 An example floor plan
+'s represent connections between blocks

Four primary editing functions are described. They represent the minimum operations required to produce a useful hierarchy of abutting rectangular blocks.

Bisection of a rectangle
Declaration of a sub-block
Connection between abutting blocks
Placement of a lower level module

5.20
2.1 Bisection of a Rectangle

Figure 2a shows a new block that is to be floor planned. The silicon area represented is partitioned into smaller rectangular areas by a process of "cheese slicing", as shown in figures 2b to 2d:

![Figure 2a](image)

**Figure 2a** New block - to be floor planned

![Figure 2b](image)

**Figure 2b** First bisection made
2.2 Declaration of a Sub-block

Rectangular partitions of a block thus created may be assigned to a sub-function by "naming" the area - ie sub-blocks may be declared.
The regions resulting from the cheese slicing (figure 2d) are assigned to sub-blocks as shown in figure 3, below.

![Diagram of sub-blocks](image)

Figure 3 Sub-block declared within each rectangular area

Having declared a sub-block it is possible to move down into it and then resume floor planning at the lower level.
2.3 Connections Between Abutting Blocks

In order to define communications between abutting blocks, connections may be placed onto cut lines. The floor plan is completed by placing connectors onto the cut lines of figure 3. Figure 4 below gives the completed floor plan.

```
Figure 4 Connectors added to complete the floor plan; +'s are connectors
```

2.4 Placement of a Module

"Modules" can be other floor plans, or cells that are defined by the leaf cell editor. They allow general functions of a design to be used in more than one place with slightly different characteristics (orientation, disconnected pins etc).
In order to place another module it is necessary to first move into the 'module block' (the block into which the module is to be placed). For example, to place moduleP into blockC, first of all move into blockC:

![Diagram of blockC]

Figure 5 New (empty) block

Then place moduleP. If moduleP DOES already exist then it may be placed by specifying which of its pins are to be connected and which are not (figure 6 shows all of the pins in use). If the module does NOT already exist then this operation defines the interface.

![Diagram of moduleP in blockC]

Figure 6 A module placement

3. The Visible World

The view of the editor to the user is through an ASTRA "zone", the size and position of which is defined when the process is started. A description of the ASTRA shell control environment can be found in reference [6].

3.1 Zones, Areas and Windows

Each zone is a "window" to a floor plan. The window can be changed to view different parts of that floor plan at different magnifications.
3.2 The Floor Plan

The floor plan is displayed as a partitioned rectangular block representing the plan at a particular level in the hierarchy. Communications between blocks are displayed as bars, the colour represents the material of the connection; white if undefined. An arrow or arrows represent the signal direction; no arrows if the direction is undefined. The name of the current block is displayed above the floor plan and the names of sub-blocks are displayed within each sub-block.

Labels assigned to connectors can be displayed or turned off, as can the connectors themselves, depending on the amount of information required.

3.3 Marked Items

Items are selected as "marked" for editing as the cursor moves over a cut line or connector. A marked cut line appears red and a marked connector has a red box surrounding it.

3.4 The Cursor

The cursor is a highlighted red cross. It can be moved around the screen by moving the mouse on the pad. As it is moved around the floor plan, items become marked only whilst the cursor is in close proximity to the item. Whilst within the editor zone, pressing the left or right button has the following effect: if the cursor is within a block, that block will be bisected by either a horizontal or a vertical cut line depending on whether the left or right button is depressed; if the cursor is already on a cut line a connector will be added.

The effect of depressing the centre button is dependent on the current modal or immediate command being executed.

3.5 Window Commands

A zone displays a "window" to a floor plan. The parameters of this window (magnification, position, etc) can be changed using a set of simple keywords.

The keywords used with the WINDOW command are:

FIT sets the parameters to display the entire floor plan and contents plus a small margin.

REPLOT replots the window.
UP, DOWN, LEFT, and RIGHT move the centre of the window to
give the desired effect. CENTRE moves the centre to a
specific point, allowing the user to centralise around an
area of interest.

IN and OUT increase and decrease the magnification of the
displayed window. ZOOM allows the user to specify an area of
interest which can be magnified to fill the zone.

LAST plots the last window displayed even if the previous
window was at a different level in the hierarchy, while
STORE and RESTORE allow particular views to be stored and
re-displayed later.

Plotting can be aborted at any time using a control C.

4. Inserting Data

4.1 Bisecting Rectangles

Floor plans are created by producing a hierarchy of abutting
rectangular blocks. The silicon area represented by an empty
block is partitioned into smaller rectangular areas by a
process of "cheese slicing" using the left and right
buttons.

4.2 Connections between Abutting Blocks

In order to define communications between abutting blocks,
connections may be placed onto cut lines. Connections
between blocks are inserted using the left and right
buttons, but only if the cursor is currently highlighting a
line. The connector is added with a negative signal
direction (left or down) if the left button is used and
positive (right or up) if the right button is used. The
modal commands BI or UN allows the centre button to be used
to add bidirected or undirected connectors.

The MATERIAL command defines the material of connectors
subsequently added using the buttons.

Connectors can be named using NAME and QUICK_NAME.

4.3 Declaration of a Sub-block

Rectangular partitions of a block may be assigned to a sub-
function by naming the area - ie sub-blocks may be declared.
This can be achieved by using the immediate command
QUICK_NAME to name specific blocks. Having declared a sub-
block it is possible to move down into it and then resume
floor planning at the lower level.

5. Moving Around the Hierarchy

The command JUMP is used to move down the hierarchy. This modal command allows the centre button to identify the block to move down into. Alternatively the immediate command QUICK JUMP moves down into the block identified by the cursor position without the need for a button depression.

The immediate command ASCEND moves up one hierarchical level and GOTO moves to a specified block.

6. Placement of a Module

In order to place another module it is necessary to first move into the "module block" (the block into which the module is to be placed) and PLACE the new module, specifying orientation, type and array size if relevant. If the new module already exists then it may be placed by specifying which of its pins are to be connected and which are not. If the module does not exist then its interface will be defined by the placement.

Module placements can be removed using UNPLACE.

A module placement can be modified using PLMOD and specifying the parameters to be modified. Disconnected pins can be reconnected using RECONNECT.

7. Selecting and Deleting Data

7.1 Selecting

Editing functions always operate on marked data. Only single items (cut lines or connectors) can be marked, achieved simply by moving the cursor near enough to the item to become highlighted.

7.2 Deleting

Single items can be deleted very easily by using the modal command DELETE to change the state of the centre button so that each probed item is deleted. Alternatively, QUICK_DELETE deletes the currently highlighted item without the need for a button depression.

Entire blocks can be deleted using the BIGD command which removes the block and all data beneath it.
8. Other Edit Commands

Other editing functions such as MOVE and ALIGN also operate on marked items.

8.1 Moving

Connectors and lines cannot change their relative positions. An item cannot be moved beyond another item. The item that has constrained the move operation is temporarily highlighted, even if that item is at lower level in the hierarchy. To change the relative positioning of items, the item or items must be deleted and reinserted.

8.2 Aligning

It is sometimes necessary to indicate to the assembler that certain pins should be kept fully aligned with each other across the chip when the assembler would not normally do so. Connectors can be specified as being aligned with other connectors on other parallel edges using the ALIGN command.

9. Loading, Writing and Quitting

There are only two ways of loading new data into the current zone, these being EDIT and READ. These commands will only be executed if it is not going to overwrite any edits in the zone that have not been written to a file. To override this protection, a "!" must be appended to the command (eg EDIT!). In general, if a name is specified, that will be the file that is loaded if it exists. If no name is supplied the name of the existing floor plan will be used.

Using the QUIT command will terminate the current process, provided that any changes in the zone have been written out. If this is not the case, a warning will be given. QUIT! will end the session and discard the changes.

10. Additional Features

10.1 Extraction of an Outline

The floor plan editor communicates with the leaf cell editor by using "outlines". The communication information around a block can be extracted for use as a starter for a leaf cell description using the OUTLINE command. Alternatively the outline information from a pre-designed cell or floor plan is displayed when placing a module. Outlines can also be
used as input to the switch box router. The connector information is inferred by the pin names and the router produces a leaf cell containing the appropriate routing.

10.2 Extraction of a Block

The contents of a block can be recursively extracted, using EXTRACT, and written to a file, creating a new module. The name or orientation of the new module can be different to the original.

10.3 Reorientation of a Module

REORIENTATE creates a new module (but keeping the same name) oriented as requested relative to the original. The new module replaces the original.

10.4 Changing the Size of a Module

Although the floor plan has no real dimensions, it is an estimate of the relative size and shape of silicon area that a given function would occupy. It is often necessary to specify an aspect ratio that is not square. A new module is started with a default width and height. This size can be changed using SYSTEMSIZE.

10.5 Inquiring the Dimensions

The command SIZEOF can be used to inquire the dimensions (in the units that the module was started with) of a block or the separation between items.

11. The Data Structure

The key entities that make up floor plans have been described earlier. An explanation of the method of design description, by various references to, and cross references between, such entities follows.

The design of a "composition module" is described in four parts:

Hierarchy description;
Topology description;
Module placements;
Absolute positions list;
11.1 Hierarchy Description

The hierarchy description features the "parent and child" relationship between the blocks making up the composition module. In the example of section 2, blockA is decomposed into blockB, blockC, blockD and blockE. The hierarchy description will contain the following:

- **blockA**: Parent is another block. Sub-blocks (children) are blockB, blockC, blockD and blockE.
- **blockB**: Parent is blockA. Sub-blocks are other blocks.
- **blockC**: Parent is blockA. Sub-blocks are different other blocks.
- **blockD**: Parent is blockA. Sub-blocks are yet more blocks.
- **blockE**: Parent is blockA. Sub-blocks are yet more other blocks.

11.2 Topology Description

To describe the topology of a floor plan all of the lines and all of the connectors are assigned LOGICAL NAMES [7]. In the example the logical names are the integers 1 through to 21.

![Figure 7 All lines and connectors have "logical names"](image_url)
The primary entries in the topology description are LINES. A line is described in terms of its "terminating" lines, and connectors are recorded as features of lines. For example, the entries for lines 11 and 12 in figure 7 look like this:

- line 11: negative end - line 3, positive end - line 2, connectors - 14, 15, 16, 17.
- line 12: negative end - line 11, positive end - line 4, connectors - 18, 19.

11.3 Module Placements

To record the placement of a module, reference is made to the topology description. If moduleP is placed within blockC then the placement description will be:

Placement of moduleP - module block is blockC, bounding lines are line 2 on north, line 12 on south, line 4 on east, line 11 on west.

The data for blockC, within the hierarchy description, will refer to this module placement data.

11.4 Absolute Positions List

For every item in the topology description (i.e., every line and every connector) an absolute position, an "ordinate", is stored. An "x ordinate" is held for each vertical line and for each vertical connector (on a horizontal line). A "y ordinate" is held for each horizontal line and for each horizontal connector (on a vertical line). The ordinates listing for the example of figure 7 might include:

- item 3 ordinate is 0
- item 11 ordinate is 2000
- item 13 ordinate is 1000
- item 14 ordinate is 500
- item 20 ordinate is 1700
- item 21 ordinate is 2700

The combination of the topology description and the ordinates listing provide sufficient information to draw the floor plan.

11.5 Note On Assembly

The function of the assembler is to calculate a suitable
position for every line and every connector - it does not change the design topology (or hierarchy or module placement details either). The results of assembly may therefore be represented simply as a new ordinates list.

11.6 Data Structure Conventions

References to, and cross references between, all structures and strings making up design descriptions are all by LOGICAL NAMES.

12. Starting a New Floor Plan Description

A newly created floor plan description consists of a single block, the "floor plan block", which is named after the floor plan and exists within the world:

\[
\begin{array}{c}
\text{world} \\
\hline
\text{floorplan-block} \\
\hline
\end{array}
\]

Figure 8 A floor plan exists within the world

Consider how the above might be described using the principles of section 11, bearing in mind particularly:

a) All lines are normally described in terms of their terminating lines:

b) Blocks normally contain either a number of sub-blocks, or a single module placement.

Once the above floor plan block has been created subsequent additions may be made using these principles. The floor plan block may be partitioned with further cut lines that will terminate upon those seen in figure 8, and design may proceed as described in section 2.

12.1 Starting the Hierarchy Description

Consider the "world" as the highest level block that can
exist. Now the diagram of figure 8 may be seen as a floor plan of the world block, with partitioning into two regions - the "floorplan-block" and "outside".

In order that the world block does not need to be treated as a special case it needs to be assigned a parent and also four edge lines. The world block refers to ITSELF as its parent and shares edge lines with the floor plan block.

The "top" of the hierarchy description looks like this:

world                Parent is world.
                      Sub-block is floorplan-block.
floorplan-block      Parent is world.
                      Sub-blocks are other blocks.

12.2 Starting the Topology Description

Before a line can be created its two terminating lines are required. So there is a problem in creating the first four lines, those seen in figure 8, that bound the entire floor plan. The solution adopted is to create four PSEUDO lines that exist purely to terminate the floor plan edge lines. The use of the pseudo lines is shown in figure 9. They allow the normal drawing and topology analysis routines to be used without having to recognise the floor plan edge lines as special cases.

The pseudo lines do not have terminators, but do have associated ordinates. Pseudoline1 has the same ordinate as the west edge line, pseudoline2 as the east edge, pseudoline3 and pseudoline4 have the same ordinate as the north edge.
13. The Editor Tally List

Floor plans that have been edited during the current session are retained in working memory so that any one may be quickly recalled. The list that maintains the information required to revert to any of the floor plans is called the "tally list". There is one link in this list for each floor plan, and these are called "edit_tally_links". The link for the floor plan currently being edited is known as the current "tally_link".

The "edit_flag" member of the edit_tally_link indicates the "no write since last change" condition. When a change of floor plan is performed, the edit_flag is tested to determine whether any changes are to be disregarded. If they are then the floor plan is removed from working memory and the tally_link removed from the editing tally list.

14. Topological Position of the Cursor

14.1 Meaning of Topological Position

The "topological position" of the cursor means its position relative to the features (the lines and connectors) of the floor plan being displayed. For example, the topological positions of the points X and Y in figure 10 may be described:

Topological position of point X in figure 10:
Within rectangle bounded by lines 12, 13, 4 and 11 on the north, south, east and west, respectively.

Topological position of point Y in figure 10:
On horizontal line 12.
14.2 Use and Analysis of Topological Position

The reader should be aware that the response to many editing commands depends upon what is pointed at when the command is issued. Further, that when the cursor is moved into the vicinity of a line or connector, within a floor plan, that the feature is highlighted. The position of the cursor with respect to the floor plan features - ie the topological position - is graphically fed back. So the uses of the topological position are:

1) To determine required response to editing command

2) Feed back of position to designer

Analysis of the topological position requires knowledge of the absolute position of the cursor (a coordinate pair), and of the current display. Analysis take place:

1) When ever the cursor is moved - and highlighting is updated as necessary

2) Before any command is executed (the absolute position of the cursor is always received with the command text)

The routine "find_point" uses an algorithm to search, from the bottom left hand corner, through of the current floor plan for the topological position of a given absolute position. The routine "track_cursor" examines the contents of the top_position structure given and performs a few
simple tests in an attempt to find the new topological position somewhere near the old one. If these tests fail, then "find_point" is called to search from scratch.

15. Conclusions

A floor plan editor and a suitable data structure have been described that can create and maintain floor plan definitions. These structures can support both the graphical information required to create a pictorial representation of the floor plan, and the structural information required for simulation and chip assembly.

Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme.

References


The leaf cell editor is an interactive graphical symbolic cell editor for the ASTRA VLSI design system. The screen of the terminal acts as a window into the file currently being edited. The editor has a comprehensive range of editing functions to facilitate iterative refinement of cell designs. It can be used to create, move, modify or delete circuit elements, which consist of transistors, contacts and lengths of interconnect of specified material. The editor maintains an electrical representation which enables checking of circuit, construction and well errors.

1. Introduction

The editor to be described is based on the "virtual grid" [1] concept, where a topological "sticks" description [2,3] is entered on a grid, but the final mask geometry spacing is dependent on design rules. The symbolic design style minimises the overall design time, by abstracting the implementation details whilst retaining sufficient information to make appropriate decisions at the level of interest. By automatically conforming with design rules, design rule changes are reflected in user design data, and detailed designer knowledge is reduced to a minimum.

The spacing program uses a combination of virtual grid and graph based methods and is covered in detail in [4]. Constraint information is stored, and can be accessed from within the leaf cell editor, guiding the designer to reduce the final area of a cell.

This document describes the graphical symbolic cell editor, which forms part of the ASTRA VLSI design system [5]. Details are given of the basic editing functions and the items they manipulate, plus information on the more advanced
features provided by the editor.

2. User Interface

Commands consist of a command word followed by keywords and arguments. There are two types of command, namely long and short commands.

Long commands consist of two types, namely modal and immediate. On entering a modal command (such as add, delete and copy) relevant keywords can be executed without having to enter the modal command word again. An immediate command is not modal and will be executed once only before returning to the original command mode. Once a modal command has been entered, the function performed will apply to the keywords and modes appropriate to that command (exiting briefly to execute any immediate commands) until another modal command is entered.

Short commands are abbreviated command sequences which are assigned in a short commands file, and consist of single keystrokes.

2.1 Display Format

The view of the editor to the user is through a zone, the size and position of which is defined when the process is started. The current process that is running is indicated by a green border around the zone - other processes have red borders. The ASTRA shell control environment is described in [6].

2.1.1 Areas, Zones and Windows

The cell editor zone is divided into at most four different "areas". The aspect ratios determined by the position of the intersection of the areas. Each area is a window to a leaf cell. The cell, when displayed as a stick diagram, is totally enclosed by a cell boundary. A green cell boundary indicates the current area, i.e. the area to which 'window' commands refer. The window to the cell can be changed to view different parts of that cell at different magnifications. The same cell can be displayed in other areas at the same time, yet with different parameters set, enabling the designer to work on different parts of the same cell without the need to keep changing the window parameters. Alternatively, different cells can be displayed in different areas, allowing design work to be carried out on different cells simultaneously.
Commands will be executed on the cell that is displayed in the current area, though in situations where a different area can be identified from a mouse input, the command will refer to that area. Changes made to a cell that is displayed in more than one area will be reflected in all relevant areas.

A grid can be displayed as any integer multiple of grid units. Mouse input is snapped to the nearest grid point that has been set whether the grid is displayed or not. The grid can be viewed as either a grid of lines or a grid of spots. The grid has no real dimensions - its use is to indicate the relative positions of items. It is referred to as a "virtual" grid and it is made up of virtual grid points, or "virgos".

2.1.2 Stick and Log Diagrams

The leaf cell is normally displayed in sticks format. Interconnections are in solid or dashed lines (dashed indicates selected items marked for editing), the colour refers to the interconnection material eg. metal, polysilicon etc. Other symbols represent contacts, transistors and pins. The dimensions of the symbols give some indication of the dimension multipliers assigned to the components (up to a maximum of two grid spaces).

The cell can also be displayed in logs format. This is similar to sticks except that the interconnect is displayed with a width that gives an indication of the width of the track that it represents.

2.1.3 Moving Around the Display

The cursor can be moved around the screen by moving the mouse on the pad. Whilst within the editor zone, pressing any of the three buttons gives coordinate information to the editor process. The coordinates are interpreted as appropriate to the current modal command or to the command assigned to a particular button.

Each area in a zone displays an independent window to a symbolic cell. The parameters of this window (magnification, position, etc) can be changed using a set of simple keywords. The window can be fitted to the boundary, zoomed in, out, moved up, down, left and right. Windows can be stored for restoring later. Plotting can be aborted at any time by a control C.
3. The Data Structure

3.1 Lines, Points and Rectangles

The various elements making up a symbolic cell are stored as line, point and rectangle items. Line items have a start and end point with a specified width. Point items have x and y dimensions. Rectangle items are defined by the minx, miny and maxx, maxy coordinates. Additional information such as material, name, type and comments are stored with the items as necessary. The basic data structure is a linked list of such items with a header holding the name of the cell and boundary information.

3.2 Actual, Selected, Deleted and Error Structures

Four structures are used to describe a cell whilst running the editor. The main data is stored in the 'actual' data structure, where a correctly constructed cell is maintained. Data that has been marked for further editing is transferred to the editing the marked data. After editing, the 'selected' structure holds any newly inserted or modified data, whilst the 'deleted' structure holds any data that has just been removed. This allows the edit to be reversed by exchanging the 'selected' and 'deleted' data structures. The error checker also uses these structures to identify the data that has been modified and updates the 'error' data structure accordingly. The 'error' data structure stores the errors as line, point, and rectangle errors. A track move is shown in Figure 1.

4. Inserting Data

Data can be inserted using the modal 'add' command. Once sufficient keywords have been entered to fully identify the component required, the component is added at the appropriate position. Positional information can be entered as coordinate pairs from the keyboard, in which case the component is added to the cell in the current area. Alternatively, the mouse can be used and the component is added at the position indicated by the cursor - regardless of the current area.
Stages in MOVING a track. Selected data is placed in deleted data structure and transformed with an offset to the actual data structure. An UNDO will switch the SELECTED and DELETED structures.

Figure 1.
4.1 Adding Tracks

A track can be added by changing the item keyword to 'track' and the material keyword to the appropriate material eg. 'metal'. An appropriate track will be added on receipt of two coordinate pairs in the same area that describe a finite length, track within the cell boundary. Subsequent coordinate input can extend the track. If a fine grid makes input difficult, the "SNAP" angle can be varied to allow coarse input to generate horizontal or vertical tracks. The track can be terminated by probing the last point twice. This unselects the track and allows further insertion of the same type of track elsewhere. Changing the track material type also terminates the existing track and allows insertion of tracks of the new material type.

All tracks are added with a default width multiplier unless otherwise specified by a width keyword. In that case once the track is terminated or the material is changed the width will revert back to the default. If, however, the width keyword is preceded by 'default', all subsequent tracks of that type will be added with that width multiplier.

If the insertion of a track forms a transistor by crossing a diffusion track with a polysilicon track, a transistor symbol is automatically inserted. The dimensions of the transistor will be the default transistor dimensions or the track dimensions, whichever is the greater.

If the track overlaps the boundary it will be trimmed and a pin automatically added at the point where the track meets the boundary.

A track can be given a name and this name will be automatically transferred to any pins created by the insertion.

The track, together with any automatically created components, will be added as 'marked'. This enables adjustment of the track, or immediate removal if a mistake has been made. As subsequent tracks are added, previous ones are unselected, and are merged.

4.2 Adding Contacts

A contact can be added by entering the 'contact' keyword. If the material type is already set at the correct material no more keywords are required. If the material is not of the required type or is an invalid contact material, the required material must be entered. The material type 'appropriate' can be used to let the system decide on the material type to be used in unambiguous situations.
On receipt of a coordinate pair, a contact of the required material type will be added, provided the point is within the cell boundary and an appropriate track exists at that point.

The x and y dimensions assume the default unless specified by the multiplier keywords.

All point components are added as marked and can be unselected by probing the same point twice. If a point item of the same type already exists at a point, that point will take the new name and the larger of the point's dimensions.

4.3 Adding Probes and Dominant Nodes

It is convenient in simulation to examine internal nodes of a cell. Probe points are used for this purpose. Dominant nodes indicate which active components in a cell are dominant in sourcing/sinking capability. Probes and dominant nodes can be added in a similar way to contacts, using 'probe' or 'dominant' item keywords. These items do not require dimensions, though probes may have names and types.

4.4 Adding Transistors and Pins

Transistors and pins are created automatically by the system and can subsequently only be modified. Transistor dimensions cannot be smaller than the tracks that have created them.

4.5 Adding Wells

Wells can be added by changing the item keyword to 'well' and specifying the appropriate 'well' material. Two points are required to define the opposite corners of a 'well'. Probing the same corner twice will unselect the item.

4.6 Defaults

The real internal dimensions of items are determined during the spacing stage [4]. The dimension is evaluated by multiplying the size (in microns) specified for a particular type of item in a "default sizes file", by the multiplier specified for a particular item (stored with the item). The initial default multiplier (ie the multiplier that is stored with the item in the absence of instructions to the contrary) is usually unity unless predefined within a "default multipliers file".
5. Selecting and Deleting Data

Editing functions always operate on marked data. The "target components" that are marked can either be items that have just been operated on and hence are still marked, they can be selectively marked using the select commands, or can be operated on directly using the appropriate target keywords with the required edit command.

5.1 Selection

Select is a modal command, allowing unrelated data to be marked prior to a bulk edit, such as 'delete', 'orient', 'move' or 'copy', by staying in select mode and just changing the target keywords. Artifacts such as 'tracks', 'contacts', 'probes' and 'wells' can be selected at a point. Additional keywords allow for selecting all data in the current area, and selection of items within a specified box. A material keyword can optionally be used to identify targets of a particular material.

The select philosophy is to assume that the selected tracks are to be moved or deleted and therefore point items that could be left unconnected are also selected. This also means that transistors and pins cannot be selected independently of the tracks that have created them.

5.2 Unselecting

Data can be unselected by using the unselect command - the default keyword is marked so that all data in the current area, previously marked for editing, becomes unselected.

Specific items and materials can be unselected by using the appropriate keywords.

5.3 Deleting

Data that has been marked for editing can be deleted using the delete command. The target keyword defaults to marked so this keyword need not be explicitly specified. Deletion causes all marked data to be removed from the relevant areas. Data can be retrieved using the undo command.

Target keywords have the same components as select, except that the target components are deleted immediately. These keywords are useful to the experienced user when small corrections need to be made.
6. Other Edit Commands

Other editing functions such as 'move', 'shift', 'copy' and 'modify' also operate on previously marked data or on specific target components; 'tracks', 'part', 'points', 'contacts', 'probes', 'dominants' or 'wells'. The 'orientation' command operates on selected data only. These functions can be reversed by using the 'undo' command.

Data can be moved and copied either to different parts of the same cell or to different cells in other areas.

6.1 Moving

Data to be moved is always marked first, and the modified data appears as marked in its new position. The vector distance moved is calculated from the vector distance between the first and last points used to execute the command. Eg. a 'move part' would mark the part of track within the box defined by the first two points and move it the vector distance between the first point and a third point (input after the data was marked).

When moving tracks, new transistors and pins may be created and tracks may be trimmed to fit in the boundary. If tracks are trimmed or data is moved outside of the cell boundary, a warning is generated. Data lost in this way can only be retrieved by an 'undo'. Attempts to move the data back to its original position will only operate on marked data, ie. the already trimmed tracks and point items that were moved to legal positions. Electrical connectivity is not necessarily maintained and only whole items (not single vertices) will 'move'.

6.2 Shifting

Shifting is very similar to moving except that it attempts to maintain electrical connectivity. If connectivity cannot be maintained (in a diagonal shift for example) the resultant operation is the same as for 'move'.

6.3 Copying

Copying is similar to moving, except that a copy of the original data is unselected before moving the components to the new position. This enables further editing to be performed on the newly copied data without having to re-select it.
6.4 Modifying

The modify command allows the user to alter one of more of the attributes (such as material, name, size etc) of target items.

6.5 Orientation

Data is normally orientated 'north east'. The orientation command transforms selected data about a point into a valid direction ('east north', 'south west', 'north west' etc). Data remains selected after the transformation.

7. Undoing Edits

The 'undo' operation is an immediate command that can be executed at any time without affecting the modality of the previous 'modal' command. 'Undo' provides a quick and easy way of reversing edits made in error. Its prime function is to recover from disastrous mistakes and to ensure that deleted data can be retrieved provided it is used immediately after the mistake occurred. Commands such as 'select' and 'window' that do not remove or create new components cannot be reversed using 'undo'. These actions can be reversed by choice of the correct command (eg 'window last' etc).

The 'undo' command attempts to reverse the last edit command, if it cannot be done, a reason is given. An 'undo' can undone.

8. Duplicating Data

Parts of a cell can be transferred to a new cell (which may be empty) by using the 'copy' or 'move' commands. Alternatively, a cell may be re-edited in another area under a different name. If several views of the same cell are required, the data must be duplicated by the 'area copy' command. This ensures that subsequent changes on that data are reflected in all the relevant areas.

9. Reading and Writing Data

There are several modes of loading new data into the current area, namely 'start', 'edit', 'read', 'lookat' and 'view'. These commands will only be executed if no data will be lost by doing so.
9.1 Loading data

To commence a scratch pad or new cell 'start' is used (provided that the cell does not exist elsewhere). 'Edit' loads data held in a file into the current area. 'Read' ensures that a cell which is to be used as a basis for a new cell does not overwrite the original. 'Lookat' examines the spaced output of a cell in mask generation data form. 'View' allows users to edit a spaced symbolic cell with the display reflecting the real dimensions and separations of the cell.

9.2 Writing to a File

To update or modify files, changes made during an editing session must be written out. By supplying a cell name the data held in the current area will be written out to a file of that name. If the name is different to the original cell name and the file name already exists, a warning will be given and a "!" must be appended to the command to override. If no name is supplied and the modified cell has a name, the data will be written to a file of that name. If writing has been barred by the edit mode, the file will not be written.

10. Additional Features

10.1 Circuit Checking

Each new item added to the data structure is checked to ensure that it is a valid construction for the design [7]. Whilst the editor itself disallows certain circuit configurations, it is necessary to allow certain constructions whilst building up a circuit. Hence errors are generated when floating contacts or dangling tracks are added, the errors are then removed when the particular error is resolved. The error count is maintained for all areas, but the errors reported after each edit operation refer to the current area only.

A specific 'well check' is needed to check for 'well' errors. The errors reported become out of date after the next edit operation and must be re-checked.

Information about errors in a particular area can be gained by using the show errors command: the 'all' keyword reports all errors for a particular circuit; the 'new' keyword shows the errors that have been generated by the last edit. Errors are also displayed graphically to pinpoint their location.

Errors are stored with the circuit when written to a file,
so the circuit need not be complete.

If error checking has been suspended, the whole circuit will be re-checked if necessary when checking is resumed. A warning is generated if the error list is out of date.

10.2 Inquiring the Circuit

The inquire command provides a quick way of interrogating the data for more information about the circuit than is displayed. If there are point items at an inquiry point, only information about those items will be reported. If only track items exist, track information will be reported, and if only wells exist, just well information. If no items exist at the point, the name of the cell displayed in that particular area will be reported, along with the edit mode and whether the circuit has been modified.

10.3 Inquiring the Separation Between Grid Lines

After the spacing process, 'view' reads the spacing cell data information generated by the former, and displays the cell in mask form with non-linear grid spacings. The 'separation' command allows the user to inquire the distance that has been assigned between grid lines and the reasons for the constraint. With this information available, the designer can iteratively modify the cell until it is compact enough for his requirements. After changes have been made, the leaf cell information is written out, spaced and then only the new constraint information is read back into the editor. This allows a user to make changes and to view the effect quickly.

10.4 Inserting Extra Grid Lines

Sometimes it will be necessary to increase the size of a cell boundary in order to add more details. A convenient way of doing this is to insert extra grid lines at the required position. This procedure is called 'tear', and is effected by specifying two grid points, indicating the position, direction and length of 'tear'. Two-dimensional tears are possible.
10.5 Removing Redundant Grid Lines

It is often convenient, to tidy up the display, to remove redundant grid lines from the circuit. All or specific redundant grid lines can be removed by using the 'squeeze' facility.

10.6 Identifying Electrically Connected Items

The 'span' command marks all items that are electrically connected to a specific item. This allows the node to be inspected or to be removed entirely from the data structure, and as such can be used as an extension of select.

10.7 Removal of Redundant Items

It is occasionally useful to be able to remove redundant items automatically. The 'nibble' command will trim back dangling tracks and remove redundant contacts from the circuit. Pins and transistors are removed where necessary. Dangling tracks and incomplete contacts are detected by the circuit checker.

10.8 Control of Display

The 'plotability' command gives the user some control over the data that is displayed on screen, and also the way in which it is displayed. A choice of keywords gives the option of not plotting certain items, to give a clearer display; plotting certain items as filled or unfilled; or plotting tracks as 'logs', giving some indication of track dimensions.

11. Conclusions

The leaf cell editor provides a process-independent data capture environment, where users do not require detailed design rule knowledge. The process independence means that different mask layouts can be readily generated by simply changing design rule files, which is particularly useful in cases where manufacturers change process characteristics whilst a designer is still in chip layout phase.
Acknowledgments

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme. Our thanks to Dr. P A Ivey and Dr. M C Revett for their joint management of the ASTRA work, and the many individuals who have contributed towards the leaf cell editor especially R T Scarfe and R Weeks.

References

1. Weste, N
   "Virtual Grid Symbolic Layout",
   Proc. 18th Design Automation Conf.,
   June 1981

2. Williams, J D
   "STICKS - A New Approach to LSI Design",

3. Williams, J D
   "STICKS- A Graphical Compiler for High Level LSI Design"

4. Scarfe, R T
   "The Leaf Cell Spacer",
   The ICD Book (Edition 1), Delft University Press,
   The Netherlands, Jan 1986.

5. Revett, M C
   "The ASTRA System : Symbolic Design of Custom VLSI"
   The ICD Book (Edition 1), Delft University Press,
   The Netherlands, Jan 1986.

6. Roe, B J
   "The ASTRA Shell Control Environment"
   The ICD Book (Edition 1), Delft University Press,
   The Netherlands, Jan 1986.

7. Weeks, R
   "The Interactive Leaf Cell Checker"
   The ICD Book (Edition 1), Delft University Press,
   The Netherlands, Jan 1986.
THE INTERACTIVE LEAF CELL CHECKER

R Weeks
British Telecom Research Laboratories

ABSTRACT

This document describes the features of the interactive leaf cell checker that forms part of the symbolic leaf cell editor within the ASTRA VLSI design system.

1. Introduction

An interactive leaf cell checker is an integral feature of the symbolic leaf cell editing facility [1], which forms part of the ASTRA VLSI design system [2]. By virtue of its close association with the symbolic editor, the leaf cell checker is able to provide an immediate means of error detection and correction. The close coupling of the leaf cell checker with the editor ensures that, from within an editing session, it is possible to generate 'correct' leaf cell data that is guaranteed to be acceptable to all the programs that subsequently operate on the leaf cell data.

2. Facilities

The user sees the leaf cell checker as offering four distinct facilities:

a) a construct checker whose concern is the legality and completeness of each symbolic item with respect to any co-located items;

b) a well checker that checks the relationship between well regions, their anti-latchup contacts, and the presence of items improper to specific well regions;

c) a circuit spanner that highlights all items electrically connected to a specified item;

This paper was produced as part of Task VI of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.

5.52
d) a circuit checker that fully resolves the leaf cell's electrical connectivity and then identifies unusual circuit configurations.

The construct checker can be operated in a continuous mode, in which each editing operation is automatically checked. Used in this mode, the checker offers immediate feedback to the user about the legality of editing operations. Alternatively, the construct checker can be operated in an on-demand mode, in which it is effectively disabled except when the user requests a construct check to be done. This mode of operation is particularly appropriate when editing causes the number of errors to transiently become large (and of little interest) until further editing cancels out the majority of the errors.

The other checking facilities (the well checker and the circuit checker) do not offer a continuous mode of operation. To do so could impose a significant CPU time penalty - simple editing operations can totally modify a leaf cell's electrical topology. Furthermore, well errors and circuit errors are likely to exist until a late stage in the evolution of a leaf cell's design; the continual reporting of such errors is therefore of little benefit.

3. The Construct Checker

The construct checker operates in a process-independent manner, driven from data files which it reads during its initialisation phase. These files contain rules for each item acceptable to the process; the absence of a rule for an item indicates that the item is not valid for the process rules in force. The rules for each item - its error 'profile' - are expressed in terms of the acceptability of other co-located items.

Extra rules are used to describe the special checks that have to be performed at the end of each track in order to detect whether a track end is 'dangling', and whether an editing operation has 'satisfied' the dangling track end. The data that drives the construct checker is therefore conveniently divided into two files - a main 'profile' file, and a subsidiary 'dangler' file.

3.1 The Main Construct Checker

Appendix I lists a small part of the contents of a typical error profile driver file appropriate to a two-level-metal CMOS process. The format of the file is quite free - the text that appears in parentheses is ignored by the file parser.
The file contains a profile for each item (apart from wells, which are dealt with by the well checker). The format for a profile entry is:

item
number of errors that can be generated by the item's addition
qualifier item status error-number
qualifier item status error-number
etc

number of errors that can be resolved by the item's addition
qualifier item error-number
qualifier item error-number
etc

An item's profile is capable of representing the following facts:

a) the addition of an item may generate an error because another item, that should be present, is missing. For example, when a 'contact mp' is added it requires a 'track poly' to be present.

b) the addition of an item may generate an error because another item, that conflicts with it, is also present. For example, when a 'contact mp' is added, the presence of a 'contact vdd' is illegal.

c) the addition of an item may remove an error because it satisfies the requirements of an existing item. For example, the addition of a 'track poly' is able to remove the error generated by an unsatisfied 'contact mp'.

d) qualifiers may be applied to items under certain circumstances to define the item more closely. Valid qualifiers are:

* - no further qualification
v - only match a vertical track
h - only match a horizontal track
c - only match a co-linear track
u - only match a track going upwards
d - only match a track going downwards
l - only match a track going left
r - only match a track going right

Note that, in the error profile file, errors are identified by unique numeric codes. The textual information in the error profile file is only present to improve its readability; it is not necessarily the same text that is presented to the user when the errors are decoded. The translation of the error's code into a text form is carried out when the error is notified to the user - these

5.54
translations are held in a separate error message file.

An important feature of the error profile file is that the same entry can be interpreted to apply to both the addition and deletion of an item. Just as the addition of an item can both generate and resolve errors, so can the deletion of an item.

Using examples (b) and (c) as before, but applying them to the inverse operation, it can be seen that the same error profiles can be used 'in reverse':

bb) the deletion of an item may resolve an error because the presence of another item, that conflicts with it, is no longer invalid. For example, when a 'contact mp' is deleted, the presence of a 'contact vdd' ceases to be illegal.

cc) the deletion of an item may generate an error because the requirements of an existing item are no longer satisfied. For example, the deletion of a 'track poly' may generate an error due to the presence of an unsatisfied 'contact mp'.

A close inspection of the error profile file might suggest that some of its entries are superfluous. For example, the fact that the leaf cell editor automatically generates transistors at diffusion/polysilicon intersections would suggest that the explicit specification in the error profile file that a transistor needs a polysilicon track is not strictly necessary. However, the presence of such entries in the error profile file serves two purposes: it guards against the unexpected side-effects of complex editing operations, and it allows for the fact that the leaf cell may not originally have been created via the leaf cell editor but may have come from a completely different (and possibly less scrupulous) source.

3.2 The Dangling Error Checker

Appendix II lists part of the contents of a typical dangling error driver file appropriate to a two-level-metal cmos process. Again, the format of the file is quite free - the text that appears to the right of the entries is ignored by the file parser.

The file contains, for each type of track, a list of other items any of which, if present at the end of the track, will satisfy a dangling track end error. Then follows, for each type of point item that can affect dangling errors, a list of dangling errors that the point item can satisfy.
As with the error profile file, errors are identified by unique numeric codes, and, again, the same entry can be interpreted to apply to both the addition and deletion of an item.

4. The Well Checker

At any stage of a leaf cell editing session, the leaf cell may be checked to see if there are any errors associated with its wells, anti-latchup contacts, and diffusion tracks. The well checker is capable of operating with any single-well process that employs either n-wells or p-wells, or a two-well process employing both types of well. In this section, these processes are referred to as 'n', 'p', and '2' respectively.

The well checks can be divided into the five distinct groups shown below. The associated error number for each check is indicated.

4.1 Anti-latchup Contact Positions - First Check

Each anti-latchup contact must be in the correct position with respect to the well regions. In detail:

p: each VSS contact must be in a p-region
    each VDD contact must be outside a p-region

n: each VDD contact must be in a n-region
    each VSS contact must be outside an n-region

2: each VSS contact must be in a p-region
    each VDD contact must be in an n-region

4.2 Anti-latchup Contact Positions - Second Check

Each region must have at least one appropriate anti-latchup contact. In detail:

p: each p-region needs at least one VSS contact
    each non-p (x-) region needs at least one VDD contact

n: each n-region needs at least one VDD contact
    each non-n (x-) region needs at least one VSS contact

2: each p-region needs at least one VSS contact
    each n-region needs at least one VDD contact

5.56
4.3 Diffusion Track Positions

No part of any diffusion track may be co-located with an inappropriate type of well. In detail:

p: no pdif track must be inside a p-region
no ndif track must be outside a p-region

n: no ndif track must be inside an n-region
no pdif track must be outside an n-region

2: no pdif track must be outside an n-region
no ndif track must be outside a p-region

4.4 Well Geometry - First Check

No two regions may meet purely at a single point unless other wells mask the point-contact effect. In detail:

p: no two p-regions may have point contact

n: no two n-regions may have point contact

2: no two p-regions may have point contact
no two n-regions may have point contact
no p-region and n-region may have point contact

4.5 Well Geometry - Second Check (Two-well Process Only)

Different types of well must not touch or overlap. In detail:

2: no p-well and n-well may touch
no p-well and n-well may overlap

5. The Spanner

The user is able to highlight an electrical path within a leaf cell by selecting any point on that required path and requesting a 'span' to be performed. The spanner recursively 'walks' along all adjoining tracks of the same material and traverses the associated contacts to walk along the other tracks associated with the contacts. In terms of the leaf cell's electrical topology, the spanner can be considered to select a 'node'. Another function of the spanner is to form the basis of the leaf cell circuit connector, which in turn forms part of the leaf cell circuit extractor [3]. Circuit connection is achieved by repeated application of the spanner until no more tracks remain to be spanned. As each node is identified by the spanner, it is

5.57
assigned a unique node number.

It is not until the electrical connectivity of the leaf cell has been resolved in this manner that the final set of checks - the 'circuit' checks - can be performed. These circuit checks are described in the next section.

6. The Circuit Checker

When the electrical connectivity of a leaf cell has been resolved (see previous section) a group of checks can be carried out in an attempt to spot 'circuit' errors. The circuit checker performs the following checks:

6.1 Reserved Name Checks

The names 'vdd', 'vss', and any name starting 'phi...', are considered to be reserved names.

For every node, a check is made on all the names associated with that node. If the names include more than one reserved name, or a reserved name plus a non-reserved name, an error message is issued.

6.2 Probe Name Checks

An error message is issued for each probe name that is associated with more than one node.

6.3 Anti-latchup Contact Checks

Each anti-latchup contact must be electrically associated with the correct type of power rail - a 'vss' contact with a 'vss' rail, a 'vdd' contact with a 'vdd' rail. An error message is issued for each invalid anti-latchup contact.

6.4 Checks for Isolated Circuit Segments

A track is deemed to be isolated if it cannot 'see' a pin or the node of a transistor that itself can see a pin. An error message is issued for each isolated track segment.

6.5 Checks for Shorted Transistors

An error message is issued for each transistor that has any two of its electrodes shorted together.
6.6 Floating Gates

A transistor's gate is deemed to be floating if it only 'sees' other transistors' gates. An error message is issued for each transistor gate that is isolated in this manner.

7. Error Reporting

The leaf cell editor offers two methods for reporting the errors detected by the construct, well, and circuit checkers. There is a graphical indication scheme, whereby the errors are highlighted together with their error numbers. The second, independent, method of error reporting displays the text messages associated with the errors.

The error numbers thus serve two purposes. Not only are they used as part of the internal operation of error checkers, but they also serve as a useful shorthand representation for the graphical error reporting method.

A further advantage of separating the internal error representation from the text messages lies in the freedom it offers for easy modification of the messages, on a user basis if necessary.

Another use of the error texts is to indicate error severities. This is a particularly useful facility when dealing with those types of error that cannot be rigidly assigned a fixed error severity.

For example, a well region reaching to the edge of a leaf cell but without an anti-latchup contact may or may not be considered to be an error, depending on circumstances. These circumstances could depend on the contents of adjacent cells, or even on the user's style.

Appendix III shows a list of error messages that is consistent with the error profile and danger files of Appendices I and II.

8. Other Applications of the Circuit Checker

The previous sections have explained the role of the leaf cell checker as an interactive error reporting mechanism. This section outlines some further uses of the checker, both within the symbolic leaf cell editor and outside it.

8.1 Circuit Nibbling

The leaf cell editor incorporates a 'nibbler' that
automatically pares back unwanted parts of the circuit. The capability of the construct checker to detect superfluous circuit items is harnessed within the nibbler.

In essence, the nibbler inspects the construct errors, looking for partially satisfied contacts and dangling track ends. It then nibbles these items away. The check-nibble sequence is repeated until no further nibbling can be performed.

8.2 The Spanner

As already indicated, the spanner forms the core of the leaf cell circuit extractor. The circuit extractor is part of a suite of programs that provide the path to simulation [3] and certain classes of hierarchical consistency checking.

8.3 Checking within Other Programs

The leaf cell checker is also invoked from within several other programs. In some cases the programs are ones that feed off the symbolic leaf cell editor's output - the spacer [4] and the circuit extractor programs [3] are in this category. In these programs, the leaf cell checker needs to be invoked both to report any outstanding errors, and to ensure that the checks are up to date.

In other cases, the leaf cell checker is associated with programs that may feed into the symbolic editor, or even be completely independent of it - programs that manipulate man-readable leaf cell representations for inter-system data transfer, for example. In these programs, the leaf cell checker needs to be invoked in order to check the circuit in place of the facilities offered within the symbolic editor.

9. Internal Operation

A general appreciation of the internal data representations of the symbolic leaf cell editor permits a broad comprehension of the way the leaf cell checker is integrated into the editor. In the context of the storage and manipulation of error representations, the following points are relevant:

a) The leaf cell data is held in one main data structure - the 'actual' structure, which corresponds to the normally displayed circuit. Two other data structures are used to hold transient data. The 'selected' structure holds data that is in a state of change, as the result of an insertion or modification operation, perhaps. There is also a
'deleted' structure, which holds newly deleted data, enabling an 'undo' mode for many classes of operation.

b) Data within each structure is represented in the same manner. Separate sub-divisions of the data structure are used to store point items, line items and rectangle items.

In terms of these points, the integration of the error checker is achieved by:

a) Holding the errors in a further data structure - the 'error' structure, which has an identical format to the other structures. The errors can then automatically be manipulated along with the other data. Most importantly, they can easily be transformed by editing operations, and be written out to the same file as the rest of the data.

b) Representing the errors as point, line, and rectangle errors. Fortunately, for most errors this is a natural segregation.

The various parts of the error checker behave differently upon invocation. The continuous element of the checker (the construct checker) is fed by the selected data, each element of which can be considered as an independently added item, and by the deleted data, each element of which can be considered as an independently deleted item. Thus the construct checker is guaranteed to be able to deal with any editing operation, however complex.

Operation of the 'on-demand' parts of the error checker (the well checker and the circuit checker) is simpler. They do not have to deal with transient circuit states, and can ignore all the data in the error structure, simply considering the circuit to be the combination of all the data in the actual and selected structures.

10. Conclusions

Extensive operational experience of the leaf cell checker has proved its capability to provide a useful 'on-line' error checking mechanism. For all but the most complex of editing operations on large leaf cells, the operation of the construct checker in its continuous mode does not significantly impede the symbolic leaf cell editor's speed performance.

Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of
the EEC under the 3744/81 Microelectronics programme. My thanks go to G R McMullan for all his effort in improving the original well checker algorithm and implementing it in 'C'.

References

APPENDIX I - FRAGMENTS OF A TYPICAL ERROR PROFILE FILE

contact mp
6 (number of 'errors generated' entries)
* track metall required 3 (no metall track for mp contact)
* track poly required 4 (no poly track for mp contact)
* contact mdn illegal 11 (clash vs mp and mdn contacts)
* contact mdp illegal 12 (clash vs mp and mdp contacts)
* contact vdd illegal 13 (clash vs mp and vdd contacts)
* contact vss illegal 14 (clash vs mp and vss contacts)
0 (number of 'errors resolved' entries)

track poly
6 (number of 'errors generated' entries)
* contact mdn illegal 21 (clash vs poly track and mdn contact)
* contact mdp illegal 22 (clash vs poly track and mdp contact)
* contact vdd illegal 23 (clash vs poly track and vdd contact)
* contact vss illegal 26 (clash vs poly track and vss contact)
c track ndif illegal 30 (clash vs co-linear poly and ndif tracks)
c track pdif illegal 31 (clash vs co-linear poly and pdif tracks)
8 (number of 'errors resolved' entries)
* contact mdp 4 (no poly track for mp contact)
v vtransistor n 32 (no poly track for n-type v transistor)
h htransistor n 35 (no poly track for n-type h transistor)
v vtransistor p 38 (no poly track for p-type v transistor)
h htransistor p 41 (no poly track for p-type h transistor)
* probe poly 46 (no poly track for poly probe)
* dominant poly 51 (no poly track for poly dominant node)
* cap poly 59 (no poly track for cap poly)

track ndif
5 (number of 'errors generated' entries)
* contact vdd illegal 24 (clash vs ndif track and vdd contact)
* contact vss illegal 27 (clash vs ndif track and vss contact)
c track pdif illegal 29 (clash vs ndif and pdif tracks)
c track poly illegal 30 (clash vs co-linear poly and ndif tracks)
* contact mdp illegal 55 (clash vs mdp contact and ndif track)
8 (number of 'errors resolved' entries)
* contact mdn 6 (no ndif track for mdn contact)
l vtransistor n 33 (no ndif-l track for n-type v transistor)
r vtransistor n 34 (no ndif-r track for n-type v transistor)
u htransistor n 36 (no ndif-u track for n-type h transistor)
d htransistor n 37 (no ndif-d track for n-type h transistor)
* probe ndif 47 (no ndif track for ndif probe)
* dominant ndif 52 (no ndif track for ndif dominant node)
* cap ndif 60 (no ndif track for cap ndif)
APPENDIX II - FRAGMENTS OF A TYPICAL DANGLING ERROR FILE

metal2  101  2  track-type error-number  number-of-associated-items
  contact via  associated item
  pin metal2  associated item
metal1  102  7
  contact via
  contact mp
  contact mdn
  contact mdp
  contact vdd
  contact vss
  pin metal1
poly  103  5
  contact mp
  htransistor n
  vtransistor n
  htransistor p
  vtransistor p

contact via  2  item-type  number-of-associated-dangling-errors
  101  102  the error numbers
  102  103
contact mp  2
  102  103
contact mdn  2
  102  104
contact mdp  2
  102  105
contact vdd  1
  102
contact vss  1
  102
pin  metal2  1
  101
pin metallic  1
  102
pin poly  1
  103
pin ndif  1
  104
pin pdif  1
  105
htransistor n  1
  103
vtransistor n  1
  103
htransistor p  1
  103
vtransistor p  1
  103

5.64
APPENDIX III - A TYPICAL ERROR MESSAGE FILE

1. missing meta12 track for via contact
2. missing metall track for via contact
3. missing metall track for mp contact
4. missing poly track for mp contact
5. missing metall track for mdn contact
6. missing ndif track for mdn contact
7. missing metall track for mdp contact
8. missing pdif track for mdp contact
9. missing metall track for vdd contact
10. missing metall track for vss contact
11. clash between mp and mdn contacts
12. clash between mp and mdp contacts
13. clash between mp and vdd contacts
14. clash between mp and vss contacts
15. clash between mdn and mdp contacts
16. clash between mdn and vdd contacts
17. clash between mdn and vss contacts
18. clash between mdp and vdd contacts
19. clash between mdp and vss contacts
20. clash between vdd and vss contacts
21. clash between poly track and mdn contact
22. clash between poly track and mdp contact
23. clash between poly track and vdd contact
24. clash between ndif track and vdd contact
25. clash between pdif track and vdd contact
26. clash between poly track and vss contact
27. clash between ndif track and vss contact
28. clash between pdif track and vss contact
29. clash between ndif and pdif tracks
30. clash between co-linear poly and ndif tracks
31. clash between co-linear poly and pdif tracks
32. missing poly track for n-type v transistor
33. missing ndif-l track for n-type v transistor
34. missing ndif-r track for n-type v transistor
35. missing poly track for n-type h transistor
36. missing ndif-u track for n-type h transistor
37. missing ndif-d track for n-type h transistor
38. missing poly track for p-type v transistor
39. missing pdif-l track for p-type v transistor
40. missing pdif-r track for p-type v transistor
41. missing poly track for p-type h transistor
42. missing pdif-u track for p-type h transistor
43. missing pdif-d track for p-type h transistor
44. missing meta12 track for meta2 probe
45. missing metall track for metall probe
46. missing poly track for poly probe
47. missing ndif track for ndif probe
48. missing pdif track for pdif probe
49. missing meta12 track for meta2 dominant node
50. missing metall track for metall dominant node
51. missing poly track for poly dominant node
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>missing ndif track for ndif dominant node</td>
</tr>
<tr>
<td>53</td>
<td>missing pdif track for pdif dominant node</td>
</tr>
<tr>
<td>54</td>
<td>clash between mdn contact and pdif track</td>
</tr>
<tr>
<td>55</td>
<td>clash between mdp contact and ndif track</td>
</tr>
<tr>
<td>56</td>
<td>missing metall track for mon metall</td>
</tr>
<tr>
<td>57</td>
<td>missing metal2 track for cap metal2</td>
</tr>
<tr>
<td>58</td>
<td>missing metall track for cap metall</td>
</tr>
<tr>
<td>59</td>
<td>missing poly track for cap poly</td>
</tr>
<tr>
<td>60</td>
<td>missing ndif track for cap ndif</td>
</tr>
<tr>
<td>61</td>
<td>missing pdif track for cap pdif</td>
</tr>
<tr>
<td>101</td>
<td>end of metal2 track dangling</td>
</tr>
<tr>
<td>102</td>
<td>end of metall track dangling</td>
</tr>
<tr>
<td>103</td>
<td>end of poly track dangling</td>
</tr>
<tr>
<td>104</td>
<td>end of ndif track dangling</td>
</tr>
<tr>
<td>105</td>
<td>end of pdif track dangling</td>
</tr>
<tr>
<td>211</td>
<td>VSS contact is not inside a p-well</td>
</tr>
<tr>
<td>212</td>
<td>VDD contact is not outside a p-well</td>
</tr>
<tr>
<td>213</td>
<td>VDD contact is not inside an n-well</td>
</tr>
<tr>
<td>214</td>
<td>VSS contact is not outside an n-well</td>
</tr>
<tr>
<td>221</td>
<td>no VSS contact for p-well area including rectangle</td>
</tr>
<tr>
<td>222</td>
<td>no VDD contact for non p-well area including rectangle</td>
</tr>
<tr>
<td>223</td>
<td>no VDD contact for n-well area including rectangle</td>
</tr>
<tr>
<td>224</td>
<td>no VSS contact for non n-well area including rectangle</td>
</tr>
<tr>
<td>231</td>
<td>pdif track is inside a p-well</td>
</tr>
<tr>
<td>232</td>
<td>ndif track is outside a p-well</td>
</tr>
<tr>
<td>233</td>
<td>ndif track is inside a n-well</td>
</tr>
<tr>
<td>234</td>
<td>pdif track is outside a n-well</td>
</tr>
<tr>
<td>241</td>
<td>p-wells have point contact</td>
</tr>
<tr>
<td>242</td>
<td>n-wells have point contact</td>
</tr>
<tr>
<td>243</td>
<td>p- and n-well have point contact</td>
</tr>
<tr>
<td>251</td>
<td>p-well and n-well touch</td>
</tr>
<tr>
<td>252</td>
<td>p-well and n-well overlap</td>
</tr>
<tr>
<td>261</td>
<td>n-well invalid for this process</td>
</tr>
<tr>
<td>262</td>
<td>p-well invalid for this process</td>
</tr>
<tr>
<td>301</td>
<td>missing metal2 track for metal2 pin</td>
</tr>
<tr>
<td>302</td>
<td>missing metall track for metall pin</td>
</tr>
<tr>
<td>303</td>
<td>missing poly track for poly pin</td>
</tr>
<tr>
<td>304</td>
<td>missing ndif track for ndif pin</td>
</tr>
<tr>
<td>305</td>
<td>missing pdif track for pdif pin</td>
</tr>
<tr>
<td>401</td>
<td>probe name associated with more than one node</td>
</tr>
<tr>
<td>402</td>
<td>bad pin name (reserved name clash)</td>
</tr>
<tr>
<td>403</td>
<td>bad probe name (reserved name clash)</td>
</tr>
<tr>
<td>404</td>
<td>metal2 track totally isolated from outside world</td>
</tr>
<tr>
<td>405</td>
<td>metall track totally isolated from outside world</td>
</tr>
<tr>
<td>406</td>
<td>poly track totally isolated from outside world</td>
</tr>
<tr>
<td>407</td>
<td>ndif track totally isolated from outside world</td>
</tr>
<tr>
<td>408</td>
<td>pdif track totally isolated from outside world</td>
</tr>
<tr>
<td>409</td>
<td>vdd alu contact not on vdd rail</td>
</tr>
<tr>
<td>410</td>
<td>vss alu contact not on vss rail</td>
</tr>
<tr>
<td>411</td>
<td>transistor channel connected between vdd and vss</td>
</tr>
<tr>
<td>412</td>
<td>transistor has source connected to drain</td>
</tr>
<tr>
<td>413</td>
<td>transistor has gate connected to source or drain</td>
</tr>
<tr>
<td>414</td>
<td>transistor gate floating</td>
</tr>
</tbody>
</table>

5.66
THE LEAF CELL SPACER

R T Scarfe

British Telecom Research Laboratories

ABSTRACT

This paper describes the leaf cell spacer used within the ASTRA VLSI Design suite. The spacer converts the symbolic leaf cell into design rule correct physical layout using rules described in a technology file. In addition, the leaf cell spacer identifies those edges that will be significant to the process of cell abutment, during chip assembly.

1. Introduction

The symbolic cell editor [1] within the ASTRA suite [2], captures a leaf cell design on a virtual grid [3]. The leaf cell spacer converts the cell to a minimum sized design rule correct physical layout by determining the row and column separations. In addition, the leaf cell spacer identifies those edges that will be significant to the process of cell abutment, during chip assembly [4].

The leaf cell spacer is completely independent of all technology and process rules. It converts the symbolic cell to spacer objects, and imposes constraints between those objects according to design rules for a specific process, as defined in 'the_rules_file'.

2. Circuit Conversion

The symbolic cell describes a leaf cell as a collection of transistors, contacts, pins and connecting tracks on a virtual grid.

The spacing function assigns to each circuit element nodal information. Related elements are assigned the same node number. The spacer permits same node and different node constraints.

The elements placed by the leaf cell are converted and sized to point, line, area, edge and corner objects of particular

This paper was produced as part of Task VI of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
materials and node number.

2.1 Nodal Evaluation

Nodal evaluation is performed across an entire leaf cell.

2.2 The Point Object

The point object is a centre defined rectangle, of particular material with a width, height and one or more node numbers, both the width and height are fixed from the symbolic dimension and the conversion sizing. For example, a CMOS transistor is a point object with three nodes, gate, source and drain.

2.3 The Line Object

The line object is a material, of a fixed width, between a pair of grid points with a node number. All lines are either vertical or horizontal. The width is determined from the symbolic dimension and the conversion sizing. The length of the line depends upon the grid spacing between its defining points.

Lines may be terminated by a point of the line material. The dimension of the point is the maximum of the line and any point that may exist there.

2.4 The Area Object

The area object is a corner defined rectangle of a material. The corner points are located at grid points; the internal dimensions are determined entirely from the grid spacings defining the area. For example, a CMOS well is an area object.

Areas are merged to produce edges, with an inside and an outside, and corners.

2.5 The Edge Object

The edge object is defined by two points, as for the line object, with an inside and outside, and a node number. Edges are the result of merging area objects. Edges with the same node describe the bounding polygon of a set of merged areas.
2.6 The Corner Object

The Corner object is defined at a point, as for the point object, with an inside and outside characteristic, and node number. Corners are the result of merging area objects. Corners with the same node describe the corner points of a bounding polygon of a set of merged areas.

Conversions from symbolic cell elements to spacer objects is described in 'the rules file'. The sizes of the objects are derived as a combination of the symbolic size and the conversion factors (multipliers and adders), and the node numbers are transferred from the symbolic elements to the spacer objects. For example a contact to n-diffusion may be converted thus:

```
POINT CONTACT NDIFFUSION POINT METALL 0 1.5 0 1.5 1 1 1
POINT CONTACT NDIFFUSION POINT NDIFFUSION 0 2.0 0 2.0 3 3 3
POINT CONTACT NDIFFUSION POINT WINDOW 0 0 0 0 1 1 1
```

The contact is converted to a point of metal with additional x and y sizing of 1.5 and the metal node number, a point of diffusion with additional x and y sizing of 2.0 and the material node number, and a contact cut with no sizing and the metal node number.

3. Circuit Constraints

The design rules for a particular process are described as constraints between materials, with the same node or different node, plus reference to the actual design rule e.g.

```
RULE POLYSILICON EXTERNAL NDIFFUSION EXTERNAL
SAMENODE 2.5 10.1 POLY to NDIF same node
DIFFERENTNODE 2.5 10.2 POLY to NDIF different node
```

These rules specify that polysilicon objects will be separated from all n-diffusion objects by 2.5 units irrespective of node number. The design rules are referenced as 10.1 and 10.2 and commented.

Parallel lines of the same material may be permitted to merge, where as parallel transistors, with the same node numbers, which would otherwise be permitted to merge, can be constrained from one another by defining a conversion from transistor to non-merging points.

4. Imposing Constraints

The major feature of any spacing process is identification
of all necessary constraints. The virtual grid, used in leaf cells simplifies identification. Grid lines will be spaced by the minimum distance required to satisfy all the constraints required between objects on the grid lines. The spacing routines passes through the grid three times to space adjacent grid lines, non-adjacent grid lines and diagonally separated grid points.

4.1 Imposing the Adjacent Constraints

The first phase separates adjacent grid lines. Objects with a common grid ordinate on adjacent grid lines are constrained. The spacing imposed gives an absolute minimum distance between the grid lines. Edges, lines, points and corners are spaced.

4.2 Imposing the Non-adjacent Constraints

Objects with a common grid ordinate on non-adjacent grid lines are constrained. The number of grid lines traversed during this phase is determined from the start object size, the particular constraint, the current grid spacing and possible target object size - the maximum for each grid line is recorded. Edges, lines, points and corners are spaced.

4.3 Imposing Diagonal Constraints

During this phase only the objects defined at a point are considered.

Objects with no common grid ordinate are considered. The range of grid line traversal is determined, in each direction, as for the non-adjacent constraints.

5. Enforcing Constraints

Assigning real space between adjacent grid lines is the function of the constraints enforcement procedure. The adjacent constraints are enforced immediately, and give a useful indication of the minimum inter-grid line spacing.

Diagonal constraints may be solved in a number of methods:

a). Converted to a constraint in the x-axis
b). Converted to a constraint in the y-axis
c). Converted to a constraint in the axis that increases the total area of the cell by the least amount.

Non-adjacent constraints are converted to adjacent constraints. The grid lines are traversed, as each grid line
is reached unresolved constraints with previous grid lines are resolved by forcing space before the grid line.

A compaction point may be positioned by a designer. The compaction is used as a starting point for the leaf cell spacer and hence it works like a gravity point "pulling" circuit elements to it. The cell spacer may re-position the compaction point so that no non-adjacent constraint traverse it.

6. Stretch Lines

To facilitate cell abutment during mask data generation [5], each leaf cell has a stretch line adjacent to each boundary and between each pair of pins. The stretch line indicates where additional spacing, as requested by the chip assembler, may be imposed.

The chip assembly program has sole responsibility for identifying all the constraints necessary to fit a cell into its environment. Any additional spacing is imposed at the stretch line adjacent to the leaf cell boundary.

Pins may be separated, by the assembly program, to pitch match the cells environment. All additional spacing, for this purpose, will be placed at the stretch line between the two pins. Thus all points between a pair of stretch lines will 'move' with the pin.

Stretch lines may be placed manually or generated automatically. All materials that cross a stretch line will be extended, thus the position of stretch lines may be critical. To this end 'the rules file' contains weighting for each material, for example, in CMOS stretching of metal is usually prefferable to stretching in polysilicon or diffusion. When considering the generation of a stretch line, the number, width and weighting of materials is combined.

7. Significant Edge Generation

Edges that may be significant to cell abutment are identified by the cell spacer. For each boundary of the leaf cell, all edges referred to in 'the_rules_file' that may be significant are output.

The chip assemble program may request extra spacing within a cell. Hence particular attention is paid to generating both edges that are significant and those that may become significant. For example a line that crosses a stretch line may become significant as the items closer to the boundary are moved apart to facilitate cell abutment.
8. Conclusions

The spacer methodology makes it both technology and process independent. The flexibility of the spacing algorithm is directly related to the imagination of the composer of 'the rules file'. Several sets of single level CMOS design rules have been implemented. Rules files for NMOS and double level metal CMOS processes will be written when required.

Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme. The author would like to thank the other members of the ASTRA development team for their advice and encouragement.

References

THE CHIP ASSEMBLER

P R Benyon
British Telecom Research Laboratories

ABSTRACT

This document describes the chip assembly program used in the ASTRA VLSI design system. The program is built upon a constraint based data structure which is formed from a hierarchical floor plan and a set of stretchable cell outlines. The program is technology and design rule independent, and its output may be to generate design rule correct mask data.

1. Introduction

The design of a VLSI integrated circuit using the ASTRA system [1] is based upon the interconnection of rectangular blocks with specific connection points arranged in a hierarchical floor plan. The lowest levels of blocks in a completed design place cells.

The purpose of the chip assembler is to assemble all or part of a design by combining a hierarchical floor plan, which defines interconnection and relative block placement, with the minimum dimensions, connecting pin positions and edge material data for the cells placed. The program creates a layout floor plan where all dimensions are real and all lowest level blocks place stretched versions of cells. This new floor plan can then be used to generate mask data for the design [2]. An important requirement of this program is that it should be design rule and technology independent.

![Diagram of chip assembler process]

Figure 1 The function of the chip assembler

This paper was produced as part of Task VI of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
2. Input Considerations

In this section, the three basic types of input data required by the program are described.

2.1 Hierarchical Floor Plan Data

A hierarchical floor plan is an essential part of the ASTRA design philosophy. It represents a useful method of user-to-program and user-to-user communication. It defines how cells are to be connected together, and may be generated in one of two basic forms, LIBRA text language files or floor plan data structures which are created by the floor plan editor [3].

2.1.1 Why use a Floor Plan?

The use of a floor plan to structure a design has many advantages, the most important of which are:-

1. The designer is encouraged to partition the design in a hierarchical manner at an early stage of development. The interface between the individual designers can also be defined at this stage.

2. Interconnection is considered at every stage of the design rather than being left until the major blocks have been designed - interconnection is 'managed' from the start.

3. Each block may be simulated in isolation or after connection has taken place, although care must be taken to ensure that the cells are not stretched so as to change their function.

4. In many cases, fault finding and correction is easy because changes may be made to a small part of the design prior to the re-assembly of the affected areas.

5. The floor plan data can be checked against the placed cells to ensure that the cells obey certain simple composition rules.

The floor plan hierarchy used has single block-and-cell pairs at the lowest levels. This form of partitioning is a particular case of a 'separated hierarchy' [4]. Cells and blocks may be used more than once, with many facilities being available for disconnecting pins and reorientating cells.
2.1.2 The LIBRA Language Files

LIBRA (Layout of Interconnected Blocks as Rectangles that Abut) is a text language developed by British Telecom. Its primary use is to define hierarchical floor plans.

A LIBRA floor plan consists of a named rectangular block composed of a set of named rectangular sub-blocks with interconnection pins along their shared edges. As such the floor plan defines both the structural connection of the constituent functional sub-blocks as well as their relative placement. A graphical representation of the floor plan can be produced automatically from the LIBRA language data.

A LIBRA floor plan is hierarchical in form, with each sub-block being itself a block defined in an equivalent way. Each block is contained in a separate file, the name of the file being derived from the name of the block.

In addition to the normal use of the language to specify floor plans as input data, the language was extended to be used as an interface format between several of the ASTRA programs (chip assembler, mask data generator[2], cell spacer[5]).

For further information on the syntax of LIBRA the reader is referred to the documentation on the LIBRA language [6].

2.1.3 Floor Plan Data Structures

Floor plan data structure files are written in a machine readable ASCII format [7]. Unlike LIBRA, the data is divided up into modules, each module being either a cell or a simple hierarchy of blocks. For each module, the chip assembler is given information for the top level module and the lowest level blocks that may place other modules. Information regarding any intervening hierarchy in a module is not passed to the assembly software.

The topology of the floor plan is also passed to the chip assembler which consists of a top or 'world' level block partitioned using a series of cut lines. Each cut line may have a number of connectors associated with it, which in turn have layer, direction and name information associated with them.

A design may consist of many modules, some placed repeatedly, possibly but not necessarily as parts of arrays.

The structures are quick to load and convenient to use and form the primary form of input to the chip assembler. Interface routines to create the structures without the aid

5.75
of the floor plan editor have also been written and could be used in automatic floor plan generation programs.

Programs are available to convert LIBRA to floor plan data structures and vice versa. The only restriction is that each stage of chip assembly should use data in only one of the two acceptable forms.

2.2 Cell Minimum Outline Data

Cell minimum outline data is generated by the grid-based cell spacer program [5] for each cell placed at the lowest levels of the floor plan. The cell spacer applies the rules to the symbolic cell information and generates a file to tell the chip assembler everything that it needs to know when connecting the cell to other cells. The file is written in extended LIBRA language.

The information passed in these files is as follows:

1. Cell dimensions

The dimensions of the cell boundary are included using a WIDTH and a HEIGHT statement. The rectangular boundary of the cell does not have to enclose all the elements of the cell but simply represents the smallest area that any stretched version of the cell can occupy in that the boundaries of adjacent cells must abut. All connecting pins of a cell lie on this boundary.

2. Pin data

The position, width, layer and name of any connecting pin is included using a PIN statement. Pins always lie on the cell boundary and are certain minimum distances from others adjacent to them in the x-dimension or the y-dimension. The width and layer data specifies how wide the connecting track is at the pin and the type of material(s) from which it is made. The pin name specifies the name of the pin used in the cell.

3. Significant edge data

The position, material and electrical node of any mask rectangle edge that may be significant when design rules are applied between cells placed close to one another is written out as an EDGE statement.
An example of a LIBRA file for a cell is given below:

```
DEFINE BLOCK simple WITH NORTH[1] EAST[0] SOUTH[1] WEST[0]
WIDTH 0
HEIGHT 0
PIN NORTH[1] AT 0 WITH LAYER=METAL1 WIDTH=400 NODE=1
PIN SOUTH[1] AT 0 WITH LAYER=METAL1 WIDTH=400 NODE=1
EDGE WEST OF EXTERNAL METAL1 WITH POSITION=SOUTH[1]-200 &
END[SOUTH]=SOUTH[SIDE]+0 END[NORTH]=NORTH[SIDE]+0 NODE=1
EDGE EAST OF EXTERNAL METAL1 WITH POSITION=SOUTH[1]+200 &
END[SOUTH]=SOUTH[SIDE]+0 END[NORTH]=NORTH[SIDE]+0 NODE=1
EDGE WEST OF EXTERNAL PWELL WITH POSITION=WEST[SIDE]+O &
END[SOUTH]=SOUTH[SIDE]+0 END[NORTH]=NORTH[SIDE]+0 NODE=2
EDGE SOUTH OF EXTERNAL PWELL WITH POSITION=SOUTH[SIDE]+0 &
END[WEST]=WEST[SIDE]+0 END[EAST]=EAST[SIDE]+0 NODE=2
EDGE EAST OF EXTERNAL PWELL WITH POSITION=EAST[SIDE]+0 &
END[SOUTH]=SOUTH[SIDE]+0 END[NORTH]=NORTH[SIDE]+0 NODE=2
EDGE NORTH OF EXTERNAL PWELL WITH POSITION=NORTH[SIDE]+0 &
END[WEST]=WEST[SIDE]+0 END[EAST]=EAST[SIDE]+0 NODE=2
END simple
```

2.3 Design Rule and Technology Data

The design rule and technology data is taken from a single text file normally referred to as 'the_rules_file'. This file specifies the design rules to be used to space out cell elements, the layers required for mask making and any parameters used in post-processing such as minimum grid size.

The data is also used to drive the cell spacer and chip assembler, allowing them to be independent of technology. Although the chip assembler is primarily used for CMOS design, it is not restricted to such processes in any way.

The data may be altered by editing an existing file or by creating a new file. The ASTRA user selects the desired process using a special 'set_process' utility.
3. Process Considerations

This section describes the process of chip assembly in terms of the data structures created and the basic algorithms used.

3.1 Brief Description of the Process

The operations performed by the chip assembly program can be summarized in pseudo code form as follows:

```
begin
  read floor plan data into a constraint based data structure
  store constraints for lowest levels as determined by spacer
  solve constraint graph by determining graph node coords
  do until a stable solution is found
    compute additional stretching that is required at cell boundaries to prevent all design rule violations
    solve constraint graph by determining graph node coords
    compute additional stretching that is required to route around rigid blocks or rigid cells
    solve constraint graph by determining graph node coords
  enddo
  write out stretching instructions in the form of a stretched floor plan which can be used to create mask data
end
```

The data structure must be such that the constraint graph can be solved very quickly and therefore a great deal of attention has been given to this aspect of program design.

3.2 The Constraint Based Data Structure

The chip assembler uses its input data to create a data structure based upon the minimum distance constraints that must be imposed between the pins and edges of the minimum sized cells. A data structure in this form is usually referred to as a constraint graph.

3.2.1 Cell Pins and Constraints

For each block in the floor plan, the chip assembler stores two sets of constraint based data elements in its data structure, one set for the pins and edges in the x-dimension and the other set for the pins and edges in the y-dimension.
When the block places a cell at the lowest level of the floor plan hierarchy, each element may have a constraint value associated with it, together with a pointer to one other constraining pin or edge.

In all cases the constraints must be arranged so that the layout will still be legal with respect to the design rules after stretching has taken place. Since a cell is stretched by spacing the virtual grid lines on which it is designed, it follows that the constraints must be imposed between adjacent pins considering both sides of the cell at once as shown in Figure 2.

Cells can be re-orientated in any of the eight possible orientations. The orientation transformation is applied to the input data before the pin and edge elements are stored. The method used to specify orientation follows the convention adopted by Rosenberg and Weste [8].
3.2.2 Rigid Blocks and Rigid Cells

Any block or cell placement in a floor plan hierarchy may be classed as 'rigid'. This simply means that the block or cell being placed is to be connected to others using simple river routing tracks around all four sides rather than being stretched as for non-rigid placements.

If a cell is placed rigidly, then the data for the minimum sized cell can be taken from the cell data file as for an ordinary cell placement. However, if a block is placed rigidly, then the data for the minimum sized block must be taken from the data file created by the chip assembler when the block itself was assembled since it is only this file that contains the necessary information. It follows that blocks placed as 'rigid blocks' need to be assembled before they are used at a higher level.

The constraints for a rigid placement differ from those for a cell in that pins on opposite sides do not necessarily have to be kept in a set order due to the flexibility afforded by the routing channels. These constraints are shown in Figure 3.

![Figure 3 The constraints for a rigid placement](image-url)
It should be noted that the chip assembler has been optimised to give the best results when cells are stretchable. Nevertheless, the concept of rigid placement does allow users to route blocks where necessary.

3.2.3 Blocks and Hierarchy

The hierarchical floor plans used in ASTRA allow many levels of blocks to be defined above the lowest levels. These blocks may also have pin and edge data elements associated with them, although they will have no constraint values.

3.2.4 Holes

The data structure permits an area of floor plan to be designated as a 'hole' which is a lowest level block that does not have any contents. A hole may be unconditionally stretched by its environment and is especially useful when part of a design is missing or when a routing cell is to be designed.

3.2.5 Pin Connections and Alignments

Each floor plan pin connection associates two connecting pins together and specifies that the pins must share the same coordinate value in the final design layout. When a block is not a lowest level block, the connection also associates the connecting pins at lower levels in the design hierarchy. When two blocks abut, then two block sides must be associated in a similar fashion.

To achieve this association of pins and sides, an alignment data structure element is created which associates a linked list of pins or sides with a single coordinate value. Solving the constraint graph involves setting these values so that the constraints are satisfied.

The concept of an alignment is a very useful one, so useful in fact that the floor plan input formats have been designed to allow users to add their own pin alignments should they wish to constrain two pins to line up, even when they are not normally constrained to do so. It should be noted that it may not always be possible to carry out a users' requirements, if the resulting constraint graph becomes insoluble or cyclic.
3.2.6 Significant Edges

When cells are joined together, it is necessary to ensure that design rules are not violated. This problem has been tackled by many people in many different ways with varying degrees of area efficiency [9,10].

To achieve a compact solution in situations where cells are being stretched, it is necessary to know about the characteristics and positions of the edges of all cell elements that can interfere with each other when the design rules are applied. It is also necessary to keep track of where the edges are as stretching takes place, since moving a pin or an edge to a new position may also move the edges of some of the cell elements, potentially generating new design rule violations.

The ASTRA cell spacer writes out a set of 'significant' edges for each cell. This information is then loaded into the data structure in such a way that the positions of the edges are related to the position of the aligned pins and edges and move with them by definition.

The significant edge concept is fairly easy to understand and although the amount of data involved can be large, the computations involved in solution are simple and fast.

Each significant edge is associated with a single cell side and is significant in a particular direction and for a particular material. The position of the edge is fixed with respect to three pins or edges of the data structure, the distances involved being constants or minima when measured from block sides or constants only when measured from pins.

Edges may be unconditionally aligned with block sides, and this allows the chip assembler to cancel out the edges of abutting wells. Edges may also have electrical node numbers associated with them which allows the chip assembler to find out which edges are electrically connected.

3.2.7 Missing Pins

In a multi-level hierarchical floor plan, there are many situations where it is desirable to remove a connecting pin from a cell or a block. It may be that a cell is required in two different places with different connections or that an array of blocks is required with fewer external connections than internal ones.

One approach to this problem would be to force the user to define different cells for each situation. However, this is unlikely to be a desirable solution both in terms of the
amount of data involved and the numbers of frustrated users!

The ASTRA user may disconnect a pin at any level of the hierarchy and in any situation.

Where pins are disconnected from a lowest level cell placement, then instructions are stored in the layout plan files to tell the mask generator that the connecting track is to be moved away from the boundary of the cell. This operation is termed 'nibbling'.

Where pins are disconnected from a lowest level rigid placement, then instructions are stored to tell the mask generator to miss out a routing track and make repairs to the potentially violating track ends where necessary.

Where pins are disconnected at a higher level in the hierarchy, then the chip assembler removes all associated pins down to those at the lowest level. This operation is termed 'gobbling'.

All these operations are performed as the data structure is created so that the algorithms to solve the constraint graph only deal with pins that actually form electrical connections.

3.2.8 Composition Checking

The data structure is subjected to a number of composition checks before being passed on to the solution routines. Pin data stored in the floor plan is checked against the cells and the cells are also checked against each other to ensure that only legal connections take place.

3.3 Solving the Constraint Based Data Structure

After creating the constraint based data structure and reporting any errors in the input data, the chip assembler attempts to solve the constraint graph that the data structure represents. In fact two graphs are being used at this stage, one for each dimension, with the floor plan topology being used to determine the direction of stretch when any diagonal constraints are solved.

3.3.1 Pitch Matching

For each node the constraint graph will have a set of constraints associated with it as extracted from the lowest level placements.
The constraint graph is such that the position of some nodes can only be determined after the positions of certain other nodes have been evaluated. Therefore, the nodes are ordered in two linked lists so that each node is only dependent upon the nodes before it in its list.

The strategy used for solving the graph is to set to zero the coordinates of the first nodes, which correspond to the leftmost block sides of the design in the x-dimension and the lowest block sides in the y-dimension. The position of the other nodes is then evaluated in turn, with the x-dimension positions being considered first.

3.3.2 Boundary Violations

The positions of the significant edges for the lowest level blocks are used to add extra constraints along the block boundaries so that design rule violations are prevented. These extra constraints are then added to the constraint graph.

When the violation involves two edges, one of which is unconditionally aligned with a boundary, then the chip assembler puts the extra constraint along that boundary. These are termed 'internal' violations. When the violation involves two edges, neither of which is unconditionally aligned with a boundary, then the chip assembler splits the extra constraint into two equal parts and imposes these between the edges and their associated sides. These are termed 'external' violations.

Violations are investigated in a particular order, short-range violations between edges whose sides are aligned being considered before long-range violations which may traverse several cells. Internal violations are always investigated before external violations.

3.3.3 Routing Rigid Blocks and Rigid Cells

When a rigid placement is encountered, the chip assembler must determine how to river route the cell or block placed. This routing is done after the pitch matching has been done and boundary violations have been prevented.

Unlike a soft placement, the distances between the pins of the minimum sized cell or block do not represent the minimum permitted distances between the routing tracks surrounding the placed entity. These distances are determined by the rules that apply between the routing tracks themselves. Therefore, the constraints involved in a rigid placement can be modified by the chip assembler to reflect the constraints
of adjacent blocks and this process reduces the overall layout size.

The process of routing involves aligning a pin across a routing channel for each dimension and working out how much space is to be left for the routing. These extra constraints are then added to the constraint graph.

Further stretching may be necessary to prevent boundary violations created when the routing was added, and the assembler carries on until a stable solution is attained.

4. Output Considerations

The output from the chip assembler falls into two categories, layout floor plan data and user diagnostics.

4.1 The Layout Floor Plan Data

This is a floor plan where all dimensions are real and where the lowest level blocks place stretched versions of the cells. The data is written out in pseudo-LIBRA language and includes all the information required to tell the mask data generator how to stretch and route cells and blocks and create mask data in a number of different languages. The use of this format means that the generation of mask data is not done unless the results of assembly are favourable, and this can constitute a very significant time saving.

When the constraint graph has been solved, the chip assembler looks at every block and works out which blocks are the same in name, pin positions and placements. The program then generates a LIBRA file for every unique block. This means that the number of LIBRA files generated for a block that is placed more than once will vary depending upon its environment. This allows the assembler to minimise chip area especially where arrayed blocks are stretched irregularly.

For further information, the reader is referred to the documentation on the mask data generator [2].

4.2 Blotch Plots

A blotch plot attempts to show where and by how much the placed cells have been stretched past their compacted sizes in a graphical form. It does this by showing areas of fully compacted cell in red and areas of stretched cell in dark grey. A plot of the cell outlines is then superimposed upon this to show the pins responsible for stretching.
This form of plot enables a user to find the cells that are constraining their environment by looking for simple features such as very red areas which show the cells that are constraining their neighbours and grey lines which show a section of layout that is being held up or out to the right.

Blotch plots can be generated to show stretching in one or two dimensions.

5. Conclusions

The chip assembler has been used to assemble a few designs, consistently producing design rule correct layout. The stretching techniques used reward good design with compact layout, and the predictable nature of both the grid based cell spacer combined with the graph based chip assembler allows a user to improve cells with a fair degree of confidence in the results.

Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme. I would like to thank the other members of the ASTRA development team for their advice and encouragement.

References


Laboratories, Nov. 1982.


THE AUTOMATIC ROUTE TO SIMULATION IN THE ASTRA SYSTEM

R Weeks
British Telecom Research Laboratories

ABSTRACT

This document describes the suite of programs that enable the automatic generation of simulation input data within the ASTRA VLSI design system.

1. Introduction

The generation of simulation input files from ASTRA leaf cell and floor plan data, usually entered via ASTRA's graphical editors [1,2], (which form part of the ASTRA VLSI design system [3]), is largely an automatic process. The simulation data is drawn from precisely the data that is subsequently used in the path to mask generation, to guarantee that the simulation accurately reflects the eventual physical circuit performance.

Although the noble aim is a total automation of the process of translating source data into simulator input files, several difficulties impede the achievement of this goal; unfortunately the resolution of all these problems does not easily lie within the scope of software remedies. In particular, the performance of the simulators themselves cannot generally be tailored to the requirements of the ASTRA environment. Another area of difficulty is the generation of waveform and other control inputs. Generally these problems tend to increase with the number of simulators served, since their restrictions are usually different, and often mutually exclusive.

Nevertheless, it has been possible to automate the translation process within the ASTRA environment to a large extent. The main features are:

* a single 'auto-translate' command that generates an output file after running the minimum number of intermediate programs to ensure that the output data is

This paper was produced as part of Task VI of the ICD project which was received partial support by the Commission of the EEC under the MR-09 contract.
fully up to date

* intermediate data representations for leaf cell circuits and floor plan structures tuned to the requirements of subsequent translation to simulation languages, to avoid the need for unnecessary, time-consuming, circuit extraction operations

* rigorous leaf cell and floor plan checks, including name consistency checks, to prevent the simulation of inconsistent data

* an intermediate translation stage common to all simulation language translations that feeds a final translation module for specific languages. Translation modules are currently in use for the circuit simulation language, SPICE, and for the logic simulation language, HITEST, which is used predominantly as a switch-level simulator. Modules for other hierarchical simulators could be provided with little effort.

The essence of the simulation translation philosophy is that the user is presented with the single 'auto-translate' command whose detailed operation can be concealed. However, a broad understanding of the internal operation of the translation software suite can give an insight into the overall performance and restrictions of the simulation translation facility.

2. Constituent Parts

The translator is invoked in its most complex mode when a floor plan, or a block within a floor plan, is to be translated to a circuit-level simulation language. Under these circumstances, circuit representations of several leaf cells and floor plans must be combined in a hierarchical manner along with physical information (process- and layout-dependent parasitic capacitances, for example).

At the other end of the complexity scale would rate the translation of a single leaf cell to a switch-level simulation language. In this case no hierarchical or physical layout information need be considered.

The general interaction of data with programs both inside and outside the translator suite is shown below.
For the sake of clarity, several subsidiary files have been omitted from the diagram. These files include:

* a rules file giving process-dependent details
* a file of default sizes for the leaf cell elements
* the message file produced by the structure extractor
* the message file produced by the translator
* (for HITEST) a list of reserved names
* (for HITEST) a generated skeleton waveform file
* (for HITEST) a generated skeleton names file
* (for SPICE) a file containing transistor models
* (for SPICE) an input file containing control statements

Figure 1 General arrangement of programs within the translator suite
The circuit extractor generates an electrical representation of each leaf cell in the floor plan to be simulated in a net-list form (to give a group of data files marked 'a'). The capacitance extractor generates a list of transistor dimensions and parasitic capacitances for each leaf cell (to give a group of data files marked 'b'). The division of tasks between the circuit and capacitance extractors is clear-cut; the circuit extractor only deals with process-independent items, and the capacitance extractor deals with all the other items.

(Before the capacitance extractor can be run, the leaf cells must be 'spaced' according to the process rules. This task is achieved by the spacer [4], which forms part of the chip assembly/mask generation path rather than being considered as part of the translator program suite. The spacer produces the group of data files 'c'.)

The structure extractor generates a hierarchical electrical representation of the floor plan to be simulated. It needs to refer to the leaf cells placed within the floor plan in order to be able to perform consistency checks at floor plan boundaries. This is conveniently done by reference to the extracted circuit files, which are far more quickly accessed than the prime leaf cell data, and contain the required information in a more readily usable form. The structure extractor is process-independent and therefore does not need to refer to the data generated by the capacitance extractor.

The translator proper is driven by the output from the structure extractor together with the outputs from the circuit and capacitance extractors. Two important tasks are performed before control is passed to a specific language translation module:

a) the connectivity of the whole hierarchical circuit is resolved from the leaf cell level upwards. In essence, the operation involves the removal of any duplicated signal paths at all the boundaries within the circuit (called 'pin reduction' - see section 6.2). By this means the initial geometrical nature of the circuit is transformed into an electrical representation acceptable to simulators.

b) signal names are checked at each boundary as an attempt to detect unintentional connections.

Finally, the appropriate translation module is then invoked to generate the output file that is to be presented to the simulator.

All the above programs and data are manipulated by a single 'auto-translate' command. In order to generate an up-to-date
simulation file, various parts of the software are automatically and selectively invoked, depending on the date relationships of the various prime and intermediate files.

The following sections describe the constituent parts of the translator in more detail.

3. The Leaf Cell Circuit Extractor

An early part of the translation process is the generation of an electrical representation of each leaf cell that contributes to the floor plan to be simulated. These intermediate representations, one for each leaf cell, represent the electrical behaviour in a net-list form that is sufficiently general to be translated to specific simulation languages at a later stage.

From the programming point of view, the provision of such an intermediate data representation aids the division of the overall translation task into its several distinct modular components. The benefit to the user is also substantial; any re-translation following leaf cell modifications only forces those leaf cells that have actually been modified to undergo the circuit extraction process again.

The circuit extractor only deals with the process-independent parts of a leaf cell's electrical behaviour, leaving the capacitance extractor to handle the process-dependent aspects. Separation of the leaf cell extraction task in this manner, with the circuit and capacitance extractors generating physically independent outputs, has two main benefits:

* capacitance extraction need only be performed when the target simulation language demands it, thus a HITEST translation can be achieved without the unnecessary time penalty of capacitance extraction;

* redundant, process-independent data is not generated if translation to more than one process is required - although multiple versions of the capacitance extractor's output would by necessity have to exist, only one version of the circuit extractor's output would need to be generated.

Nor is there any benefit in combining the two extractors from an algorithmic point of view; their operations have little in common.
3.1 The Circuit Extractor's Operation

Four distinct phases can be discerned in the circuit extractor's operation:

1) reading in the leaf cell data
2) checking the leaf cell and reporting errors
3) generation of pin coincidence information
4) writing the various sections of the output file

The circuit extractor is able to call on a group of three leaf cell checkers [5] to bring the circuit's error status up to date. Only those checks (if any) that are out of date are performed, before the extractor reports all the errors that have been detected.

Resolution of the leaf cell's electrical connectivity is achieved by repeated applications of the circuit spanner [5], which is invoked as part of the circuit checking phase. The connectivity is achieved by the assignment of unique numbers to electrical nodes.

3.2 The Output File

In its output phase, the circuit extractor writes five main sections to its output file. Broadly, these sections contain:

1) a header
2) pin coincidence information
3) pin names and node numbers
4) transistor types and node numbers
5) probe names and node numbers

Appendix I shows the results of the circuit extraction of a small leaf cell.

Although the format of the output file is mainly tuned to the requirements of the translator, consideration has also to be given to the fact that also the structure extractor uses the circuit extractor's output. The structure extractor's demand is simple; it just needs to have the numbers of pins and pin coincidences for each side of the leaf cell. For this reason, the relevant pin information occurs early in the output file, enabling the structure extractor merely to 'glance' at the file.

The only section that merits a more detailed description is the pin coincidence section, which is used to indicate the physical co-location of pins of more than one material. Although pin co-location is of no significance when a single leaf cell is to be simulated, it may be of vital importance.
when a hierarchical translation is to be performed. Without knowledge of pin coincidences, the structure extractor cannot deal with the disconnection of pins at the boundaries with orientated modules.

Pin coincidence is represented by a linear array of digits, one for each pin. For a solitary pin, the digit is zero. Each coincident group of pins takes on a unique positive integer which is assigned to all pins in the group. This convention is demonstrated in the following example which shows some of the possible configurations for five pins along the edge of a leaf cell. Where pins are coincident, the number of pins is indicated below the pin group.

<table>
<thead>
<tr>
<th>pin configuration</th>
<th>coincidence representation</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-X-X-X-X-X-X-</td>
<td>0 0 0 0 0</td>
<td>no coincident pins</td>
</tr>
<tr>
<td>--X--X-X-X-</td>
<td>1 1 0 0 0</td>
<td>pins 1&amp;2 coincident</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------X-------</td>
<td>1 1 1 1 1</td>
<td>pins 1-5 coincident</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--X--X--X--</td>
<td>1 1 0 2 2</td>
<td>pins 1&amp;2, 4&amp;5 coincident</td>
</tr>
<tr>
<td>2 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--X---X----</td>
<td>1 1 2 2 2</td>
<td>pins 1&amp;2, 3-5 coincident</td>
</tr>
<tr>
<td>2 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2 Some examples of pin-coinidence representation

The final case shows how a simple boolean indication of whether a pin is coincident or not would be unable to cope with the case of neighbouring groups of coincident pins.

4. The Leaf Cell Capacitance Extractor

When simulation at a circuit-level is required, the capacitance extractor must be brought into play, operating on each relevant leaf cell. The capacitance extractor operates in a similar manner to the circuit extractor, producing one intermediate output file for each leaf cell. Again, the representation of extracted data in an intermediate format has the twin benefits of language independence and the minimisation of unnecessary recomputation.

By its very nature, capacitance extraction is a process-dependent procedure, depending heavily on the process
environment. Whereas circuit extraction is simply achieved by a single operation on the prime leaf cell circuit data, capacitance extraction is more complex in two aspects. First, it is a two-stage process, requiring the spacer to have previously been run. Second, the capacitance extractor needs to read the various files that contain the process-dependent information (as does the spacer program itself).

Thus the capacitance extractor has to bind together several types of information about the leaf cell. It combines the geometrical and electrical information from the prime leaf cell data with the physical row and column separations derived by the spacer, taking into account the parasitic effects appropriate to the process to be simulated. Since the output from the capacitance extractor is fully scaled, the translation stage proper is able to operate in a completely process-independent manner.

4.1 The Capacitance Extractor's Operation

Four distinct phases can be discerned in the capacitance extractor's operation:

1) reading in the leaf cell, spacer, and control files
2) computation of transistor channel areas and perimeters
3) computation of other (linear) track capacitances
4) writing the various sections of the output file

To a certain extent the operation of the capacitance extractor is aimed towards the requirements of the SPICE simulator, although this bias is reflected more in the format of the output file than in the program's internal operation.

4.2 The Output File

In its output phase, the capacitance extractor writes three sections to its output file. Broadly, these sections contain:

1) a header
2) transistor geometry details
3) linear inter-node capacitances

To take advantage of SPICE's transistor modelling capability, the area and perimeter of the source and drain of each transistor is computed. By modelling the parasitic effect of transistor diffusion in this manner, its non-linear nature can be taken into account. Where a number of transistors share an area of diffusion, that diffusion's area and perimeter are equally apportioned between the
contributing transistors.

All other capacitances - those from track to track and from track to substrate, including capacitances for diffusion that is not associated with any transistors - are modelled as linear capacitors.

Appendix II shows the results of the capacitance extraction of the same small leaf cell used as the example for Appendix I.

4.3 The Capacitance Extraction Algorithm

A major problem in the selection of a suitable capacitance extraction algorithm is the striking of a sensible balance between the accuracy of the generated results and the processing time required. Fortunately, extreme accuracy is not needed. Device measuring and modelling inaccuracies and spreads in process parameters all contribute to uncertainties in the simulation results. Refining the capacitance extraction algorithm too far would therefore be of no benefit.

The capacitance extraction algorithm can therefore be based on the requirement to produce a 'reasonably' accurate representation of parasitic effects. In any case, a very accurate representation of the physical layout would require generation of the mask data itself (in general requiring assembly of the circuit under consideration), whereas simulation is intended mainly to be used to aid the earlier circuit design stages.

Briefly, the operation of the capacitance extractor is as follows:

1) in a first pass through the leaf cell data, all items are given true positions and dimensions, and a list of all x- and y-locations of the edges of all the leaf cell's elements is built up;

2) a two-dimensional grid of 'boxes' is dynamically allocated. The edges of each box map to adjacent pairs of the x- and y-values from stage 1. Each box is initially empty, but is capable of indicating the presence or absence of each material layer that the circuit contains;

3) a second pass through the leaf cell data builds up the contents of the boxes. The method of generation of the boxes ensures that a material layer is simply present in, or absent from, the box - it is not possible for a box only to be partially full of a material;
4) information for transistor source and drain areas and perimeters is gathered by a recursive 'walk' starting from the channel of each transistor;

5) the contributions for all the other parasitic capacitances are summed over the whole grid system.

5. The Structure Extractor

Structure extraction lies at the core of the translation process. It gathers information about a floor plan's electrical structure from all the disparate floor plans and leaf cells that must finally be combined to generate input to the simulator. All the structure information is written to a single intermediate file which is then used to drive the translator proper.

The circuit extractor must be exercised on all the leaf cells that form part of the floor plan to be simulated before structure extraction is attempted. This is because the structure extractor needs to 'glance' at the leaf cell data in order to glean information about the leaf cell's pins and their co-incidences.

An important part of the structure extractor's operation is its checking phase. Several types of inconsistencies can exist in the hierarchical circuit data, mainly as a result of the multiple-file nature of the data. Broadly speaking, the checks carried out by the structure extractor can be considered as a subset of those associated with the chip assembler [6], but tuned to the requirements of the simulation path. Nevertheless, the structure extractor's checks are important in their own right because it is not normally convenient or appropriate to invoke the assembler before a simulation can be carried out, purely to check for the relevant classes of error.

5.1 The Structure Extractor's Operation

Six distinct phases can be discerned in the structure extractor's operation:

1) reading in the floor plan and leaf cell data
2) linking and ordering the floor plans and their blocks
3) checking pin consistencies at floor plan boundaries
4) reporting all errors (and terminating if necessary)
5) upwards propagation of pin coincidence information
6) writing the various sections of the output file

In its reading phase, the structure extractor recursively takes in floor plan information, starting with that for the
specified parent floor plan. After this process the structure extractor is aware of all the leaf cells that are invoked, and it glances at all the leaf cell data.

The linking phase consists of forming links between each circuit block and its children, which may be other blocks in the same floor plan, other floor plans, or leaf cells. As a result of the linking procedure, the blocks are placed in a suitable order for generation of the output file - an order in which the lowest level blocks appear first, ensuring that each block is defined before it is invoked.

After the linking phase, it is possible to perform a complete consistency check on the structure. If any fatal errors are detected, the structure extraction process terminates at this stage. A description of all the checks performed is given in the next section.

The next phase involves the application of the array, disconnection, and orientation information at module boundaries. This process has to take into account the pin coincidence information which is initially provided by the leaf cell data and subsequently made to propagate upwards.

5.2 Structure Checking

The structure extractor's checker has a similar relationship to floor plans as the leaf cell checker has to a single leaf cell. The main difference lies in the fact that structure checking is not provided as an 'on-demand' facility from inside the floor plan editor because of the need to access all the contributing floor plans and their associated (circuit-extracted) leaf cells. Thus the structure extractor not only serves to form part of the translation chain, but also stands in its own right as structure checker.

The following classes of check are performed.

5.2.1 Recursion

During its linking phase the structure extractor can detect if a cyclic sequence of floor plan placements is implied.

The message format is:

"*** recursion involving floor plans ... <list>"

5.2.2 Problems at Floor Plan Boundaries

There are three kinds of error that can occur at a module...
boundary. All these errors are reported with a common message header with the following format:

"*** errors at module boundaries within floor plan <name> in <orientation> pxq placement of floorplan|leafcell <name> by block <name>"

(the placement is of an array of $p$ x-elements by $q$ y-elements.)

The errors that can occur are:

1) the numbers of pins simply don't match. The message format is:

"because block <name> has $n$ pins and $n_2$ discons (= $n_3$ total) but the placed module <name> offers $n_4$"

(where $n_3 = n_1 + n_2$, but $n_4 \neq n_3$)

2) an invalid discon number has been specified. The message format is:

"because discon <side><number> is outside valid range of $1->\text{max-discon}"

3) the module cannot be arrayed as requested because there are unequal numbers of pins on its opposing sides. The message format is:

"because of insufficient pin symmetry"

5.2.3 Incomplete, Poorly-defined, or Bad Floor Plans

Error messages will be generated if floor plans or leaf cells are missing, or if floor plans are incomplete (empty blocks), have undefined placements, or the top-level block does not have the same name as the floor plan itself. The message formats are:

"*** these floor plans are absent ..."
"*** these leaf cells are absent ..."
"*** these floor plans are incomplete ...
"*** these module placements are of undefined type ...
"*** these floor plans have bad contents ...

5.2.4 Name Clashes

Certain classes of name clashes between floor plans and leaf cells, and between blocks within floor plans, are reported as warnings. They do not prohibit the structure extractor
from continuing, but may indicate unintentional name duplication, and can be awkward for the translator to handle. The message formats are:

"*** these are both floor plan and leaf cell names ..."
"*** these block names occur in several floor plans ..."

5.4 The Output File

In its output phase, the structure extractor writes three main sections to its output file. Broadly, these sections contain:

1) a header
2) leaf cell information
3) floor plan information

The leaf cell information in the output file is used to tell the translator which leaf cells to translate. The translator gains all its other data from within the output file itself.

Appendix III shows the results of the structure extraction of a simple floor plan that places two instances of the leaf cell used as Appendix I's example.

6. The Translation Phase Proper

After circuit and structure extraction (and possibly capacitance extraction as well) the derived data is in a suitable form for translation to a simulation language. This task is achieved in the translator proper, which operates in two main phases. The first phase is a language-independent pre-processing stage that carries out all the general tasks that can be performed before control is passed to a language-specific output module. The second phase consists of the execution of the appropriate output module to generate the files to be passed to the simulator. The division of tasks in this manner, with the removal of as much common processing as possible from the final translation phase, paves the way for the easy introduction of extra language modules, with the existing modules being able to serve as 'fleshy templates'.

6.1 The Translator's Operation

Four distinct phases can be discerned in the translator's operation:

1) reading in the structure and leaf cell data
2) pin reduction and connectivity propagation
3) name consistency checking
4) executing a language module

The two intermediate phases are discussed below in more detail.

6.2 Pin Reduction and Connectivity Propagation

The connectivity of the circuit is progressively propagated up the hierarchy, starting from the connectivity information provided by the leaf cells. Hand in hand with this goes the process of 'pin reduction'. It is this pin reduction process that forms the essential bridge between the geometrical representation supplied by the graphical editors and the electrical representation that needs to be passed to a simulator.

Basically, pin reduction involves the removal of electrically duplicated pins from the 'argument lists' of the circuit macros which form a hierarchical simulation input file. Here, the inverter of Appendix I is used as an example.

Geometrically, the external appearance of the inverter is:

```
\begin{center}
| \hspace{1cm} \hspace{1cm} | \hspace{1cm} \hspace{1cm} | \hspace{1cm} \hspace{1cm} |
| vdd X \hspace{1cm} | X vdd |
| in X \hspace{1cm} | invert \hspace{1cm} | X out |
| vss X \hspace{1cm} | X vss |
\end{center}
```

Figure 3 External appearance of inverter leaf cell

This could give rise to a macro representation with an argument list of the form:

\[
\text{invert (vss, out, vdd, vss, in, vdd)}
\]

where the pins have been presented in the standard form - (north), (south), east, west, (left to right), bottom to top.

Most forms of software will abhor this representation, requiring deletion of the duplicated pins (at least if they refer to the same electrical node). An acceptable, pin-reduced argument list would be:
invert (vss, out, vdd, in)

These are the bare bones of the concept of pin reduction. Presented in a simple case, as it is here, it requires little in the way of explanation. In practice its implementation is considerably complicated by the problems of missing (ie disconnected) pins at module boundaries. In such cases, the pin reduction is generally not achieved by retention of the first occurrence of a node in the argument list and rejecting all other occurrences, but requires more care.

6.3 Name Checking

The name checker provides an aid to the detection of unintentional mis-connections at block and module boundaries. For each node in the circuit to be simulated, the name checker gathers all the names locally associated with that node, and makes an entry in the message file if more than one distinct name is detected. The messages are normally assigned a warning status, but are promoted to error status if a reserved name is involved.

6.4 Language-dependent Processing

The final phase of the whole translation chain is executed by a specific language module. The two modules currently in use offer translation to SPICE and HITEST. Apart from the facility to provide an control file to the SPICE module, the process is completely automatic. The sub-sections below purely mention the facilities provided in addition to the generation of the basic simulation file.

6.4.1 The HITEST Module

The HITEST module has the following features:

* name checking versus HITEST reserved names
* generation of skeleton .cwl and .vnm files if they do not exist
* representation of interconnections as tri-state wires with a long hold delay to model charge storage effects
* generation of 'wired-ors' to enable the association of multiple pin (and probe) names with single nodes

Appendix IV shows the HITEST input file generated as a result of the translation of Appendix III's simple
hierarchical circuit.

6.4.2 The SPICE Module

The SPICE module has the following features:

* incorporation and translation of a user supplied control file that refers to nodes by name

* a pre-processor stage that accepts a .REDUCEW pseudo control card to reduce transistors' widths by a specified amount, and .RANGE pseudo control cards to enable different transistor models to be applied for different ranges of transistor dimensions

* the propagation of internal nodes to the top level of the circuit.

Appendix V shows the SPICE input file generated as a result of the translation of Appendix III's simple hierarchical circuit.

7. Automatic Control of Translation

The need to remember which programs have to be run after an extended editing session in order to ensure that the simulator input file is brought up to date places an unnecessary burden on the user. An 'auto-translate' facility is therefore supplied to bind together all the programs in the translation suite. By this means translation to any supported language can be achieved with a single command.

Single programs within the suite can still be independently run to advantage. Invocation of the structure extractor as a structure consistency checker, for example, is a useful facility.

8. Conclusions

It has proved possible to automate all the main stages in the transformation of graphical leaf cell and floor plan data into simulator input files. From the users' point of view, all these stages are integrated into a single operation. The ability to deal with such disparate simulators as SPICE and HITEST indicates that the provision of translation facilities to further hierarchical simulation languages should not be difficult.
Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme. My thanks go to G R McMullan for all his effort in improving, extending, and implementing the capacitance extractor algorithm.

References

APPENDIX I - AN EXAMPLE OF LEAF CELL CIRCUIT EXTRACTION

Consider the simple CMOS inverter below.

```
vdd ++++++++++++++++++ vdd
     |
     |
in ------?------ out
     |
     |
-------------
```

For this circuit the circuit extractor would produce the following file (comments and the comment symbol '!' do not appear in the actual file):

```
circuit extractor 1.0
502387479 invert
502387465
0 0 0 0 0
u 1 metall vss
u 3 poly out
u 2 metall vdd
u 1 metall vss
u 4 poly in
u 2 metall vdd
1 1
4 ip
```

The transistor flags indicate if the transistor is unidirectional, resistive or has a dominant node. The transistors in this example have none of these properties.
APPENDIX II - AN EXAMPLE OF LEAF CELL CAPACITANCE EXTRACTION

For the circuit of Appendix I the capacitance extractor could produce a file similar in format to the following, depending on the process rules in force (comments and the comment symbol '!') do not appear in the actual file):

capacitance extractor 1.0
invert
502388501
502388805
2.50 3.50 14.88 10.30 14.88 10.30
2.50 3.50 14.88 12.00 14.88 12.00
1 0 0.61
2 0 0.61
3 0 1.40
4 0 4.32

Note:

1) the transistor order is the same as in the circuit extractor output file

2) the transistor dimensions are -
   width, length, source area, perimeter, drain area, perimeter
   (source == left/lower node, drain == right/upper node)

3) linear capacitances are expressed in the form -
   upper-node lower-node value
   (node 0 == substrate)
APPENDIX III - AN EXAMPLE OF STRUCTURE EXTRACTION

Consider a floor plan called 'chain' that simply places two of the inverter stages of appendix I in tandem:

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>vdd</td>
<td>X</td>
<td>---</td>
<td>X</td>
<td>---</td>
<td>X</td>
</tr>
<tr>
<td>in</td>
<td>X</td>
<td>invert</td>
<td>X</td>
<td>---</td>
<td>X</td>
</tr>
<tr>
<td>vss</td>
<td>X</td>
<td></td>
<td>X</td>
<td>---</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Figure A3 A simple floor plan using the CMOS inverter

Within the floor plan editor, this effect could be achieved in several ways, either by placing a 2x1 array of the leaf cell, or by employing two separate blocks, each placing one instance of the leaf-cell. For this example an arrayed placement has been used.

For this circuit the structure extractor would produce the following file (comments and the comment symbol '!' do not appear in the actual file):

```
structure extractor 1.0
502390681
chain
1
502390642 invert
chain
1
1
502390642 invert
chain
1
1
'in' = places leaf cell
1
2
6
5
6
vss 1
out 2
vdd 3
vss 4
in 5
vdd 6
invert 1 0
6
7 8 9 4 5 6
invert 2 0
6
1 2 3 7 8 9
```

5.107
APPENDIX IV - AN EXAMPLE OF HITEST OUTPUT

The hierarchical circuit of Appendix III generates the following HITEST output:

```
** HITEST file generated by translate version 1.2
** ----------------------------------------
circuit cmos INVERT_1 (VSS, OUT, VDD, PIN_1_);
** wire
   IN is PIN_1_
   IP is PIN_1_
;
** tri (10000, 10000) OUT PIN_1_
** moveif1
   gate1 (OUT, VSS, PIN_1_)
;
moveif0
   gate2 (OUT, VDD, PIN_1_)
;
** endcircuit
** ----------------------------------------
circuit cmos CHAIN (VSS, OUT, VDD, IN);
** INVERT_1 INVERT_1 (VSS, CHAIN_8_, VDD, IN);
** INVERT_1 INVERT_2 (VSS, OUT, VDD, CHAIN_8_);
** supply1 VDD;
supply0 VSS;
** endcircuit
```
APPENDIX V - AN EXAMPLE OF SPICE OUTPUT

Driving the SPICE translator with the following control file:

```
in pulse (0 5 10ns 10ns 10ns 10ns 20ns)
out in
.tran 2ns 100ns
```

produces the following SPICE file for the hierarchical circuit of Appendix III:

```
** SPICE file generated by translate version 1.2
* *-----------------------------------------------------------------------*
* * transistor models go here
* *-----------------------------------------------------------------------*
* M1 3 4 1 0 NTYPE W=2.50U L=3.50U AD=14.9P AS=14.9P PD=10.3U PS=10.3
* M2 3 4 2 5 PTYPE W=2.50U L=3.50U AD=14.9P AS=14.9P PD=12.0U PS=12.0
* VDD 5 0 5V
* C1 1 0 0.61FF
C2 2 0 0.61FF
C3 3 0 1.40FF
C4 4 0 4.32FF
*
* *-----------------------------------------------------------------------*
* X1 __ 1 8 3 5 _INVERT
X2 __ 2 1 2 3 8 _INVERT
* VDD_ 3 3 0 5V;
VSS_ 1 1 0 0V;
* VIN 5 0
+ PULSE (0 5 10NS 10NS 10NS 10NS 20NS)
* translation ...
*.PLOT TRAN OUT IN
* ```
THE MASK DATA GENERATOR

P R Benyon and R T Scarfe
British Telecom Research Laboratories

ABSTRACT

This document describes the mask data generation program used in the ASTRA VLSI design system. The program creates mask data in a number of languages using the output of the chip assembler and the cell spacer.

1. Introduction

The purpose of the mask generator is to combine the layout floor plan created by the chip assembler [1] with data for the minimum sized cells as created by the cell spacer [2] to produce mask data in a number of languages. An important requirement of this program is that it should be independent of technology.

![Diagram of the mask data generator process]

Figure 1 The function of the mask data generator

The mask data generator forms part of the ASTRA VLSI design system [3]. This document describes the program in terms of the operations performed and the mask data output produced.

2. Methodology

When the ASTRA system was first conceived, it was thought that the task of mask generation could be combined with that of chip assembly. However, the two tasks have now been

This paper was produced as part of Task VI of the ICT project which has received partial support by the Commission of the EEC under the MR-09 contract.
separated, since this approach has two advantages:

1. A user is not forced to generate mask data for a chip assembly in cases where the assembly results are unacceptable and where the design can be corrected without reference to mask data.

2. The same assembly results can be used to create mask data in more than one language.

3. Input Considerations

This section will describe the three basic types of input data required by the program.

3.1 Layout Floor Plan Data

The chip assembler combines a hierarchical floor plan with the minimum dimensions, connecting pin positions and edge material data for the placed cells. A layout floor plan is created where all dimensions are real and all lowest level blocks place cells which may have been stretched to fit their environment. This data is written in LIBRA format [4] which specifies the size and pin positions for each block, together with the positions of any sub-blocks placed.

The floor plan is such that sub-blocks completely fill the block in which they are placed unless the placed block is to be routed. In the case of the latter, one block is placed completely within the other and an offset coordinate is specified to allow space for the necessary routing channels.

3.2 Cell Data

The cell spacer takes a symbolic cell description and determines where the mask rectangles should be placed assuming cells in their most compact forms. The symbolic information is stored in the cell editor data file, and the grid line positions are stored in a spaced data file. The mask generator requires both files for each of the cells used in the floor plan.

3.3 Design Rule and Technology Data

The mask generator reads the same design rule and technology file used by the chip assembler and the cell spacer. It uses the information to determine how close any routing tracks may be placed, to specify which mask layers should be produced for each cell element and to obtain parameters for
merging. The file also contains data to allow oversize masks to be generated. The use of this file allows the program to be completely design rule and technology independent.

4. Process Considerations

4.1 Cell Stretching

The layout floor plan defines the new positions of the pins and sides of cells after stretching has taken place. Any necessary additional spacing of the minimally spaced leaf cell will be imposed at the stretch lines [3] of each cell instance.

4.2 Routing Rigid Blocks and Rigid Cells

The mask data generator creates routing tracks to connect everything that is placed rigidly. A 'rigid' placement is one where the block or cell placed is of fixed size. The routines used by the mask data generator to generate the routing tracks are very similar to those that the chip assembler uses to determine the size of routing channels.

Routing can take place on as many levels as desired provided that the design rules are such that levels do not interfere with each other. This is because the program routes each level independently and therefore the rules are only applied between tracks that are on the same routing level.

In CMOS design this usually means that polysilicon tracks have to be routed on the same level as diffusion tracks. The number of levels and their characteristics is determined by the design rule and technology file.

4.3 Merging

All mask data is merged together by the mask data generator on a cell-by-cell basis. Overlaps can occur but only where blocks overlap at the boundaries. The removal of overlaps within blocks usually reduces the amount of mask data generated and improves the speed of post processing and plotting software.

5. Output Considerations

Currently, the program will produce mask data in one of two formats, MGD or PHL. MGD is used internally within British Telecom, the definition of MGD is included as Appendix 1.
PHL is a language used mainly as an intermediate form in the creation of CIF [5].

5.1 Mask Data Constructs

The mask data constructs used by the program are restricted to rectangles and mask data block placements. No polygons or other complex shapes are used and the rectangles are all strictly orthogonal. This approach means that the program can be modified to generate files in new language formats with relative ease.

5.2 File Structures

The mask generator creates a separate file of mask data for each block. The name of the file is derived directly from the name of the block.

6. Conclusions

The mask data generator is used to generate design rule correct layout in a number of languages. The separation of mask data generation from chip assembly can lead to a significant reduction in design time.

Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme. The authors would like to thank the other members of the ASTRA development team for their advice and encouragement.

References


Appendix 1. Informal Definition of MGD (Mask Geometry Data).

MGD data consists of a number of very simple text constructs which place rectangles on masks. Each statement consists of a series of keywords, strings and integers terminated by a semicolon.

The basic statements are as follows:

**MASK number;**

This specifies that rectangles following this statement are to be placed on the mask denoted by the number. The specification remains in effect until another MASK statement or an ENDGROUP statement is encountered.

**RECT xorigin,yorigin;width,height;**

This specifies that a rectangle is to be placed with origin at (xorigin,yorigin) and with dimensions width,height.

**NEWGROUP name;**

This statement indicates the start of a group definition. An MGD group consists of a collection of rectangles on masks and may also contain calls to other groups (see GROUP). Note that a group definition may not contain the definition of another group, only a call to another group.

**ENDGROUP;**

This statement indicates the end of a group definition.

**GROUP name,xorigin,yorigin,XREP,xrpt,xspa,YREP,yrpt,yspa;**

This statement allows an instance of a group to be placed within a group definition. The origin values specify where the origin of the group is to be placed relative to the origin of the placing group. The repeat and spacing values and the associated keywords XREP and YREP are optional and allow the group to be repeated in either or both dimensions. The number of repeats is given by xrpt and yrpt, and the distance between the group origins of adjacent repeats is given by xspa and yspa.

**FINISH;**

This statement indicates that no more MGD is to be found in a file.
COMMENT string;
This statement allows MGD to be commented.

Certain extra information has been found useful for plotting the MGD data from ASTRA and is generated by the MGD output software in the form of COMMENTS. The syntax of these comments is given below.

COMMENT SCALE number;
This specifies the scaling factor used to convert user units to MGD units.

COMMENT BOUND xbound, ybound;
This statement specifies an approximate bounding rectangle that may be used to set the screen viewing window for plotting purposes.

COMMENT PIN NAME xcoord, ycoord, name;
This specifies that an external pin called name is to be found at the coordinate (xcoord, ycoord).
THE LOGICAL NAMING STRATEGY

R T Scarfe
British Telecom Research Laboratories

ABSTRACT

This paper describes a method of storing and retrieving randomly linked 'C' structures in files. Data allocated dynamically by an application program can be saved in a compact form, and, if required, transported in this form between machines. A set of routines allows this facility to be used easily by the applications programmer.

1. Introduction

The 'C' programming language provides facilities for dynamic storage allocation and data structure linkage. However, there are no facilities to permit the storage and retrieval of such linked structures in files. The logical naming strategy performs dynamic storage allocation. It produces compact data files that may be transported between computers and retrieved, whilst maintaining the inherent data associations.

2. The Standard Strategy for Data Linkage and Storage

The 'C' programming language permits structures to be defined as a set of predefined types and pointers to similar structures. Thus almost any data associations may be achieved. The task of writing and reading the data is left to the programmer. To reduce storage and maintain associativity each structure must be stored once, and once only. With some linkage strategies this may be very difficult without at least an extra flag in the structures to signal if the structure has been stored. Upon retrieval all associativity must be reasserted—a daunting and time consuming, if not impossible, task without some sort of logical naming strategy.

This paper was produced as part of Task VI of the ICD project which was received partial support by the Commission of the EEC under the MR-09 contra
3. The Logical Naming Strategy

All data structures are defined and introduced to the logical naming routines. Code is generated to handle each structure type. This code will enable the structures to be allocated, stored, retrieved and formatted for manual validation.

The code for allocating, storing and retrieving the structures must be linked with the logical naming functions and the applications program.

3.1 Storage Allocation and Data Access

The logical naming functions allocate storage space and assign a logical name for each data structure as the space is requested. Subsequent associativity and access to the structure is via that logical name. Given the logical name the pointer to the structure may be obtained from the logical naming functions. Thus all data structures must contain space for logical names to define data linkage. This may either be in addition to, or as a replacement for, the standard 'C' pointers.

3.2 Data is Output Once

The logical naming functions maintain a list of active logical names. Hence the applications programmer is relieved of the need to pass each of the structures individually to some storage routine. All the current logically named structures are output.

3.3 Data Associativity is Maintained on Retrieval

As the data is retrieved space is dynamically allocated and the mapping of logical name to structure is re-established. Once all the data has been read the data associativity will be complete. The structures contain linkage via logical names and these are been mapped to the relevant structures.

3.4 Compact Data

The volume of data stored may be less than that allocated at run-time. All pointers that are asserted during a program's active phase are redundant during storage, upon retrieval they must be re-asserted by the application, thus no data is stored for pointers. When the data is retrieved all pointers in the structures will be set to the null pointer.
3.5 Portability

Only standard 'C' types (integers, characters etc.) and logical names need to be stored. Hence the data may be stored using standard ascii output. Thus the files so generated may be transported between computers.

N.B. If the files had been a simple binary dump of each structure portability would have been very difficult, different computers may have different type sizes and even a different byte order.

3.6 Data Format

Additional functions are generated automatically that will format a data file to display each of the structures. Thus although the raw file may not be man readable, a form can be created to assist the file to be viewed.

4. Conclusions

Many functions have been written using the logical names. Data has been successfully transmitted between machines and functions. An interactive editor has been written that relies entirely upon logical names for its data associativity. Even with the additional task of translating from logical name to address, to access the data, no degradation of speed has been noticed by the users.

Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the Commission of the EEC under the 3744/81 Microelectronics programme. The author would like to thank the other members of the ASTRA development team for their advice and encouragement.
The STAR (Signal Transform ARchitecture) is a signal processing chip based upon a serial architecture which can be simply and dynamically configured to perform a wide range of conventional signal processing functions. The architecture consists of an array of identical Elemental Signal Processors (ESPs) with interconnections between them being set up by a simple instruction. The resulting chip design is modular, easy to integrate and is capable of being extended as improved fabrication techniques and finer geometries allow higher levels of integration. It thus represents a true VLSI solution to the problem of digital filtering of speech-band signals.

The STAR has been designed using VLSI design software developed by British Telecom Research Laboratories, and implemented in 2.5 micron CMOS technology. The device features a novel memory structure, with the serial multipliers, adders and control logic utilising a dynamic four-phase precharge-discharge clocking scheme.

1. Introduction

In speech-band communications digital filters are used in a variety of applications. Finite impulse response (FIR) filters are used in, for example, adaptive equalisation and echo cancelling. Infinite impulse response (IIR) or recursive filters are used as high Q filters in applications such as speech synthesis and multi-frequency signalling.

The STAR (Signal Transform ARchitecture) is a serial signal processing chip operating at a clock rate substantially above speech-band sampling rates. It is based upon a modular architecture and can be configured to perform a wide range of conventional signal processing operations on speech-band data. Its design arose out of the study of a number of dedicated integrated digital filters designed for telecommunications applications.

The approach taken was to examine the digital signal processing (DSP) functions to be performed and, by identifying as much commonality as possible, to produce an
architecture which is capable of performing all these functions. A requirement of the architecture was that it should be dynamically reconfigurable. This allows it to be multiplexed across a number of different tasks within a single system.

The resulting architecture is much more general purpose than the dedicated circuits it is designed to replace, but has none of the programming overheads associated with microprocessor-like DSP chips. Because the architecture is modular, it is extendable as improved technology allows greater levels of integration. It thus represents a true VLSI solution to the problem of digital filtering of speech-band signals.

2. Architecture

To establish a useful building block for general signal processing (SP) applications we first examine the basic signal processing operations to be performed. These can be specified by a small number of equations eg:-

(a) Conventional digital filtering:

\[
y_k = \sum_{n=0}^{N} a_n x_{n-k} + \sum_{n=1}^{K} b_n y_{n-k}
\]

Where \(y\) is the output data sequence, \(x\) is the input data sequence and the coefficients \(a\) and \(b\) define the characteristics of the filter.

(b) Modulation:

\[
y_k = m_k x_k
\]

Where \(m\) is the modulation sequence.

(c) Image processing and transforms:

These are of the form:

\[
y_{nm} = \sum_{j=0}^{N} \sum_{k=0}^{M} a_{jk} x_{n-j,m-k}
\]

We note that any sum \(S = \sum_{i=0}^{N} a_i x_i\) can be calculated iteratively using the following scheme:

\[
S = S_{i-1} + a_i x_i
\]
The circuit of figure 1 implements this calculation and so can be used to construct a processor capable of handling all the above operations. To clearly identify the intended function we have called it an "Elemental Signal Processor" (ESP). It consists of a multiplier, an adder and a programmable memory or delay of an integral number of sampling periods (nT).

\[ S_i - 1 + S_i \text{ memory} \]

Figure 1. Elemental Signal Processor

The ESP circuit is similar to the "Inner Product Step Processor" described by H T Kung and C E Leiserson [1]. S-Y Kung [2] has demonstrated how such elements may be connected via suitable delays in the communicating paths to form a one-dimensional array of processors featuring only local intercommunication. Such an array is capable of performing a range of signal processing tasks such as correlation, convolution and IIR filtering. However, this array suffers from two drawbacks which are of interest to us. Firstly only alternate processors are operational at any one time as a result of the delays inserted in both the input and output signal paths, and secondly high order IIR filters realised in direct form as a simple chain of processors require long word lengths to obtain satisfactory filter characteristics.

To overcome these drawbacks we have taken a more pragmatic approach to the interconnection of our ESPs. The removal of delays in the input signal path means that each processor is in operation every system cycle. This means that communication is no longer strictly local, but as we are considering a serial architecture this does not constitute the bus routing and driving problem that it would do in the parallel machine that Kung discusses.

In order to overcome the requirement for long wordlengths in high order IIR filters we have grouped our ESP array into groups of four, each group having its own associated peripheral circuit. This basic building block of the array we have called a "second order block". Each block can adopt a number of configurations including that of a second order...
IIR section. It is well known that to achieve stable and accurate filtering characteristics without resorting to very long word lengths, the most efficient method of implementing higher order IIR filters is as a cascade of second order sections. This can therefore be achieved either by cascading second order blocks, or for data rates less than the throughput rate of the block by multiplexing the block across the task.

A single second order block is the circuit we chose to integrate as the STAR. It consists of a linear array of four ESPs, each with an associated selector to set up nearest neighbour connections, plus a small amount of I/O circuitry, including an input selector, an input scaler and two adders. It can be configured as an N pole M zero IIR filter, where \( N+M=4 \), as a four point transversal filter, or as a time delay and integrate (TDI) circuit. In addition the array can be operated as four independent ESPs which are accessed in parallel and which can be used to calculate up-dated tap coefficients in adaptive filtering applications, operated as modulators or externally interconnected to construct a transform processor or lattice filter sections. The principle is extendable in that sophisticated selectors which set up connections between ESPs which are not adjacent would allow a greater number of configurations to be set up within the array. A further possible extension is to two dimensional arrays of ESPs, for applications such as image processing and speech recognition.

Typical lengths for FIR filters are 48, 64 and 128 points, however, IIR filters of only 2nd or 4th order are often used. If these are implemented directly they require very different numbers of ESPs. When they are to be implemented using a fixed number of ESPs the problem is overcome by providing each ESP with a delay/memory of programmable length and recycling the output of the ESPs back to the input to build up higher order circuits. This has been done in the STAR by incorporating an on-chip feedback path. The programmable delay associated with each ESP is set to \( nT \), \( n \) being the number of times that the output is recycled. The delay or latency between a data sample being input and a corresponding (intermediate) result being output is the operating period, \( T \). Such a scheme relies upon the operating cycle of the STAR being much shorter than the input sampling period.

The fast operating rate of the STAR can also be exploited by multiplexing it across a number of lower sample rate channels, or operating it in a combination of the recycling and multiplexing modes. Also, because it is dynamically reconfigurable (i.e. its configuration can be changed every operating cycle without data corruption or loss) it can be multiplexed over a range of different signal processing functions. In this context, more sophisticated processors can be built up by the inclusion of a push-down stack [3] in the feedback path. This allows the effective implementation of branching networks or trees of signal processors. It can be shown that any tree of filters can be calculated using a
stack to store intermediate (branching point) values provided that the algorithm of always going to the highest remaining branch of the tree is followed. The feedback path and push-down stack are incorporated in the peripheral circuit associated with each second order block.

3. Design of the STAR

The STAR is a serial signal processing chip operating at 10MHz with a 32 bit operating cycle. 2s complement LSB first format 16 bit data and coefficient words are used, with a 24 bit internal word length to allow for bit growth as results accumulate, giving stability and accuracy at least as good as current dedicated FIR and IIR filters (Adams et al., [4]).

The choice of a serial architecture was made for a number of reasons. Apart from the fact that serial I/O is common in many speech-band systems (e.g. telephone circuits) serial circuits have the same area-time products as parallel circuits, if not better. In addition they have lower communication requirements with local communication predominating and they therefore do not suffer from large on-chip interconnection costs or pin-out limitations. Most Signal Processors fall into a very simple class of machines in which control operations cycle through a fixed sequence that is almost completely independent of the result of an operation. For such machines the same performance can be achieved in the same silicon area for serial implementations as for parallel implementations by duplicating serial circuits and performing calculations bit-serial but word-parallel, and by the use of pipelining. By going for a serial approach we can get high performance without the control and operation sequencing overheads associated with a more general purpose parallel processor.

Although serial architectures have been brought to prominence by Lyon [5] and Denyer et al. [6], who have established methodologies based upon this approach, the attractions of serial architectures have made them common in signal processing and other telecoms applications for a number of years.

3.1 The Multiplier

The multiplier used in the STAR is a simple 16 bit parallel-serial circuit in which the coefficient is clocked in on the operating cycle before it is required, and latched into the multiplier at the start of multiplication. The output is rounded back to prevent bit growth problems in recursive configurations of the ESPs. This form of multiplier suffers from the drawback that the 32 clock cycles needed to produce a result limit the throughput rate. This leads us to select an operating period of 32 clock cycles in order to make the basic delay or latency (T) of each ESP equal to the throughput rate, a feature that is
essential to maintain the reconfigurability of the STAR.

Although Lyon [7] has presented a multiplier that has a throughput of one rounded 15 bit product every 16 clock cycles, his multiplier suffers from a high latency (delay between a two values being input and their product being output) which makes it impossible to exploit its high performance and achieve a throughput rate equal to the delay for each ESP. Also a study (Myers, Ivey [8]) showed that the area-time product of the Lyons multiplier was no better than for the simple form of 2s complement multiplier that we have used.

3.2 The Memory/Delay

The 10 MHz operating rate of the STAR allows it to be multiplexed over up to 32 channels sampled at 8 KHz (single telephone channel). In order to fully exploit this each ESP must be provided with a programmable wordlength memory of 31 24 bit words, giving a total on-chip memory requirement of 3 Kbits. To provide this as shift-register is very wasteful of silicon area whilst the use of 3-transistor RAM suffers from addressing problems. To overcome this a novel memory structure has been devised (Ivey, [9]) known as a SAM (Sequential Access Memory). It is based upon a 3 transistor RAM in which the Read and Write lines have been replaced by a single Column Select line, and addressing is performed by two pointer shift registers. The memory length is programmed by presetting the pointer position of one of the shift registers. SAM is as simple to use as conventional shift register, whilst having similar silicon usage to RAM. Figure 2 shows a simple SAM of fixed length.

The memory in each ESP allows us to operate the STAR as, for example, a 128 point Transversal filter, 32 2nd Order IIR filters or 16 4th Order filters operating on data sampled at 8 KHz. The trade off between number of filters, their order and the input sampling rate is obvious.

![Figure 2. Sequential Access Memory](image-url)
3.3 The Push-Down Stack

From a computational point of view, a binary tree of processors requires the storage of the greatest number of intermediate results. The 32 words of internal memory in each ESP allow us to program the STAR as a tree of up to 32 branches, giving us a five level binary tree as a worst case. This requires the storage of 5 intermediate values, so a 5 word deep push-down stack has been incorporated in the feedback path of the STAR.

In keeping with the format of the rest of the chip, the stack has been implemented in serial form as five 32 bit shift register "snakes" which can bite their own tails or the tails of the snakes immediately above or below them, depending on the stack instruction. Figure 3 illustrates the arrangement of the Stack.

Figure 3. Serial Push-down Stack

3.4 The Configuration Circuitry

The ESPs are configured by a 2:1 selector at each data input and two small PLAs at each ESP output. The 2:1 selector simply determines whether the ESPs are accessed in parallel or from a common data input line. The output configuration circuitry consists of a Translater PLA which takes the global configuration instruction and translates it into specific instructions for the ESP concerned. The output of the Translater PLA is fed into a Router PLA which sets up the required interconnections.
3.5 Timing and Control

The timing of the STAR is such that data in each 32 bit processing slot is entirely isolated, and that control signals relating to that slot arrive only at the moment that they are required. In each configuration of the STAR delays round internal loops are multiples 32 clock cycles, ensuring that data values passing through each ESP join each other in phase. The STAR can therefore be dynamically reconfigured (i.e. it has the capability of being reconfigured every processing cycle). This means that in a complex system a single STAR can be used to perform a number of different filtering or signal processing functions, acting for example simultaneously as both a 64 point transversal filter and a bank of 8 4th order IIR filters.

The STAR is controlled by means of a 32 bit instruction word, divided into conveniently small fields to make it easy to program. These are four 5 bit fields to independently set the memory lengths from T to 32T, a 4 bit field to set the STAR configuration, a 4 bit input scaler field and 4 bits to control I/O and the push-down stack. Figure 4 gives details of the instruction word, whilst figure 5 shows the STAR configured as (a) a Second Order Filter Section (b) an FIR filter.

Figure 4. The Instruction Word

Figure 5. (a) STAR configured as a biquad filter (b) STAR configured as a 4 point FIR filter
4. Implementation

The STAR has been designed using the ASTRA VLSI design software developed by British Telecom [10]. Figure 6 shows the chip floor plan at an intermediate point in the hierarchy. The chip has been designed for a bulk CMOS process (the design software allows the process choice to be deferred). The design style adopted for the sequential logic circuitry is a form of clocked Domino Logic requiring 4-phase clocks (Myers, Ivey [11]), which are generated on-chip. Figure 7 shows a transistor level diagram of an adder designed in this style together with the required clock waveforms. A photomicrograph of the chip is shown as a colour plate. The chip contains approximately 25,000 transistors.

![Figure 6. Floorplan of STAR chip](image)

![Figure 7. Four-phase adder circuit](image)

5. Conclusions

A structured VLSI serial signal processing architecture has been presented, along with an implementation which provides a solution to a range of signal processing problems, and is easy to program and operate.
Acknowledgements

The work described in this document formed part of Task VI of project MM09, which was supported by the commission of the EEC under the 3744/81 Microelectronics programme.

References


Chapter 6
Functional Design
Definition of the syntax and semantics of the Modelling and Design Language, MoDL.


This report consists of six parts with the following content:

PART I: Semantic definition of the Modelling and Design Language MoDL,

PART II: The sequential MoDL programming language, which is used to describe the behaviour of functions in a hardware description,

PART III: The parallel MoDL hardware description language,

PART IV: The MoDL to C compiler,

PART V: An alphabetical cross reference list,

PART VI: MoDL case study: A single chip digital signal processor.

This paper was produced as part of Task VIII of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
1.1 General Introduction

This report starts with a rather short introduction about the main features of MoDL. These points are all related in more detail in the remaining chapters. However, we felt that a short introduction in which all essential aspects of MoDL are shortly mentioned might be useful for the expert in the field. Those who find this introduction too short are encouraged to read the remaining material first and to revert to this introduction afterwards.

Scope of the MoDL language. The principal target of the Modelling and Design Language, abbreviated in the remainder of this report to MoDL, is high level design of digital circuitry. Certain time discrete analog designs such as CCD designs could benefit also from a design system like MoDL. However, this report will concentrate on the design of digital circuitry. MoDL is capable of describing design entities at one or more levels of abstraction, or at mixed levels of abstraction. To that end it is capable to capture elements from the gate level to the system level.

The design model. A design is described in MoDL in terms of functions, interconnect (between the functions) and information about relative placement. The MoDL user can define models which may contain a predefined composition of the above mentioned fundamental semantic units and references to nested models. We were very eager to abandon arbitrary restrictions on the composition of models, so models may be used to compose functional conglomerates, to describe interconnect or to describe layout in conjunction with our abutment algebra or any mixture of the above. The design discipline covered by the current MoDL is synchronous design. This has two major advantages:

1. all tolerance sensitive timing aspects are isolated from the high level aspects of the design, and
2. many design aspects are provable for synchronous designs.

The MoDL timing analysis tools can be used to map such a design in an optimal way to a certain technology and the favorite clocking scheme in that technology, such as a multiphase clocking scheme in a MOS realisation. Similar provable techniques for asynchronous designs are under development.

Modularity. Each function, model or abutment formula stands on its own and can be completely understood from its MoDL description, either directly or in terms of the models being referenced. Behavioral descriptions of (partial) designs or
functions can be written at any desired level with the MoDL programming language, which executes functions in a sequential manner much like in Pascal or C in the same environment as the MoDL hardware description system. This aspect of an embedded environment makes it possible to check descriptions at several levels against each other.

**Hierarchy.** A design can be (de)composed in an arbitrary way, using functions, models and abutment formulas as primitives. Whether the composition was done in a top-down, bottom-up or hybrid form of these two, cannot be distinguished when the design is finished. This aspect becomes more clear during the iterative refinement of the given design. Alternate designs can be used in a judicious way, for instance as part of the iterative refinement process, or to tailor the complexity of the design instance to the computational resources available, given the formal equivalence of the designs.

**Validation.** A simple but rather effective way to obtain a validated design is by automatic simplification or automatic construction, based on user specified rules. Complicated designs can be derived from a rather compact rule set by induction. Alternate designs can be checked either by studying the behaviour of the system in which they are substituted or by studying their behavioral equivalence. Another validation aspect is related to algebraic soundness. It is in this respect well known that no two function-outputs can be connected to the same wire in a well defined design. This aspect is frequently not validated in many design systems, just because these design systems do not attempt to logically interconnect physically connected wires. There are few other methods left than simulation in such environments to find the design error. MoDL uses its inherent symbolic processing capabilities to catch such errors directly in the design process, i.e. without simulation.

**Distributed control.** Functions can be disabled by the MoDL case ... endcase language construct. The corresponding function value is taken to be ineffective if the condition in the case is not met. This has several physical interpretations, of which a “high impedant state” of the corresponding output(s) is one. The conditions which control multiple function outputs connected to a single wire should be mutually exclusive. Software to (dis)prove this design constraint is currently under development. MoDL comes very close in its capabilities to the switch level, as it can be used to describe distributed functions with the pulldown and pullup language constructs, however no attempt is made to model switches which
can propagate signal values in two directions.

**Additivity.** The fact that MoDL implements the basic semantic axes of the problem domain in a purely orthogonal way makes it relatively easy to take physical constraints of a design into consideration. One need not extract the parasitic capacitances of a layout instance in order to calculate its effect on the timing of a design. It is instead sufficient to do so for a single design instance. This makes it attractive to collect such information only once and rely on the fact that MoDL will logically interconnect the corresponding wiring and update the corresponding database entries, irrespective of the way in which the models were interconnected, by composition or by abutment.

### 1.1.1 The Implementation of the Orthogonal Design Axes

**Motivation**

The successful realisation of an algorithm in VLSI cannot be performed without simultaneous evaluation of its specification. This however requires that the many views, which can be taken in the VLSI design discipline, should be freely representable in the VLSI design environment. Figure 1.1.1 shows the major relations in this respect.
Semantics of MoDL

VLIS = Algorithm & Specification

High level design language.

Evaluation tools.

MoDL

Syntax and Semantics.

Parallel language. Algorithmic specification.
Hierarchical composition.
Model. Function.
Algorithmic correctness.
Layout structure.
Simulation.
Timing.
Critical path(s).
Electrical effects.
Wire list.
Layout generation.

Specific Views.

Figure 1.1.1. The major MoDL aspects.

These aspects should be placed in the context of their design view. Many high level design systems just implement a single point in the design space. Synthesis programs frequently map from one point in the design space to another one. Figure 1.1.2 gives a summary of some existing synthesis systems, showing the semantic points which they bridge.

Level of Abstraction: System:

FUNCTIONAL

- Ultima: Finite state mach

ALGORITHMIC

- IBM PLEX, IBM Otten

INSTRUCTION SET

- CMU-DA

- MacPitts

RT-BEHAVIOUR

- Darringer & LOGE

- DSL (Karlsruhe)&

RT-STRUCTURE

- MODEL: Lattice Logic

LOGIC

- PLA

- Expresso &

- "Quine-

- Mc Cluskey"

GATE

SWITCH

SYMB. LAYOUT

- Magic

LAYOUT

Figure 1.1.2. Existing synthesis systems.
Some 50 to 100 synthesis systems could be defined and identified in this way to cover all possible pairs in the design space. It should be clear that such an enormous amount of VLSI design tools would be unmanageable and that some standardisation is highly needed. A major disadvantage of this approach is also that it is inherently impossible to take effects of other semantic axes into account during the synthesis process. The design problem is far better treated in a context in which all design aspects can be stated and evaluated all simultaneously. This requires a homogeneous design spaces, spanned in an orthogonal way by the implementation. Figure 1.1.3 shows the major design views now brought together on orthogonal axes.

The SPIRIT CAD system implements some specialized tasks as separate modules, just for reasons of efficiency and complexity. The orthogonal implementation of MoDL is chosen to span the whole design space as good as possible with highly flexible tools. This makes it possible to confine with a single specification for a given design.

Figure 1.1.3. Major design views.

The SPIRIT CAD system implements some specialized tasks as separate modules, just for reasons of efficiency and complexity. The orthogonal implementation of MoDL is chosen to span the whole design space as good as possible with highly flexible tools. This makes it possible to confine with a single specification for a given design.
It should be noted however, that it is not sufficient to implement an orthogonal design space in the datastructures which describe a design, if the language constructs used cannot express all possible designs. This remark is far from trivial and it has got special attention in the implementation of MoDL. The language constructs are given in figure 1.1.4.

*Primitives:*

1. `FUNCTION F (...)`, `ENDFUN;`
2. `MODEL M (...)`, `ENDMOD;`
3. `F (...), M (...),`
4. `CASE Cond1, Cond2, ENDCASE,`
5. `LOOP EXITIF Cond, ENDLOOP,`
6. `ALUReg := A + B,`
7. `ABUT Object, M1 - M2;`
8. `MASKGEN instance;`

*Examples:*

- `Not, And, Or, +, - ...`
- `Accumulator, ...`
- `Composition, ...`
- `dynamic communication`
- `clock circuitry`
- `clocked register`
- `layout`
- `technology`

*Figure 1.1.4. MoDL language constructs.*

The basic features of the orthogonality and extensibility of MoDL depend on:
1. Language design.
2. Compatible design views.
3. View transformation functions, like: `MODEL, ABUT, DEMAND, MASKGEN.`
4. A layered system implementation, built on 3.

The importance of those issues will not be mentioned explicitly in the rest of this report.
A MoDL description is composed of semantic units of five kinds:

1. **Behaviour**, described by function definitions expressed in the sequential MoDL programming language.

2. **Interconnect**, implicitly described in model expressions, explicitly described with labeled expressions (assignment statements) and in parameter lists of models.

3. **Layout** information concerning:
   - The relative ordering of wires on the sides of a model, expressed in the *parameter list* of the model, and
   - The relative placement of models, expressed in *abutment* formulas.

4. **Communication**, i.e., the relation with time: This issue has to do with the uniqueness of the execution of the algorithm and the propagation of results through the parallel computational net defined by the design.

5. **Technology and Implementation**. This last semantic entry concerns all relevant information needed for a particular technology and implementation. For instance, a wiring-list for a gate array has a format which differs from the one used for a 7400 series TTL design for use on a printed circuit board. Similar remarks can be made for the layout techniques possible in different technologies.
Figure 1.1.1. The layered MoDL semantic views.

Figure 1.1.1 gives a survey of the layered implementation of MoDL with respect to its semantic units.

The functions, models and abutment formulas used in a MoDL design span the basic semantic axes of the MoDL system, being behaviour (functional description), interconnect (wiring), layout (relative placement) and communication (the relation with time). It is a fundamental point in the design of the MoDL system that those major semantic axes are implemented in an orthogonal way. The conflict which arises when two function-outputs are connected to the same wire is easily detected in MoDL, as it represents interconnect as interconnect and communication as communication. Many other implementations represent for instance a feedthrough wire (interconnect) in a cell as a function (behaviour) with an infinitesimal small delay (communication). Such an implementation can be proven to be correct for a simulator, but new ideas have to be worked out if one wants to do a timing analysis in which capacitive effects of wiring should be included in the timing model. An orthogonal system implementation spans the design space for all possible tools, making the toolset additive. This unique feature makes MoDL highly flexible and user friendly.
1.2 Introduction to MoDL

A hardware description contains several semantic aspects, all modeled within MoDL with appropriate language constructs, among which:

1. A LISP programming language interface based on a high level syntax.
   For instance the value of N! is computed, for positive integers, with the following function:
   
   ```
   function Fac (N),
   case N < 2    {1},
   default {N*Fac(N-1)},
   endcase,
   endfun;
   ```
   
   This programming language interface is used to describe behaviour of hardware functions as well as the main programming tool for the implementation of the MoDL hardware description language. This programming interface is a sequential language derived from LISP.

2. Hardware description with language elements from the same syntax.
   For instance a high level description of a three-bit adder is:
   
   ```
   model ThreeBitAdder (A, B, Cin, S, Cout),
   Sum: A + B + Cin,
   S: mod(Sum, 8),
   Cout: quotient(Sum, 8),
   endmod;
   ```

3. Validation of design.
   A bit adder can be defined as well. For instance, the definition of the bit-adder below shows how the radix base can be changed, by writing mod(.,2) and quotient(.,2) instead of mod(.,8) and quotient(.,8).
   
   ```
   model BitAdder (Al, B1, Cin, Sl, Cout),
   Sum: Al + B1 + Cin,
   Sl: mod(Sum, 2),
   Cout: quotient(Sum, 2),
   endmod;
   ```

   This refined design can now easily checked against its high level
counterpart in the same MoDL environment.

4. **Hardware primitives.**

Models can be expressed in hardware primitives already supported by MoDL, like _xor, _or and _and.

```plaintext
model BitAdder (A1, B1, Cin, S1, Cout),
S1: _xor (_xor (A1, B1), Cin),
Cout: _or (_and (A1, Cin), _and (B1, Cin),
_endmod;
```

5. **Hierarchical description.**

Models can be interconnected to form new computational structures, for instance one can synthesise a three-bit adder from three bit-adders in the following way:

```plaintext
model ThreeBitAdder (Cin, A1, A2, A3,
B1, B2, B3,
S1, S2, S3, Cout),
BitAdder (A3, B3, Carry2, S3, Cout),
BitAdder (A2, B2, Carry1, S2, Carry2),
BitAdder (A1, B1, Cin, S1, Carry1),
_endmod;
```

The three-bit adder is a fully parallel computational network which calculates the sum-bits S1, S2, S3 and the outgoing carry, Cout.

Note that this latter description is actually a low level description of the three-bit adder.

6. **Reformulation of a design.**

MoDL implements a relatively large amount of automatic simplification rules. This set can however be extended by the user in a straightforward way. A rewrite rule for the function _xor of the form:

```plaintext
_xor (A, B) -> _or (_and (A, _not (B)),
_and (_not (A), B)),
```

is specified as:

```plaintext
model _xor (A, B),
_or (_and (A, _not (B)), _and (_not (A), B)),
_endmod;
```
The rule:

\[ \neg (\neg (A)) \rightarrow A \]

cannot be specified in the same way, as the action to be taken depends on the fact that the argument of the \( \neg \) operator is itself negated. A user can however use a modelfunction to obtain the needed conditional model-expansion:

```plaintext
model function \( \neg \) (Args),
local (A),
A: modelexp(first Args),
case first A = \'\neg\' {second A},
default \{\'\neg\(A)\}',
endcase,
endmod;
```

Additional user-defined rules can be specified to convert boolean expressions automatically into a canonical form. This can be used for several purposes:

1. To match either the needs of a specific technology or a specific implementation.
2. To show the isomorphism between different descriptions of a single design, a step in a hardware correctness proof.

The semantics of MoDL, when used as a programming language, differ quite drastically from those when used as a hardware description language. In the former application MoDL follows the semantics of LISP and acts as a sequential language. When used as a hardware description language, MoDL has the semantics of a parallel language, fully suited to cover the parallelism found in hardware.
I.3 Variables, Execution and Communication

The semantics of program execution is in MoDL much the same as in other programming languages. There is a notion of progress of computation which refers to the existence of an abstract "program counter", as it is found in such languages as PASCAL, ALGOL, LISP or FORTRAN.

Almost all syntactic language constructs used in the MoDL programming environment can also be used to describe hardware models, if we take some fundamental semantic differences into account.

Figure 1.3.1. shows three versions of a simple algorithm which can be executed in either mode. The algorithms are described in terms of variables, \((x, y, z, a, b)\), functions \(+\) and \(*\), with actual arguments as specified, an assignment operator \(:\) and a comma used as separator.

<table>
<thead>
<tr>
<th>version 1:</th>
<th>version 2:</th>
<th>version 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x + y,)</td>
<td>(b: a * z,)</td>
<td>(b: (x + y) * z,)</td>
</tr>
<tr>
<td>(b: a * z,)</td>
<td>(a: x + y,)</td>
<td></td>
</tr>
</tbody>
</table>

Calculated values for \(x: 3, y: 6, z: 2\) are:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: 9)</td>
<td>(b: 0)</td>
<td>(b: 18)</td>
</tr>
<tr>
<td>(b: 18)</td>
<td>(a: 9)</td>
<td></td>
</tr>
</tbody>
</table>

The result of the algorithm when executed in programming mode depends on the version selected. For instance version 2 of the algorithm calculates a value of 0 for \(b\), if all variables had an initial value of 0. This dependency on program progress is typical for all languages which support the idea of an abstract "program counter".
A parallel language assumes the simultaneous presence of all functions and operands referred in the expressions specified in the form of a network. The variables are used to describe the branches (wires) in the network. The functions produce results which are computed from their arguments and they communicate these results to other functions along the branches of the computational network. The notion of program progress found in a sequential algorithm has its counterpart in the communication aspect of a parallel algorithm, as communication expresses the relation with time in a parallel context.

Figure 1.3.2. shows a graphical representation of the algorithm which is given in several different representations in figure 1.3.1. Note that the three versions of the sequential algorithm, given in figure 1.3.1, all represent the same algorithm when the semantic rules of a parallel algorithm are applied to these descriptions, i.e. when the presented expressions are part of an expression list in a model body.

1.3.1 Syntax and Semantics

A <variable> is defined in our syntax diagrams with the rules:

The signs for arithmetic greater and arithmetic less are used throughout the text to reference terms defined in the syntax. No other symbols have a meta meaning, as we use, in this text, only a
single selection from possible syntactical constructs. Computer jargon is avoided in this report as much as possible. Commonly known terms, such as object list will be included in a glossary which will be included in a future version of this report.

We can now recapitulate the issues presented so far:

**Syntactical unit:**

```
<identifier>  element from object list
<variable identifier>  <identifier> with associated value
```

The associated value represents:

1. A program value, which depends on computational progress in a MoDL program.
2. A signal value, independent of computational progress, in a model body.
1.4 Functions

A function declaration has the syntax:

```
function <function identifier> <formal parameterlist>

endfun;
```

A function is referenced with the syntax:

```
<function identifier> <actual parameterlist>
```

either from within an expression or from within a model expression. The function is invoked during the execution of an algorithm. The executing function, `eval`, calculates the function body such that the values of the actual parameters are used wherever the corresponding formal parameters are specified.

With the absence of true parallel machines, MoDL executes all function references within a model body as invocations of functions. It will be clear that side effects, such as assignments to variables which result into (almost) unpredictable results in subsequent invocations of functions, cannot be tolerated.

MoDL can also handle functions with an unknown number of actual arguments. The syntax:

```
function <function identifier> <formal parameter>,

endfun;
```

describes a function which can be referenced with an arbitrary number of arguments. During execution, MoDL binds the list of all actual parameters to the single formal parameter.

**Example:**

A function `and`, which takes the logical and of all its arguments is easily defined as a function with an arbitrary number of actual arguments (see section II-4-2).

A large collection of predefined LISP-functions can be referenced directly from a model body (see section II-0). Additional functions are easily expressed with the function declaration syntax in terms of primitives already available. This makes precise descriptions of high and low level designs, as already shown in the introduction, all simply possible.
### Example:

| S: $mod(A + B + Cin, 8)$ | Descriptive level:  
|--------------------------|---------------------|
| $S_1: _xor(_xor(A_1, B_1), Cin)$ | $High$  

---

**Semantics of MoDL**  

**Functions**

---

**6.17**
I.5 Models

*Model declaration.* A model is an abstract representation of a computational (sub)network. Its formal parameters (also called terminals) define the way in which it can be embedded in a larger computational structure. The model declaration has the syntax:

\[
\text{model } \langle \text{model identifier} \rangle \ <\text{formal parameterlist}> \\
\langle \text{model body} \rangle, \\
\text{endmod};
\]

*Model composition.* The syntactical form:

\[
\langle \text{model identifier} \rangle \ <\text{actual parameterlist}>
\]

is the basic construct for the hierarchical composition and hence also the instantiation of complicated designs. This composition construct is only effective from within a model expressions.

*Top level interface.* A top-level model expression of the form:

\[
\langle \text{model expressionlist} \rangle ; \quad \text{or} \\
\langle \text{model expressionlist} \rangle \\
\]

is needed to compose a complete design.

*Locality and parameter passing.* Variable identifiers used within a model expression are local with respect to the model body in which they are defined. Variable identifiers which refer to formal arguments however, are subject to the parameter passing algorithm used for models:

1. A *<constant>* and a *<variable identifier>* are passed without any change, e.g:

<table>
<thead>
<tr>
<th>model declaration:</th>
<th>model reference:</th>
<th>instance:</th>
</tr>
</thead>
<tbody>
<tr>
<td>model Example1 (a, b), a + b, endmod;</td>
<td>Example1 (12, x)</td>
<td>12 + x</td>
</tr>
</tbody>
</table>

2. Syntactical constructs of the form:

\[
\langle \text{variable identifier} \rangle : \langle \text{expression} \rangle \\
\langle \text{variable identifier} \rangle := \langle \text{expression} \rangle
\]

are passed only once. Subsequent references are of the form

\[
\langle \text{variable identifier} \rangle
\]
Example:

<table>
<thead>
<tr>
<th>model declaration:</th>
<th>model reference:</th>
<th>instance:</th>
</tr>
</thead>
</table>
| model Example2 (a, b),
a + a + b, |
| Example2 (x: f(y), z) | (x: f(y))+x+z, |

This rule is of importance for those functions which do have a side effect, such as i/o statements, like a print statement. Multiple substitutions of x: f (y), do violate one of the semantic rules introduced later on. Multiple substitutions of f (y) are not only inefficient due to recalculation of the function f, such substitutions may cause undesired effects too. This undesired effect is demonstrated below:

Example, undesired situation:

<table>
<thead>
<tr>
<th>model declaration:</th>
<th>model reference:</th>
<th>undesired instance:</th>
</tr>
</thead>
</table>
| model Undesired(a, b),
a, b, |
| Undesired(printatom(0), printatom(7)), | [printatom(0), printatom(0), printatom(7)], |

The model reference suggests that 0 and 7 will be printed, however the undesired Model-expansion would result in a printout of 0 0 7. The implemented argument substitution rule gives as desired instance:

\[
[x: \text{printatom}(0), \text{z}, \text{printatom}(7)]
\]

which indeed gives a printout of 0 and 7 as expected from the model reference.

3. All remaining <expression>s are passed as they are, if they are referenced only once from the model body. Expressions with multiple references are tested for idempotency. Expressions which are supposedly non-idempotent are transformed into:

\[
<\text{new identifier}> : <\text{expression}>,
\]

after which rule 2 is applied.

The test for idempotency can be influenced by the user. Expressions which satisfy the Idempotency test get passed to
the model body on each of the multiple references, thereby introducing multiple copies of the Idempotent functions involved. This increases locality (eliminates global wiring) at the cost of multiple copies of the Idempotent function. There is however frequently a high chance that the Idempotent function can be absorbed in the model body without increase of computational complexity, due to the fact that local transformations can be applied to simplify the instance of the model body.

The following functions are marked Idempotent:

\_not, \_and

Example:

<table>
<thead>
<tr>
<th>model declaration:</th>
<th>model reference:</th>
<th>instance:</th>
</tr>
</thead>
<tbody>
<tr>
<td>model Example3 (a, b), a + b, endmod;</td>
<td>Example3 (x, -y)</td>
<td>x - y.</td>
</tr>
<tr>
<td>model Example3 (a, b), _and(a, _not(b)), endmod;</td>
<td>Example3 (x, _not(y))</td>
<td>_and(a, b).</td>
</tr>
</tbody>
</table>

**Semantics of models as compared to functions.** It should be noted that the semantics of model instantiation are fundamentally different from the semantics of function invocation.

Model instantiation is a process in which the instantiated body replaces the form <model identifier> <actual parameter list> with the model body with appropriate actual arguments substituted according to the specification given. An important consequence of the model instantiation process is that parallelism is fully propagated.

A function can be invoked only if all its actual arguments have been calculated. Given the actual values of the arguments the function calculates the function value. The function invocation process is sequential in nature, i.e. a function responds only with a new function-value if it has obtained a new set of argument-values.
The static communication conflict. Two functions which operate in parallel produce in general different values. This implies that the parallel algorithm depicted in figure 1.6.1 has an inherent communication conflict, as a labeling of two different values with one variable identifier has an inherent ambiguity.

Example:

\[
\begin{align*}
  z : & F(a, b), \\
  z : & G(c, d).
\end{align*}
\]

is invalid, as x cannot be used to represent the value of F and G all at the same time.

This ambiguous situation is allowed in a Von-Neumann language such as ALGOL, PASCAL, FORTRAN, etc. The value of x depends in such languages on execution order. The MoDL hardware description language will always flag this situation as an error, no matter how non-trivial the interconnect is which caused the error.

Figure 1.6.1.

We will now introduce the syntax and semantics of the MoDL "case expression", followed by the MoDL "register expression" and the MoDL "loop expression" in the next chapter.
1.6.1 The Case Expression

Dynamic communication. The case expression is used to activate the functions governed by the case expression selectively:

\[
\text{case } \text{Cond } \{ x: F(a), y: G(x, b) \} \text{ endcase.}
\]

The instances of \( F \) and \( G \) within the case expression are active if the condition \( \text{Cond} \) is true and inactive otherwise.

Dynamic communication requires mutual exclusion. The condition in the case expression determines which functions are active and which are not. No two active functions can however be used to govern the value of a single signal label. This implies that their guarding conditions should be mutually exclusive. Figure 1.6.3 gives an example. Automatic proofs for mutual exclusion will be performed (for a predefined class of cases) in a future version of MoDL.

The case expression itself also returns a value. Each guarded clause: \( \text{Cond1 } \{ \text{expr1}, \text{expr2}, \ldots \text{exprn} \} \) produces \( \text{exprn} \) as its value.

This can be used to describe similar situations in another form:

\[
x: \text{case } \text{Cond1 } \{ F(a, b) \}, \quad \text{or } x: \text{case } \text{Cond1 } \{ F(a, b) \},
\]

\[
\text{not } \text{Cond1 } \{ G(c, d) \}, \quad \text{default } \{ G(c, d) \}, \quad \text{endcase.}
\]

The first version of the case statement is frequently preferred if it is already known that the guarded clauses are mutually exclusive, as in Cond1 and not Cond1 here. The second version may be used to express here any condition which is complementary to all conditions which do define the signal value labelled with the
variable identifier \( x \).

**Example:**

```plaintext
case Cond1 \{ x: (a, b) \},
Cond2 \{ x: (c, d) \},
endcase.
```

The expression is valid if and only if the Conditions Cond1 and Cond2 are mutually exclusive.

**Figure 1.6.3. Mutual exclusion.**

### 1.6.2 Timing analysis

Timing analysis is a refinement of communication, as it takes the actual amount of time needed for the parallel computation into account. This aspect is treated in detail in the chapter III.7. The timing of a design is frequently dependent on physical constraints such as capacitive loading, output strengths of drivers etc. How these effects can be taken into account is described in chapter III.8.
1.7 Semantics of Synchronous Design

1.7.1 The Register Expression

MoDL represents a clocked register with the syntax:

\[ \langle \text{variable identifier} \rangle := \langle \text{expression} \rangle \]

Example:

\[ \text{Out} := \text{In}, \]

\[ \text{In} \rightarrow := \rightarrow \text{Out} \]

The \langle \text{variable identifier} \rangle in the register expression has an initial value which is kept unchanged during a complete clock period. The governing clock for a register is given by the loop expression which encloses the register expression.

1.7.2 The Loop Expression

MoDL implements a continuous clock with the syntax:

\[
\text{loop} \\
\langle \text{expression list} \rangle \\
\text{endloop}
\]

The expressions in the expression list within the loop are active for an indefinite number of clock periods.

A finite number of active clock cycles of a loop can be specified with the case expression:

\[
\text{loop} \\
\text{exitif Cond} \langle \text{expression list} \rangle, \\
\text{endloop}.
\]

The value specified in the last expression of the expression list, or the value of Cond if the expression list is empty, is the value of the loop expression. It is not usual that a loop is placed somewhere in an expression. It is a mere consequence of the general rule that every language construct returns a value that we need to make a remark about the return value of the loop expression.
Semantic implications of the synchronous design approach.

The fact that we do model so called algebraic loops explicitly within MoDL has its implications on the proof strength of MoDL, as communication conflicts are much better stated in a synchronous design. This are the major features of importance here:

1. There is no need to support Algebraic Loops, as registers are supported by the language. For instance, MoDL makes the user aware of a communication conflict in the following situations:

   \[ x: x + 1, \quad y: F(x, z), \quad z: G(y). \]

   Error message:
   \[ z: \text{Algebraic loop} \]

2. The following situation is allowed if and only if Cond1 and Cond2 are mutually exclusive:

   \[ \text{case Cond1} \{ y: F(x, z) \} \]
   \[ \text{case Cond2} \{ z: G(y) \} \]
I.8 Locality of Interconnect.

Models are ideally suited to describe parallel algorithmic structures in a hierarchical way. It is good practice to utilize model declarations in such a way that variable identifiers (and hence wiring) is kept local whenever possible. Nonlocal interconnect has to be specified through the use of the formal parameter-list of the model. Variable identifiers within a register expression form no exception to this rule. Variable identifiers which refer to formal arguments are subject to the parameter passing algorithm used for models. The effect of these rules will now be demonstrated with some simple examples.

Example 1:

```plaintext
model Counter (OutPut, NMax);
    case x=NMax {OutPut: 1, x := 0};
    default {OutPut: 0, x := x+1};
endcase,
endmodel;
```

Locality. The counter shown has a local state variable x, so all instances of this counter have different state variables, as the model body gets substituted at the place of the model reference, after parameter-passing.

Interconnect. The formal arguments of this model are OutPut and NMax, which are interconnected to specific arguments during the instantiation of the model.

Communication. OutPut is clearly an output-value as it is the left hand side of the assignment operator. This means that there is an unconditional communication with the outside world over this formal parameter.

The formal argument NMax is used as an input argument. An arbitrary expression can be used as actual argument. A parametrized counter results if the actual argument is actually a constant. The model gets instantiated into a counter with a variable reset condition if a <variable identifier> is passed as actual argument.

Modularity. The model declaration of the counter can be fully understood from its model description.

Hierarchy. The model declaration can, without any problem, be used in a hierarchical description of a larger design. One may think about the model of the counter as a new object, which
Semantics of MoDL

Locality of Interconnect.

becomes instantiated through a model reference. The actual arguments used in the model reference define the final form in which the counter is embedded in its environment.
I.9 Layout View

The layout view supports two locality aspects:

1. The notion of functionality \textit{inside and outside} a model, i.e. \textit{locality of place}.
2. The notion of the \textit{relative position} in the x and the y direction, as supported by the \textit{abutment algebra}.

**Locality of place in relation to functional specification**

A design may consist of basic layout cells, which have a certain functionality only when they are combined on a higher level, e.g. distributed logic with wired (N)or. This situation would violate the orthogonality concept in MoDL, as descriptive hierarchy should lead to electrical hierarchy. The converse is not true. Therefore the notion of pull-up and pull-down devices is needed at a higher level for example.

**Relative positioning**

For layout specification, MoDL supports

1. \textit{A modified argument list syntax} to specify the global layout of a cell.
2. \textit{An abutment algebra}. This bridges the gap between an algebraic description language and a layout based system. It allows the user to define new model descriptions, using \texttt{<model identifier>} as operands of the abutment operators - and + for respectively horizontal and vertical abutment. A \texttt{repetition factor} can be used as the first operand of the abutment operator.

More detailed information about the syntax of the abutment algebra is given in section II-6, together with examples.

**Orthogonality of abutment and communication.** The abutment algebra allows the description of a cell with \textit{feedthrough wiring}. 
Feedthrough-interconnects allow intracell communication or Vdd and Ground routing. The feedthrough "wire" can of course also be used inside the cell for communication purposes. A WiringCell can be used to adapt the order of arguments if they do not fit directly by abutment:

```model WiringCell ((x, (y, z), (x, (z, y))),
```

Another example is the specification of a gate-array with so-called channels. The keyword `channel` is now used to specify a mass interconnect for which the actual names of wires are left unspecified.

```
model WiringCell ((x, (y, z), (x, (z, y))),
```

Figure 1.9.1. Feedthrough wiring.

Figure 1.9.2. An application of feedthrough wiring.
This example shows that channels share the names used in the parameter lists. I.e. the variable in Channel3 matches the variable in Channel1.

The very simple feedthrough wire, defined as a layout issue of a cell to be used into the abutment algebra, has direct consequences on the (intra) cell communication as well as the (intended) mask layout. However, an important consequence of the MoDL approach is that the orthogonality of layout and communication is preserved this way.
I.10 Behaviour of Arithmetic Functions

The precise definition of the behaviour of arithmetic functions is of great importance in software and hardware design models, especially if an algorithm should be able to run for an extremely long period of time without user intervention. This point will be stressed in more detail at the end of this chapter under the heading arithmetic design.

MoDL supports two distinct arithmetic domains within model expressions: the integer domain, specified with square brackets:

```
[ <model expression list> ]
```

and the user defined arithmetic domain, specified with curly brackets:

```
{ <model expression list> }
```

These forms can either be used in a toplevel expressions followed by a terminator, ";" ";$" or ";&", or in a model expression. They initiate the model instantiation process in either of the specified arithmetic domains, when used in a top-level expression.

The default arithmetic domain for a Model is the integer domain, i.e. the operations \(+\), \(-\) and \(*\) are all interpreted as integer operations. Instances of such models use integer arithmetic irrespective of the arithmetic domain in which they were instantiated.

A model with a default user-defined domain is specified with the "dsmodel" (digital signal processing model) keyword instead of the "model" keyword. The instantiation process for model expressions in the user-defined domain maps the operators, \(+\), \(-\) and \(*\) into the user-defined operators \(+\), \(-\) and \(*\).

The integer domain provides the user with arbitrary-precision integers, giving maximum flexibility. The user-defined domain however can use a finite precision arithmetic with specific overflow provisions. The syntactic form:

```
ov{ <model expression list> }
```

maps the operators \(+\), \(-\), \(*\) within the model expression list into the
operators of the user-defined domain ov|+, ov|-, ov|·.

The value produced by the forms:

```
  *   [ <model expression list> ]
  |   <model expression list> |
  ov| <model expression list> |
```

is always the value produced by the last expression in the expression list.

The issues of Locality of interconnect, Communication and Modularity can also be shown on a model with user-defined arithmetic. A SecondOrderSection of a digital filter for instance, can be defined as:

```
dspmodel SecondOrderSection (u, y, b2, b1, b0, c1, c0),
  y := b2*tmp + b1*x1 + b0*x2,
  tmp := ov[y + c1*x1 + c0*x2],
  x1 := tmp,
  z2 := x1,
endmod;
```

Locality. The state variables x1 and x2 are local, as well as the internal variable tmp.

Communication. The formal argument u is the input of the SecondOrderSection, the formal argument y is its (unconditional) output.
The formal arguments b2, b1, b0, c1 and c0 will most likely get a constant as actual argument. This leads to a specific instance of the SecondOrderSection. An expression may be specified as well: this would define an adaptive SecondOrderSection.

Modularity. The model declaration of the SecondOrderSection given defines all aspects of locality, communication and parallelism found in the SecondOrderSection. One may think about the dspmodel of the SecondOrderSection as a new object, which becomes instantiated through a model reference. The exact implementation of the arithmetic used in the SecondOrderSection can be specified through the introduction of user-defined functions for |+|, |−|, |·| etc. The actual arguments used in the model reference define the final form in which the SecondOrderSection
Semantics of MoDL Behaviour of Arithmetic Functions

communicates with its environment.

**Fractional Arithmetic**

The idea of the user defined arithmetic domain is that a user can specify the arithmetic functions \(+, \times\) together with \(\circ\ven\circ\) and \(\circ\ven\circ\) in any desired way.

The implementation of a fractional number system becomes quite simple if integer values are used to represent the operands. This assumes however the existence of an integer value which represents the number 1.0 and hence defines the precision of the number system. The actual value of the \(\text{<variable identifier>}\) one determines the actual precision of the number system.

**Fractional Constants**

Fractional constants can be specified in a model expression using floating point notation. The constant is converted to internal representation by multiplication with the global system constant one and rounding to an integer value. The following examples show some values for a 16-bit and a 12-bit number system. Even financial systems based on the cent as a basic unit are easily defined.

**Examples:**

\begin{align*}
\text{one: } 2^{-15}; & \quad \text{16-bit number system} \\
1.0 & \rightarrow 32768 \text{ or } 8000\text{hex} \\
1.0-1 & \rightarrow 32767 \text{ or } 7FFF\text{hex} \\
-1.0 & \rightarrow -32768 \text{ or } -8000\text{hex} \\
\text{one: } 2^{-11}; & \quad \text{12-bit number system} \\
1.0 & \rightarrow 2048 \text{ or } 800\text{hex} \\
1.0-1 & \rightarrow 2047 \text{ or } 7FF\text{hex} \\
-1.0 & \rightarrow -2048 \text{ or } -800\text{hex} \\
\text{one: } 100; & \quad \text{financial type number system} \\
1.32 & \rightarrow 132
\end{align*}

**Fractional Addition**

Fractional addition is exactly the same as integer addition if one uses an arbitrary precision number system. Finite number systems are represented by a mapping to a finite representation and a
Semantics of MoDL

Behaviour of Arithmetic Functions

mapping back into the signed magnitude domain. These transformations are needed if the range of the function should be the same as its domain, i.e. if the number system needs to be closed. The most common choices here are:

1. 2-complement overflow.
2. Saturation on 1.0-1 and -1.0.
3. Saturation on 1.0-1 and -1.0+1.
4. Folding.

![Overflow Characteristics]

*Figure 1.10.1. Different overflow characteristics.*

The implemented arithmetic uses 2-Complement overflow.

It makes a difference if one implements the user defined arithmetic domain based on binary operators \(+\), \(-\) or on a nary operator \(\oplus\) which accumulates intermediate results, as an accumulator-based design may be designed with extended precision on intermediate results to avoid the need for overflow corrections on intermediate results.

The implemented arithmetic is accumulator-based and uses an arbitrary precision on intermediate results. It requires slightly different model expansion rules to implement an accumulator-based system as compared to a binary operator based system, as an accumulator-based system works with a single operator \(+\), with negated operands whenever subtraction is required instead of addition.

**Fractional Negation and Subtraction**

An accumulator-based fractional number system implements \(a-b\) as \([a+(-b)]\). This can be implemented as integer negation in combination with fractional addition.

Fractional negation, like in \(-a\), needs however be implemented explicitly if the number system is not symmetric like the 2-

page 1-10-4

6.34
Semantics of MoDL Behaviour of Arithmetic Functions

complement number system which represents all values in the range -1.0..1.0-1, or in hex 16-bit 2-complement form from 8000hex to 7FFFhex.

Note that [-a] gives -1.0 if a is -1.0, due to 2-complement overflow of the number system.

Fractional Multiplication

Fractional multiplication can be implemented with integer multiplication if one takes into account that the result should be scaled back into the original domain. So \(a \times b\) with \(a:1.0\) and \(b:1.0\) would give one*one, which should be divided by one to get a closure of the fractional number system under multiplication.

This division by one gives a rest, which may either be discarded (truncation) or be used to round either to the nearest value or towards zero. The implemented arithmetic uses rounding to the nearest value.

A finite number system like the 2-complement number system gives raise to overflow if it is not symmetric and does include an element of magnitude 1.0. The effect of a finite unsymmetrical number system becomes apparent if one wants to multiply -1.0*1.0 in a such a fractional number system, as this gives without special provisions for overflow protection -1.0 instead of 1.0.

The implemented arithmetic uses an unrestricted number system on fractional multiplication, allowing -1.0*1.0 to be 1.0.

Arithmetic Design

The proper design of an arithmetic system is a very important issue, as inconsistencies, like for instance -1.0*1.0 -> -1.0, in an arithmetic domain can lead to permanent oscillations in digital filters. It will be clear that such situations cannot be tolerated in systems which should be run without user attendance for long periods, such as telephone exchanges, control systems and the like. It is therefore of extreme importance that a designer takes these issues into consideration before the actual construction of any hardware. Our discussions in the previous sections should be sufficient to make a good choice from several alternatives.

The implemented arithmetic system is easily modified to fit a user defined arithmetic system. The MoDL simulator can then be used to evaluate this arithmetic system in the context of an application.

page l-10-5
1.11 Automatic Transformations for Interconnect

A top-level model expression of the form:

\[
\begin{align*}
[mmodel expressionlist] \{mmodel expressionlist\}; \text{ or } \\
\{mmodel expressionlist\}
\end{align*}
\]

is needed to start the instantiation of any expression with its embedded model references. The instantiation process itself is performed by a process called \textit{Interconnect}. MoDL implements this process by a central function \textit{modelexp}, which takes care of all instantiation rules. To keep the software simple and extensible, it was decided that there would be a separate function which would handle the instantiation of all individual operators. A \textit{connectfunction} is a specific model instantiation function for a specific operator. As the instantiation process is recursive in nature, it is likely that connectfunctions will themselves call the function \textit{modelexp}, to complete the recursion.

This is the syntax of the \textit{connectfunction} declaration:

\[
\begin{align*}
\text{connectfunction} \ <\text{function identifier}> \ (<\text{formal parameter}>), \\
\text{<body>}
\end{align*}
\]

The function \textit{modelexp} invokes the \textit{connectfunction} \textit{<function identifier>} with as actual argument an uninstantiated \textit{<model expression>}. The \textit{connectfunction} will apply a specific transformation rule for the \textit{<function identifier>} given. A coherent set of \textit{connectfunctions} can be designed to cover a specific application area. Rewrite rules for logic expressions cover one such area. Automatic simplification rules for algebraic expressions cover another one, which has a default implementation in MoDL.

The following list of transformation rules summarizes the effect of all \textit{connectfunctions} implemented in the module \texttt{mod.mod}; additional \textit{connectfunctions} are available in the module \texttt{dsp.mod}. These cover all transformation rules associated with the application of fractional arithmetic.

All rules given describe a transformation of a \textit{<model expression>} on the left of the arrow into a \textit{<model expression>} on the right.
Semantics of MoDL Connectfunctions

These rules are applied during the Interconnect phase.

The following abbreviations are used:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Item in the syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr, A, X, Y, Cond</td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td>Var</td>
<td>&lt;variable identifier&gt;</td>
</tr>
<tr>
<td>Fn</td>
<td>&lt;function identifier&gt;</td>
</tr>
<tr>
<td>Con</td>
<td>&lt;constant&gt; (Numeric)</td>
</tr>
<tr>
<td>()</td>
<td>empty list</td>
</tr>
</tbody>
</table>

Any form which has been subject to Interconnect is marked with a postfix quote, i.e. Var is a <variable identifier> before Interconnect and Var' is its version after Interconnect.

<table>
<thead>
<tr>
<th>Con</th>
<th>Con'</th>
</tr>
</thead>
<tbody>
<tr>
<td>fix {Con' * one}</td>
<td>(Integer)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Con</th>
<th>Con'</th>
</tr>
</thead>
<tbody>
<tr>
<td>fix {Con' * one}</td>
<td>(Floating point)</td>
</tr>
</tbody>
</table>

Integer constants are kept unchanged. Floating point constants are multiplied by the user defined constant one and converted into an integer.

false -> false
true -> true
Var -> Var'

Variable identifiers are transformed into a new, unique variable identifier, if used locally. Otherwise they take the value of the corresponding actual argument.

1. Given:

    model Fn (A, B, ...),
    <modelbody>
    endmodel;

    Fn (X, Y, ...) -> instance of <modelbody> with argument substitutions:
    A -> X
    B -> Y
    ... as specified in chapter 1-5.

page 1-11-2
2. Given:

\[
\text{connectfunction } F_n (\text{Args}).
\]

\[
\text{<body>}
\]

\[
\text{endfun;}
\]

\[
F_n (X, Y, \ldots) \rightarrow \text{invokes the connectfunction } F_n. \text{ The connectfunction } F_n \text{ is invoked with the list } (X, Y, \ldots) \text{ as single actual argument.}
\]

3. All other cases:

\[
F_n (X, Y, \ldots) \rightarrow F_n (X', Y', \ldots)
\]

\[
\text{eval (Expr)} \rightarrow \text{eval (Expr)},
\]

i.e. \text{Evaluated expression}

This rule is used to pass a LISP expression \text{Expr} in evaluated form into a <model expression>.

\[
\text{Var: Expr} \rightarrow \text{Var': Expr'}
\]

\[
\text{Var:= Expr} \rightarrow \text{Var':= Expr'}
\]

These rules connect the <expression> to the <variable identifier> \text{Var} in the Context given. This Context can either be the top-level, a loop expression or a case expression. This aspect is taken into initial consideration by the connectfunctions ":=" and ":=". The communication functions ":=" implement the fine details.

\[
[\ldots, \ldots] \rightarrow \text{Integer domain with operators } [+], [\ast].
\]

The square brackets affect nested + and * operators.

\[
\text{init (Var, Expr)} \rightarrow \text{Var' (with initial value assigned)}
\]

An explicit initial value can be assigned to a <variable identifier> in a register expression. The form \text{init (Var, Value)} can be used everywhere were \text{Var} could be used.

\[
\text{case Cond1 \{ \ldots\}, } \quad \text{case Cond1' \{ \ldots\}},
\]

\[
\text{Cond2 \{ \ldots\}, } \rightarrow \text{Cond2' \{ \ldots\}},
\]

\[
\text{endcase, } \quad \text{endcase},
\]
Semantics of MoDL

Connectfunctions

The case expression and the loop expression both define a context which affects the effect of the variables used inside the language construct. This aspect is taken into initial consideration by the connectfunctions for these language constructs. The fine details in this respect are covered by the communicationfunctions.

The following list of transformation rules summarizes the effect of all connectfunctions on a <model expression> as it is implemented in the module dsp.mod:

**connectfunction +**

\[A' + \ldots Con' + X \rightarrow Con' + A' + \ldots + X\]
\[A' + \ldots X' + Y \rightarrow A' + X' + \ldots + Y\]
\[A' + \ldots (X' + Y' + \ldots) + Z \rightarrow A' + X' + Y' + \ldots + Z\]

The rules given describe automatic simplification rules for the + operator. The rules accumulate constant terms into a single constant Con. Nested sums are merged into the top level expression.

\[0 + A' + \ldots \rightarrow A' + \ldots\]
\[Con' + () \rightarrow Con'\]
\[A' + () \rightarrow A'\]

The additive constant is eliminated if it is found to be zero. The + operator disappears if the list of remaining terms is empty.

**connectfunction -**

\[X' - Y' \rightarrow X' + (- Y')\]

Addition is preferred over subtraction.

- Con' \rightarrow (-Con')
- A' \rightarrow (-A')
- (-A') \rightarrow A'

Some simplifications on negation.

- ((#n': Con'*X') + #n') \rightarrow (#n': (-Con')*X') + #n'

Constants in excess of 1.0 are frequently excluded in
programmable hardware. The connectfunction $\cdot$ transforms such constants in the form given, a negation of this form can however be rewritten with a negative constant as indicated in the above rule.

\[-(A' + B') \rightarrow (-A') + (-B')\]

**Addition is preferred over subtraction.**

\[-\text{Con'} \cdot X' \rightarrow (-\text{Con'}) \cdot X']\]
\[-[\text{Con'} \cdot X'] \rightarrow [-\text{Con'} \cdot X']\]
\[-X' \rightarrow -X']\]

Some straightforward rules.

**connectfunction $\cdot$**

\[-A' \cdot X' \rightarrow (\text{Con'/2} \cdot A' \cdot X') + x'\]

Constants in excess of 1.0 are frequently excluded in programmable hardware. The connectfunction $\cdot$ transforms such constants in the form given, with $x$ a newly created intermediate variable identifier.

\[-\text{Con'} \cdot () \rightarrow \text{Con'}\]
\[-A' \cdot () \rightarrow A'\]

Rules for the removal of the $\cdot$ operator if only a single term is present.
I.12 Automatic Transformations for Communication

Any top-level model expression of the form:

\[
\begin{align*}
\text{[:model expressionlist]} & \text{;} \\
\text{[:model expressionlist]} & \text{;}
\end{align*}
\]

is first subject to Interconnect and then to Communication analysis. The Communication analysis process performs actions of three kinds:

1. Further simplification of the result after Interconnect.
2. Transformation of the expanded top-level model expression into executable order, for a von-Neumann type language, c.q. processor.
3. Verification of the basic semantics of parallelism as defined in chapter 1-6.

The notation and methodology used to present the actions taken by the connectfunctions will now be used to express the details of the Communication analysis phase. These details are implemented by so-called communication functions, or shortly commufunctions. Whenever appropriate we will introduce the diagrams used in chapter 6 to express nesting of contexts.

false -> false
true -> true
Con -> Con'
Var -> Var'

The Communication analysis passes false, true and constants as they are. Variable identifiers can only be used in a ⟨model expression⟩ if their value has been calculated before. If not so, the Communication analysis will place the demanded expression in front of the expression which demands it. If the demanded expression is part of any nested context, then a demand for any of the nested contexts will be generated to satisfy the current demand.

1. From within an expanded ⟨model expression⟩, given:
Semantics of MoDL

Communication functions

\[ \text{commfunction } F_n (\text{Args}), \]
\[ \text{<body> endfun;} \]

\[ F_n (X, Y, \ldots) \rightarrow \text{invokes the communication function } F_n. \text{ The communication function } F_n \]
\[ \text{is invoked with the list } (X, Y, \ldots) \text{ as single actual argument.} \]

2. From within an expanded \(<\text{model expression list}>\), given:

\[ \text{contextfunction } F_n (\text{Args}), \]
\[ \text{<body> endfun;} \]

\[ F_n (X, Y, \ldots) \rightarrow \text{invokes the contextfunction } F_n. \]

\[ \text{The contextfunction } F_n \text{ is invoked with the list } (X, Y, \ldots) \text{ as single actual argument.} \]

3. All other cases:

\[ F_n (X, Y, \ldots) \rightarrow F_n (X', Y', \ldots) \]
\[ [X, Y, \ldots] \rightarrow [X', Y', \ldots] \]
\[ [X] \rightarrow X' \]
\[ [X, Y, \ldots] \rightarrow \{X', Y', \ldots\} \]
\[ [X] \rightarrow X' \]

The last expression within \([\ldots]\) or \(\{\ldots\}\) is the result of the syntactical form. The brackets can be removed if there is only a single expression.
Semantics of MoDL

Var := Expr  ->  Sets within context, Var := Expr'

Var := Var1  ->  If there is given in the current context any demand for Var which is still in progress -> Error: Algebraic Loop.

-> Sets within context, Var := 'Var1,

case Condl { .... }, -> case Condl' { .... },
Cond2 { .... },  Cond2' { .... },
endcase,  endcase,

The case expression can be demanded because of the fact that it is encountered during Communication analysis, or because of a demand to a variable identifier within any of its Cond branches. In the first case the whole case is taken into demand, in the latter case the case is split up to satisfy the demand in question.

loop  ->  loop  loop construct for simulator.
      ...  clock ( .... ),
endloop,  endloop.

The loop expression can be demanded because of the fact that it is encountered during Communication analysis, or because of a demand to a variable identifier within its expression list.
Appendix I-A : Syntax Diagrams of MoDL.

letter :

\[
\text{letter} : \text{a..z} \quad \text{A..Z}
\]

digit :

\[
\text{digit} : \text{0..9}
\]

character :

\[
\text{character} : \text{asci}(32 \ldots 127)
\]

comment :

\[
\text{comment} : \% \text{BOLN} \quad \text{character} /\* \text{character} */
\]

identifier :

\[
\text{identifier} : \text{letter} \quad \# \quad \text{digit} \quad @
\]

unsigned integer :

\[
\text{unsigned integer} : \text{digit}
\]
infix operator:

prefix operator:

expression:

constant

variable identifier

prefix operator { expression

expression \{ infix operator \} expression

function identifier \{ actual parameter list

variable identifier \{ expression

case

loop

for

foreach

\textit{page I-A-3}
expressionlist:

[Diagram]

guarded clause:

[Diagram]

case:

[Diagram]

exitif:

[Diagram]

for:

[Diagram]

foreach:

[Diagram]

page I-A-4
loop:

- loop
- expression
- exitif

actual parameter list:

- (1)
- expression

formal parameter list:

- (1)
- formal parameter

local declaration:

- local
- variable identifier

body:

- local declaration
- expressionlist

abut parameter:

- formal parameter
- formal parameter list
abut parameter list:

[Diagram]

abut operator:

[Diagram]

abut subexpr:

[Diagram]

abut expression:

[Diagram]

abut command:

[Diagram]

rotate command:

[Diagram]
function declaration:

(function) function identifier

formal parameter list → body → endfun

macro declaration:

(macro) macro identifier

formal parameter list → body → endmac

subroutine declaration:

(subr) subroutine identifier

formal parameter list → body → endsub

modelexpression:

(constant)

list

variable identifier

prefix operator { modelexpression }

modelexpression { infix operator } modelexpression

function identifier { actual parameter list }

variable identifier :: { modelexpression }

variable identifier ::{ modelexpression }

modelcase

modelloop

(1) model expressionlist (1)

(1) model expression list (1)

(ovl) model expression list (1)
modelbody
model expressionlist:

model guarded clause:

model case:

model exitif:

model loop:

model declaration:

dspmodel declaration:
identifier list:

```
(identifier
  (identifier)
)
```

typefunction:

```
(typefunction
  prettyfunction
  connectfunction
  commufunction
  contextfunction
)
```

properlyfun declaration:

```
(properlyfun function
declaralition
  (identifier list
    (identifier list)
    (formal parameter
      (body
        endfun)
      (endfun)
    )
  )
)
```

typefun declaration:

```
(typefun function
declaralition
  (identifier list
    (identifier list)
    (formal parameter
      (body
        endfun)
      (endfun)
    )
  )
)
```

toplevel expression:

```
toplevel expression
  function declaration
  subroutine declaration
  macro declaration
  model declaration
  dsmodel declaration
  typefun declaration
  properlyfun declaration
  abut command
  rotate command
  modelexpression
  expression
```

Page I-A-9
II.0 Organisation of the Manual

This manual is not intended as an introductory guide to MoDL, it is rather a reference guide to its precise definition. Tables 1, 2, 3 and 4 in this general introduction can be used to get a quick survey of the the issues treated in this manual. We used a **bold** typeface at places where new terms are defined. The *italic* typeface is used to improve readability, such as in *and and or* and to distract attention from a certain implementation detail.
### Part II at a glance

The MoDL programming environment

<table>
<thead>
<tr>
<th>Function type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Recognizer</strong></td>
<td></td>
</tr>
<tr>
<td><strong>pair(x), atom(x),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>fixnum(x), bignum(x), fionum(x),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>integer(x), number(x),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>empty(x), trucname(x), name(x),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>datatype(x),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>member(x, list), memq(x, list),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>length(list), length(vector).</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Constructor</strong></td>
<td></td>
</tr>
<tr>
<td><strong>cons(x, y), list(x1, x2, ..., xn)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>oblst(),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>'op(x1, x2, ..., xn),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>reverse(list), append(l1, l2, ..., ln),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>copy(list), copy(vector).</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Selector</strong></td>
<td></td>
</tr>
<tr>
<td><strong>first(pair), rest(pair),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>second(list), third(list), fourth(list),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>rfirst(x) is rest(first(x)),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>frest(x) is first(rest(x)).</strong></td>
<td></td>
</tr>
<tr>
<td><strong>This holds for an arbitrary</strong></td>
<td></td>
</tr>
<tr>
<td><strong>number of r and f prefixes.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>composite: assoc(x, alist), assq(x, alist),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>nthnode(list, n).</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Modifier</strong></td>
<td></td>
</tr>
<tr>
<td><strong>replacefirst(pair, x), replacerest(pair, x),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>delete(x, list).</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Comparator</strong></td>
<td></td>
</tr>
<tr>
<td><strong>x == y, x = y, x &lt;&gt; y,</strong></td>
<td></td>
</tr>
<tr>
<td><strong>n1 &lt; n2, n1 &lt;= n2, n1 &gt;= n2, n1 &gt; n2,</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td></td>
</tr>
<tr>
<td><strong>not(x), x1 and x2 and ... and xn</strong></td>
<td></td>
</tr>
<tr>
<td><strong>x1 or x2 or ... or xn</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Assignment</strong></td>
<td></td>
</tr>
<tr>
<td><strong>name: x, set(name, x)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>setunbound(name), bound(name),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>pop(name), push(x, name),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>setprotect(name).</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Property list</strong></td>
<td></td>
</tr>
<tr>
<td><strong>put(name, p, v), get(name, p),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>remprop(name, p),</strong></td>
<td></td>
</tr>
<tr>
<td><strong>plist(name), setplist(name, v).</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Part II at a glance

The MoDL programming environment

<table>
<thead>
<tr>
<th>Function type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function definition</strong></td>
<td><code>def(list), getdef(name)</code>, <code>function f(x1, x2, ..., xn)</code>, <code>function f x</code>, <code>subroutine f(x1, x2, ..., xn)</code>, <code>subroutine f x</code>, <code>macro f(x1, x2, ..., xn)</code>, <code>macro f x</code></td>
</tr>
<tr>
<td><strong>String</strong></td>
<td><code>explode(name), implode(list)</code>, <code>explodenum(name)</code>, <code>concat(name1, name2, ..., name)n</code>, <code>gensym(name)</code>, <code>ascii(n), ord(char)</code></td>
</tr>
<tr>
<td><strong>Vector</strong></td>
<td><code>allocvec(n,v)</code>, <code>setvec(vec, n, v), getvec(vec, n)</code></td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td><code>n1 + n2 + ... + nn, n1 - n2</code>, <code>n1 * n2 * ... * nn</code>, <code>divide(n1, n2), n1 / n2, mod(n1, n2)</code>, <code>n1 % n2, abs(n)</code>, <code>max(n1, n2), min(n1, n2), sin(n), cos(n), tan(n), log(n)</code>, <code>asin(n), acos(n), atan(n), sqrt(n)</code></td>
</tr>
<tr>
<td><strong>IOPort</strong></td>
<td><code>infile(file), outfile(file)</code>, <code>open(file)</code>, <code>close(port), tty(port), charcnt(port)</code></td>
</tr>
<tr>
<td><strong>Reader</strong></td>
<td><code>getsyntax(char), setsyntax(char, syntype)</code>, <code>getmacro(char), setmacro(char, macrotype)</code></td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td><code>readlist(), preadlist(port)</code>, <code>readatom(), preadatom(port)</code>, <code>readchar(), preadchar(port)</code>, <code>raw(), scan()</code>, <code>load(file, ext)</code>, <code>transload(file, ext1, ext2)</code></td>
</tr>
<tr>
<td>Function type</td>
<td>Name</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>Output</td>
<td><code>printlist(x_1, x_2, ..., x_n)</code>,</td>
</tr>
<tr>
<td></td>
<td><code>pprintlist(port, x_1, x_2, ..., x_n)</code></td>
</tr>
<tr>
<td></td>
<td><code>printatom(x_1, x_2, ..., x_n)</code></td>
</tr>
<tr>
<td></td>
<td><code>pprintatom(port, x_1, x_2, ..., x_n)</code></td>
</tr>
<tr>
<td></td>
<td><code>newln()</code>, <code>pnewln(port)</code></td>
</tr>
<tr>
<td>Evaluation</td>
<td><code>'x</code>, <code>eval(x)</code>, <code>apply(fn, args)</code></td>
</tr>
<tr>
<td></td>
<td><code>case</code></td>
</tr>
<tr>
<td></td>
<td><code>e10 { e11, e12, ..., e1n }</code>,</td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
</tr>
<tr>
<td></td>
<td><code>em0 { em1, em2, ..., emn }</code>,</td>
</tr>
<tr>
<td></td>
<td><code>endcase</code></td>
</tr>
<tr>
<td></td>
<td><code>loop</code></td>
</tr>
<tr>
<td></td>
<td><code>e1, e2, ..., em</code></td>
</tr>
<tr>
<td></td>
<td><code>endloop</code></td>
</tr>
<tr>
<td></td>
<td><code>exif e0 { e1, e2, ..., em }</code></td>
</tr>
<tr>
<td></td>
<td><code>foreach el in list do</code></td>
</tr>
<tr>
<td></td>
<td><code>e1, e2, ..., en</code></td>
</tr>
<tr>
<td></td>
<td><code>endfor</code></td>
</tr>
<tr>
<td></td>
<td><code>for x : x1 step x2 until x3 do</code></td>
</tr>
<tr>
<td></td>
<td><code>e1, e2, ..., en</code></td>
</tr>
<tr>
<td></td>
<td><code>endfor</code></td>
</tr>
<tr>
<td></td>
<td><code>forexp x1 repeat x2 until x3 do</code></td>
</tr>
<tr>
<td></td>
<td><code>e1, e2, ..., en</code></td>
</tr>
<tr>
<td></td>
<td><code>endfor</code></td>
</tr>
<tr>
<td></td>
<td><code>catch(label, body)</code>, <code>throw(label, x)</code>,</td>
</tr>
<tr>
<td></td>
<td><code>trace(fn)</code>, <code>untrace(fn)</code></td>
</tr>
<tr>
<td>Memory</td>
<td><code>gc()</code>, <code>removename(truename)</code></td>
</tr>
<tr>
<td>management</td>
<td></td>
</tr>
<tr>
<td>Driver</td>
<td><code>toplevel_driver()</code>, <code>driver(in, out)</code></td>
</tr>
<tr>
<td></td>
<td><code>break(x)</code></td>
</tr>
<tr>
<td>Environment</td>
<td><code>shell()</code>, <code>quit()</code></td>
</tr>
<tr>
<td></td>
<td><code>time()</code>, <code>system(truename)</code></td>
</tr>
<tr>
<td></td>
<td><code>kill(truename)</code></td>
</tr>
<tr>
<td></td>
<td><code>fork(truename)</code>, <code>exec(truename)</code></td>
</tr>
<tr>
<td></td>
<td><code>signal(truename)</code></td>
</tr>
</tbody>
</table>
## Part III at a glance

The MoDL parallel language, c.q. hardware description language.

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model definition</strong></td>
<td><code>modelldef(list), getmodelldef(name), model f(x1, x2, ..., xn), dsmodel f(x1, x2, ..., xn)</code></td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td><code>x, case(list), loop(list), exitif(list), name := x</code></td>
</tr>
<tr>
<td><strong>Support</strong></td>
<td><code>support(ex1, ex2, ..., exn), pins(truename, list), dt(x, n), demandcase(list), datacase(list), cap(truename, n)</code></td>
</tr>
<tr>
<td><strong>Domain</strong></td>
<td><code>[ ex1, ex2, ..., exn ], [ ex1, ex2, ..., exn ], ov[ ex1, ex2, ..., exn ]</code></td>
</tr>
<tr>
<td><strong>Arithmetic +</strong></td>
<td><code>[ ex1 + ex2 + ... + exn ] with operator &quot;[+]&quot;</code>, <code>[ ex1 + ex2 + ... + exn ] with operator &quot;[+]&quot;</code>, <code>ov[ ex1 + ex2 + ... + exn ] with operator &quot;ov[+]&quot;</code></td>
</tr>
<tr>
<td>**Arithmetic ***</td>
<td><code>[ ex1 * ex2 * ... * exn ] with operator &quot;[*]&quot;</code>, <code>[ ex1 * ex2 * ... * exn ] with operator &quot;[*]&quot;</code>, <code>ov[ ex1 * ex2 * ... * exn ] with operator &quot;ov[*]&quot;</code></td>
</tr>
<tr>
<td><strong>Representation</strong></td>
<td><code>decode(n), encode(n), one</code></td>
</tr>
<tr>
<td><strong>Boolean</strong></td>
<td><code>_or(x1, x2, ..., xn), _nor(x1, x2, ..., xn), _and(x1, x2, ..., xn), _nand(x1, x2, ..., xn), _not(x1, x2, ..., xn)</code></td>
</tr>
<tr>
<td><strong>Word operations</strong></td>
<td><code>bool(x1, x2, op, mask)</code></td>
</tr>
</tbody>
</table>
II.1 The MoDL Programming Environment

Figure II-1.1 gives a survey of the MoDL implementation. It shows how parallel algorithms and hardware descriptions are mapped to the MoDL list level by means of the MoDL read and print functions. Part II of the reference manual defines the precise details of the MoDL programming environment. The specific tools which MoDL implements for hardware description are given in part III of the manual.

Figure II-1-1. A survey of the MoDL implementation
II.2 Recognizer Functions

The basic-recognizer functions take any MoDL data structure as their argument and return true if their argument is a specific data structure and false otherwise.

| Basic recognizer functions | pair(x), atom(x), fixnum(x), bignum(x), flonum(x), integer(x), number(x), empty(x), truename(x), name(x) |

The table below gives a survey of their operation:

<table>
<thead>
<tr>
<th>data structure</th>
<th>recognizer function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>empty</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>123</td>
<td>true</td>
</tr>
<tr>
<td>1234567890</td>
<td>true</td>
</tr>
<tr>
<td>3.14</td>
<td>true</td>
</tr>
<tr>
<td>‘cat’</td>
<td>true</td>
</tr>
<tr>
<td>‘(x y z)’</td>
<td>true</td>
</tr>
</tbody>
</table>

`empty(x)` returns true if `x == false`, otherwise false.

```
function empty (x),
  z == false,
endfun;
```

`fixnum (x)` returns true if `<x>` is an integer in the range `-fixnum-1 ... fixnum`, otherwise false.

```
function fixnum (x),
  case
    <z is a "small" integer> {true},
    default {false},
  endcase,
endfun;
```

`bignum (x)` returns true if `x` is an integer which exceeds the size limitations of `fixnum`'s.
function bignum (x),
case
  <x is a "big" integer> {true},
  default {false},
endcase,
endfun;

flonum (x) returns true if x is a floating point number, otherwise false.

function flonum (x),
case
  <x is a floating point number> {true},
  default {false},
endcase,
endfun;

truename (x) returns true if <x> is a truename, otherwise false.

function truename (x),
case
  <x is a truename> {true},
  default {false},
endcase,
endfun;

pair (x) returns true if <x> is a pair, otherwise false.

function pair (x),
case
  <x is a pair> {true},
  default {false},
endcase,
endfun;

The following recognizer functions are expressed in the given primitive recognizer functions:

atom (x) returns true if <x> is not a pair, false otherwise.

function atom (x),
  not (pair (x)),
endfun;

integer (x) returns true if <x> is a fixnum or a bignum, false otherwise.
function integer (x),
  fixnum (x) or bignum (x),
endfun;

number (x) returns true if <x> is a fixnum, a bignum or a flonum, false otherwise.

function number (x),
  fixnum (x) or bignum (x) or flonum (x),
endfun;

name (x) returns true if <x> is false or if <x> is a truename, false otherwise.

function name (x),
  empty (x) or truename (x),
endfun;

datatype (x) returns the datatype of <x>.

function datatype (x),
  <the type of the data structure x>,
endfun;

<table>
<thead>
<tr>
<th>Possible values returned by datatype</th>
<th>data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>empty</td>
</tr>
<tr>
<td>123</td>
<td>fixnum</td>
</tr>
<tr>
<td>'(x y z)</td>
<td>pair</td>
</tr>
<tr>
<td>1234567890</td>
<td>bignum</td>
</tr>
<tr>
<td>3.14</td>
<td>flonum</td>
</tr>
<tr>
<td>'cat</td>
<td>truename</td>
</tr>
<tr>
<td>'('a, b, c')</td>
<td>vector</td>
</tr>
</tbody>
</table>

member (x, list), returns false if <x> is not included in <list>. The function returns the remaining list, beginning with the element which is "=" to <x> if <x> is a member of the given list.

memq (x, list), returns false if <x> is not found in <list>. The function returns the remaining list, beginning with the element which is "==" to <x> if <x> is a member of the given list.
function member (x, list).
    case
        atom (list) {false},
        x = first (list) {list},
        default {member (x, rest (list))},
    endcase,
    endfun;

   member ('a, '(x y z))& -> false
   member ('y, '(x y z))& -> (y z)

The function memq uses "==" where member uses the more general "=" for comparison. The functions are otherwise identical.

length (x) returns the length of a list or vector.

function length (x),
    local (l),
    case
        vector (x) {<the length of the vector>},
        empty (x) {0},
        atom (x) {<error message>},
        default {l: 0,
            loop
                exitif atom (x) {l},
                l: l + 1, z: rest (z),
            endloop},
    endcase,
    endfun;
II.3 Operations on MoDL Data

The operations described in this chapter can be divided into Constructor functions, Selector functions and Modifier functions. Each of these function classes implements primitives, which are self supporting, and composite functions which do depend on other primitives defined in part II of this manual. It is in this way that the functions are best characterized by type. A good way to study the manual is to first concentrate on the primitive Constructors, Selectors and Modifiers. A second tour through the manual may then be taken to study the composites.

II.3.1

II.3.1.1 Primitive constructor functions

\texttt{cons} (x, y) constructs a new node with x in the first-cell and y in the rest-cell of the node.

\begin{verbatim}
  A general node
  \begin{tabular}{|c|c|}
    \hline
    first & rest \\
    \hline
    x & y \\
    \hline
  \end{tabular}

  Result of \texttt{cons}(x,y)

\end{verbatim}

\texttt{list} (x1, x2, ... xn) constructs a linear list of arbitrary size with as elements x1, x2, ... xn. list is an example of a function which takes an arbitrary number of evaluated arguments.

\begin{verbatim}
  function list (x1, x2, ... xn),
    case
      n=0  \{false\},
    default \{\texttt{cons} (x1, \texttt{list} (x2, ... xn))\},
    endcase,
  endfun;

  list ('a, 'b, 'c) == \texttt{cons} ('a,
    \texttt{cons} ('b,
      \texttt{cons} ('c, \texttt{false})))

  \rightarrow (a b c)
\end{verbatim}

Note: the evaluation mechanism for functions with an arbitrary number of arguments already passes the arguments of a function in evaluated form in the form of a list, thereby executing the above definition of list. This shows why the actual implementation of list has the form:

\begin{verbatim}
  list = \{cons (cons ... cons (false ... false))\}
\end{verbatim}
function list x, 
  x, 
endfun;

Note: The function lists terminates the constructed list always with false. It should however be noted that the last element of a list can also be a non-false atom, i.e. a fixnum, a flonum, bignum, truename or even a vector. Functions which are supposed to iterate over a list should check for the pair predicate, before taking the rest of the list. We are currently experimenting with a true-list implementation, which always terminates a list with false. This has the advantage of a cheaper end-of-list test.

oblist () returns the list of all names in the system. These names are read by the systems reader routines or produced by the string functions. A hashing function is used for fast access when the name is given as a string. Unnecessary names are automatically removed from the oblist by the garbage collector.

function oblist (). 
  <a list of all names in the system> 
endfun;

II.3.1.2 Composite constructor functions

The following general purpose constructor functions are defined in terms of constructor functions and other MoDL functions, such as recognizers and evaluation functions, described in corresponding chapters of this manual.

'op (x1, x2, ... xn) is equivalent with:

cons ('op, list (x1, x2, ... xn)).

    subroutine ' x, 
      cons (first (x), evallist (rest (x))), 
    endfun;
function evallist (x),
case
  pair (x)
    {cons (eval (first (x)), evallist (rest (x)))},
default
    {x},
endcase,
endfun;
1 '\+ 2';   -> 1 + 2

a: 2$ b: 4$

a '\+ b';   -> 2 + 4
'a '+'b';   -> a + b
'(a + b)';  -> a + b

append (11, 12... In) returns a single list which is formed from the individual lists 11, 12,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append ('(a b c), '(e f g), '(x y z))&
- > (a b c e f g x y z)

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;

append (1, 2, 3... In) returns a single list which is formed from the individual lists 1, 2, 3,... in through concatenation of the individual lists:

function append list,
  <copy all individual lists in the argument list
  and extend the last element with a pointer to the next list, until the last list can be appended directly, and return the concatenated list.>
endfun;
II.3.2

II.3.2.1 Primitive selector functions

`first` (pair) is used to select the first-cell of a pair. The function forces a break if the argument is not a pair.

\[
\text{function } \text{first} \ (x), \\
\text{case} \\
\quad \text{pair} \ (x) \ \{ \text{the content of the first-cell of } x \}, \\
\quad \text{default} \ \{ \text{break (list ('first, x))} \}, \\
\text{endcase}, \\
\text{endfun;}
\]

\[
\text{first} \ ('(x \ y \ z)) & \rightarrow z \\
\text{first} \ ('x) & \rightarrow \text{break}; \ \text{first} \ (z)
\]

`rest` (pair) is used to select the rest-cell of a pair. The function forces a break if the argument is not a pair.

\[
\text{function } \text{rest} \ (x), \\
\text{case} \\
\quad \text{pair} \ (x) \ \{ \text{the content of the rest-cell of } x \}, \\
\quad \text{default} \ \{ \text{break (list ('first, x))} \}, \\
\text{endcase}, \\
\text{endfun;}
\]

\[
\text{rest} \ ('(x \ y \ z)) & \rightarrow (y \ z) \\
\text{rest} \ ('x) & \rightarrow \text{break}; \ \text{rest} \ (x)
\]

`second` (list) is used to select the first-cell of the rest-cell of a MoDL data structure, i.e. the second element of a list. The functions `first` or `rest` can force a break if the <list> does not have a second element.

\[
\text{function } \text{second} \ (x), \\
\quad \text{first} \ (\text{rest} \ (x)), \\
\text{endfun;}
\]

\[
\text{second} \ ('(x \ y \ z)) & \rightarrow y \\
\text{second} \ ('x) & \rightarrow \text{break}; \ \text{rest} \ (x)
\]

`third` (list) is similar to `second` (list), except that it takes the third element of a list data structure.
function third (x),
    first (rest (rest (x))),
endfun;

third ('(x y z))& -> z

fourth (list) is similar to second (list), except that it takes the
fourth element of a list data structure.

function fourth (x),
    first (rest (rest (rest (x)))).
endfun;

gfourth ('(x y z))& -> false

Selectors are generalized to any depth:

function xxx...rest (x),
    xfun (xfun (xfun ... rest (x)));
endfun;

function xxx...first (x),
    xfun (xfun (xfun ... first (x)));
endfun;

Where: x represents either f or r and
      xfun represents either first or rest
      respectively.

Principle of implementation:

When eval encounters the undefined function say rfirst, it creates
dynamically the function:

function rfirst (x),
    rest (first (x)).
endfun;

The same is true for other generalized first rest combinations.
Note that first is identical to second and frest is identical to
third.

II.3.2.2 Composite selector functions

assoc (x, alist) performs a linear search of the association list,
<alist>. The function returns the element of the list which has a
first-cell which is "=" to <x>.

assq (x, alist) performs a linear search of the association list,
<alist>. The function returns the element of the list which has a first-cell which is "==" to <x>.

```lisp
function assoc (x, alist),
    case
        pair (alist)
            case
                pair (first (alist))
                    case
                        first (alist) == x
                            {first (alist)},
                        default
                            {assoc (x, rest (alist))},
                    endcase,
                default
                {assoc (x, rest (alist))},
            endcase,
        default {false},
    endcase,
endfun;
```

The function assq uses "==" where assoc uses the more general "=" for comparison. The functions are otherwise identical.

```lisp
timezones: ' ((usa 4 8) (england 12) (holland 13) (ussr 15 1))$
```  
\[\text{assoc ('england, timezones)} \rightarrow \text{(england 12)}\]
\[\text{assq ('ussr, timezones)} \rightarrow \text{(ussr 15 1)}\]

Note: assq can always be used instead of assoc if the first argument is a truename or an integer.

\[\text{nthnode (list, n), returns false if <list> does not have <n> nodes. Otherwise the function returns the remaining list, beginning with <n>th node of the list.}\]

```lisp
function nthnode (list, n),
    case
        n < 1
            {false},
        n = 1
            {list},
        pair (list)
            {nthnode (rest (list), n-1)},
        default
            {false},
    endcase,
endfun$
```

page II-3-6
The function nthnode can be used to manipulate lists as if they were vectors, this feature is not too important for the system, as MoDL supports vectors directly.

```
rowvec: '(a b c d e f g)& -> (a b c d e f g)
% Access to the n-th element:
first (nthnode (rowvec, 3))& -> c
% Modification of the n-th element:
replacefirst (nthnode (rowvec, 3), 'x)& -> (x d e f g)
rowvec& -> (a b x d e f g)
% Elimination of the n-th element:
nth: nthnode (rowvec, 3)& -> (x d e f g)
replacerest (nth, rest (nth))& -> (d e f g)
rowvec& -> (a b d e f g)
```

Note: This last operation, if performed on a true vector, requires that all elements of index > n are moved one position back. It is for this reason that lists can be equally efficient as true vectors for some algorithms, especially if the access is restricted to the first few elements.

II.3.3
II.3.3.1 Primitive modifier functions

`replacefirst` (pair, x) is used to overwrite the first-cell of a pair with the value x. The function forces a break if the argument is not a pair.
function replacefirst (x, y),
case
    pair (x)
    {< overwrite the first-cell of x with y>, x},
default
    {break (list ('replacefirst, x, y))},
endcase,
endfun;

replacefirst ('(x y z), 'a)& -> (a y z)

replacefirst ('x, 'a)& -> break: replacefirst (x, a)

replacefirst (pair, x) is used to overwrite the first-cell of a pair with the value x. The function forces a break if the argument is not a pair.

function replacerest (x, y),
case
    pair (x)
    {< overwrite the rest-cell of x with y>, x},
default
    {break (list ('replacerest, x, y))},
endcase,
endfun;

replacerest ('(x y z), '(b c))& -> (x b c)

replacerest ('x, 'a)& -> break: replacerest (x, a)

Note: The modifier functions have a side effect:

Let f be '(x y z) and g be f, i.e. f and g are identical data structures.

Then replacefirst (f, 'a) gives for f -> (a y z) and at the same time for g -> (a y z) too, in spite of the fact that g has not been modified explicitly. This effect is frequently undesired. A considerably better programming style can be developed at a slightly higher cost, if non-modifying functions are used, which partially copy list-structures.
11.3.3.2 Composite modifier functions

inplacereverse (list) uses the pair-cells of the original list to construct an inplacereverse of the original list. The inplacereverse is returned.

\[
\text{function}\ \text{inplacereverse}\ (\text{list}).
\]

\[
\text{local}\ (z, y),\ 
\text{loop}\ 
\text{exitif}\ (\text{not}\ (\text{pair}\ (\text{lst})))\ \{y\},
\text{x: rest}\ (\text{lst}),
\text{replacerest}\ (\text{lst}, y),
\text{y: lst, lst: z},
\text{endloop},
\text{y}
\text{endfun};
\]

\[
a: '(x y z)& \rightarrow (z y x)
\]
\[
a: \text{inplacereverse}\ (a)& \rightarrow (z y x)
\]
\[
b: \text{inplacereverse}\ (a)& \rightarrow (z y z)
\]
\[
a& \rightarrow (z)
\]

The examples show the side effect of the function inplacereverse. It is a general advice to invoke the function inplacereverse, which is substantially faster than reverse, only if the list to be reversed is used in a unique way.
II.4 Extraction of Logical Values

II.4.1 Comparator functions

\( x == y \) returns true if \( x \) and \( y \) are identical, otherwise false.

\[
\text{function } == (x, y),
\]
\[
\text{case}
\]
\[
<z \text{ is identical to } y > \{ \text{true} \},
\]
\[
\text{default} \quad \{ \text{false} \},
\]
\[
\text{endcase},
\]
\[
\text{endfun};
\]

\( x = y \) returns true if \( x \) and \( y \) are isomorphic data structures, otherwise false.

\[
\text{function } = (x, y),
\]
\[
\text{case}
\]
\[
\text{integer}(x) \text{ and integer}(y) \quad \{ x == y \},
\]
\[
\text{number}(x) \text{ and number}(y) \quad \{ \text{float}(x) == \text{float}(y) \},
\]
\[
\text{pair}(x) \text{ and pair}(y) \quad \{ (\text{first}(x) = \text{first}(y)) \text{ and } (\text{rest}(x) = \text{rest}(y)) \},
\]
\[
\text{default} \quad \{ x == y \},
\]
\[
\text{endcase},
\]
\[
\text{endfun};
\]

The \( = \) operator is exact if both operands are integers. Non-integer number combinations convert their operands to float. Numerical rounding effects may in this cause the test for equality to be unreliable. The \( = \) operation searches its operands recursively for isomorphic structures if the arguments are pairs. All remaining cases are tested for identical arguments.

The following table gives a survey of the behaviour of \( == \) and \( = \) on different data types:
MoDL Reference Manual (II)  

Extraction of Logical Values

<table>
<thead>
<tr>
<th>type</th>
<th>example</th>
<th>==</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>pair</td>
<td>'(x y) = (x y)</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>fixnum</td>
<td>123 = 123</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>bignum</td>
<td>1234567890 = 1234567890</td>
<td>1)</td>
<td>true</td>
</tr>
<tr>
<td>flonum</td>
<td>3.14 = 3.14</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>truename</td>
<td>'cat = 'cat</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>truename</td>
<td>'cat = 'fish</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>vector</td>
<td>'no action</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1) result is generally false as pairs occupy generally different nodes. The same holds for bignums which are internally stored as vectors of fixnums.

2) results depend on arithmetic rounding.

n1 < n2 If n1 and n2 are integers then n1 < n2 returns true if <n1> is smaller than <n2>. Non-integer number combinations convert their operands to float before making the comparison. A numerical error is raised if one of the operands is non-numerical.

```scheme
function < (x, y),
case
  integer (x) and integer (y)
  |z < y|;
  number (x) and number (y)
  |float (x) < float (y)|,
  default
  |break (list ('<, x, y))|
endcase,
endfun;
```

n1 > n2 If n1 and n2 are numbers then n1 > n2 returns true if <n2> is < than <n1>, else false is returned. Note that all magnitude comparisons are mapped to <. This implies that argument errors of x > y will show up as argument errors of y < x.

```scheme
function > (x, y),
y < x,
endfun;
```

n1 >= n2 If n1 and n2 are numbers then n1 > n2 returns false if <n1> is < than <n2>, else true is returned. Note that all magnitude comparisons are mapped to <. This implies that argument errors of x >= y will show up as argument errors of (not)
Extraction of Logical Values

\[ x < y. \]

\[ function \geq (x, y), \]
\[ \quad not \ (x < y), \]
\[ endfun; \]

\[ n1 \leq n2 \text{ if } n1 \text{ and } n2 \text{ are numbers then } n1 > n2 \text{ returns false if } <n2> \text{ is } < \text{ than } <n1>, \text{ else true is returned. Note that all magnitude comparisons are mapped to } <. \text{ This implies that argument errors of } x \geq y \text{ will show up as argument errors of } (not) \ y < x. \]

\[ function \leq (x, y), \]
\[ \quad not \ (y < x), \]
\[ endfun; \]

II.4.2 Logical operators and functions

The logical operations defined in this chapter permit Boolean operations on arbitrary arguments. Any nonfalse truth value is considered to be logically true.

\[ not \ (x) \text{ returns true if } <x> \text{ is false, otherwise false. Note: not is identical to empty. Issues of program readability suggest the use of not for Boolean operations while empty is recommended to test an empty list.} \]

\[ function not \ (x), \]
\[ \quad case \]
\[ \quad \quad x == false \ {true}, \]
\[ \quad \quad default \ {false}, \]
\[ \quad endcase, \]
\[ endfun; \]

\[ x1 \text{ and } x2 \text{ and ... and } xn, \text{ returns true if no arguments are supplied, otherwise it evaluates its arguments from left to right until: either any of the arguments evaluates to false, in which case false is returned or all arguments are non-false in which case the value of } <xn> \text{ is returned.} \]
subroutine and x,
    local (xn).
    case
        empty (x)
            {true},
        default
            {loop
                exitif not (pair (x)) {zn},
                exitif empty (xn: eval (pop (x))) {false},
            endloop},
    endcase,
endsub;

x1 or x2 or ... or xn, returns false if no arguments are supplied, otherwise it evaluates its arguments from left to right until: either a non-false value is found, which is returned as the value of the function, or all arguments are false in which case false is returned.

subroutine or x,
    local (zn).
    case
        empty (x)
            {false},
        default
            {loop
                exitif not (pair (x)) {false},
                exitif eval (pop (x)) {true},
            endloop},
    endcase,
endsub;
II.5 Operations on Names

MoDL represents a name as a 5-tuple, consisting of a value field, a property field, a function definition field, a string field and a fnype field. The internal representation of a name is:

<table>
<thead>
<tr>
<th>field</th>
<th>&quot;initial value&quot;</th>
<th>initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>&quot;&quot;</td>
<td>unbound</td>
</tr>
<tr>
<td>value</td>
<td>&quot;initial&quot;</td>
<td>value&quot;</td>
</tr>
<tr>
<td>value</td>
<td>any other value</td>
<td>eval (&quot;initial value&quot;)</td>
</tr>
<tr>
<td>property</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>function</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>string</td>
<td>print-name</td>
<td>false</td>
</tr>
<tr>
<td>fnype</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

The value field can be initialized with "any" predefined value, depending on the value of the MoDL-name "initial value", as is shown in the table. The print-name is the ascii representation of the name of the symbol it can contain arbitrary characters such as the space in "initial value" (see also the description of the readlist function in section 8.3.).

II.5.1 Assignment functions

The assignment statement gives the programmer an opportunity to bind an expression to a name in the system. In applicative or functional programming styles one tries to avoid assignments through the utilisation of recursion and functional composition, something for which MoDL is very well adapted already. Assignments do cause so-called side-effects which can interfere with the order of program execution, making programs harder to understand and debug. This situation is less bad as it seems, as MoDL offers local variables, which shield the side-effect to the body of a function, subroutine or macro.

set (truename, x) replaces the value of <truename> by the expression <x> and returns <x>. 
function set (truename, x),
case
  truename (x) and not (protected (x))
  {[assign x to truename]},
endcase,
x, endfun;

fish: 'goldfish;   -> goldfish
set (fish, 'red);  -> red
fish;             -> red
fish;             -> goldfish

truename : x replaces the value of <truename> by the expression <x> and returns <x>. In contrast with set there is no evaluation of the left operand. The MoDL assignment-operator ":=" is equivalent to " := " in ALGOL and PASCAL and " = " in FORTRAN and BASIC.

subroutine : (x, y),
  set (x, eval (y)),
endsub;

fish: 'goldfish;   -> goldfish
fish: 'red;        -> red
goldfish;         -> unbound
fish;             -> red

setunbound (truename) replaces the value of <truename> by the unbound value and returns <truename>.

function setunbound(x),
case
  truename (x) and not (protected (x))
  {[assign unbound to name]},
endcase,
x, endfun;

bound (truename), returns true if an assignment of a value has been made, i.e. if the value of truename is bound.
function bound (x),
case
  truename (x) and <x has a value> [true],
  default [false],
endcase,
endfun;

bound ('goldfish'); -> false
bound ('fish'); -> true

pop (truename) Operates on the list assigned to <truename> as if it was a stack, i.e. pop (<truename>) returns the first-element of the stack assigned to name and assigns to name the rest of the stack.

subroutine pop (x),
  local (stack),
  stack: eval (x),
  case
    pair (stack)
      {set (z, rest (stack)), first (stack)},
    default
      {false},
  endcase,
endsub;

stack: '(x y z)&
pop (stack)& -> (x y z)
stack& -> x
pop (stack)& -> (y z)
stack& -> y
pop (stack)& -> (z)
stack& -> z
pop (stack)& -> false
stack& -> false

push (x, truename) Operates on the list assigned to <truename> as if it was a stack, i.e. push (x, truename) inserts x as first element of the stack and returns the entire stack.
subroutine push (x, y),
  set (y, cons (eval (x), eval (y))),
endsub;

stack : ' (z)&
  push ('y, stack)&
  push ('x, stack)&
stack&
  -> (z)
  -> (y z)
  -> (x y z)

setprotect (true), protects <true> from any value change through assignment.

The following names are protected by default:

false
true
stdin
stdout
stderr

II.5.2 Property lists and related functions

The property functions put and get are used to maintain properties associated with names. Two helper functions, setplist and plist are used to access the property list of the name given. The property list itself is organized as an association list (see the description of assoc in section 2.20). The property lists provide the MoDL user with a powerful extensible database, which encourages the development of efficient and modular software packages.

put (name, p, v), associates the property <p> with value <v> to <name>. Any previous value associated with the same property-value pair is overruled. The value returned by put is <v>.

function put (x, y, z),
  local (match),
  case
    match : assoc (y, plist (x))
    | replacerest (match, z)|,
  default
    | setplist (x, cons (cons (y, z),
        plist (x)))|,
  endcase,
  z, 
endfun;

Below is a database of normalized timezones for different countries:

page II-5-4
MoDL Reference Manual (II) Operations on Names

```plaintext
put ('usa, 'timezone, '(4 8))& -> (4 8)
put ('england, 'timezone, '(12))& -> (12)
put ('holland, 'timezone, '(13))& -> (13)
put ('ussr, 'timezone, '(15 1))& -> (15 1)
put ('japan, 'timezone, '(21))& -> (21)
```

get (name, p), returns the property <p> associated with <name>.

```plaintext
function get (x, y),
  local (match),
  case
    match: assoc (y, plist (x)) {rest (match)},
    default {false},
  endcase,
endfun;
```

```plaintext
get ('usa, 'timezone)& -> (4 8)
get ('england, 'timezone)& -> (12)
```

remprop (name, p), returns the property <p> associated with <name>, and removes this property from the property list of <name>.

```plaintext
function remprop (x, y),
  local (z),
  z: get (x, y),
  <remove y from the property list of x>,
  z,
endfun;
```

Primitives to manipulate the property list of a name, which are normally used by the MoDL system only, are:

```plaintext
setplist (name, v), assigns the value <v> to the property list of <name>.
plist (name), the property list of <name>.
```

These functions should be used with utmost care, as the property lists of MoDL names are assumed to be properly formed plists, which can contain valuable information for the proper handling of the MoDL syntax by the read and print functions and similar other MoDL features.
11.5.3 Function definition

def (true name, type, formalargs, localvars, exp1, exp2, ... expn), is used by the MoDL parser to define a function of any <type> with name <true name>.

subroutine def x,
    defhelper (pop (x),
                first (x),
                second (x),
                third (x),
                x),
endsub;

function defhelper (true name, type, formalargs, localvars, body),
    testnames (formalargs),
    testnames (localvars),
    selfundef (true name, body),
    name,
endfun;

Where the function testnames is used to check the list of formal arguments and local variables for correct composition, i.e. both should be a true-list of truenames. Example:

eval ('(def testfn FUN2 (x y) false
           (print x y newln) x)) & -> testfn

is equivalent with:

function testfn (x, y),
    print (x, y, newln),
    x,
endfun;

getdef (true name), returns the value of the function-field.
function getdef (x),
case
  truename (x) |<function-field of name>|,
  default [false],
endcase,
endfun;

getdef ('testfn')
-> (FUN2 (x y) false
  (print x) (print y) (newln) x))
if testfn would have been compiled then getdef would have returned:
-> (FUN2)

Below is a table which shows the syntax used in MoDL to define functions of various fntypes:

<table>
<thead>
<tr>
<th>Function</th>
<th>Subroutine</th>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>function f(x)</td>
<td>subroutine f(x)</td>
<td>macro f(x)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>fntype: FUN</td>
<td>fntype: SUB</td>
<td>fntype: MAC</td>
</tr>
<tr>
<td>function f(x, y, z)</td>
<td>subroutine f(x, y, z)</td>
<td>macro f(x, y, z)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>fntype: FUN3</td>
<td>fntype: SUB3</td>
<td>fntype: MAC3</td>
</tr>
</tbody>
</table>

The numerical postfix denotes the number of formal arguments which the specific function takes.

II.5.4 String functions

explode (name), returns a list of names whose single character print-names correspond to the characters in in the print-name of <name>. 
**function explode (x),
   case
     name (x) {<a list of names corresponding
      with the characters in x>},
     default {false},
   endcase,
endfun;

explode ('xyz')& \rightarrow (x y z)
explode ('"a b")& \rightarrow (a " b)
explode ('"""')& \rightarrow ()
explode ('()')& \rightarrow (false)

**implode (list), returns the name whose print-name is composed of the single character print-names in the given list.

**function implode (x),
   case
     pair (x) and name (first (x))
      {<the name whose print-name consists of the print-name of first (x) concatenated with implode (rest (x))>},
     default {"""},
   endcase,
endfun$

implode ('(x y z))& \rightarrow xyz
implode ('(a " b))& \rightarrow "a b"
implode ('()')& \rightarrow ""
implode ('(false))& \rightarrow ()

**explodenum (name), returns a list of numbers corresponding with the ordinality of the characters in the print-name of <name>.

page II-5-8

6.83
function explodenum (x),
    case
        name (x) {<a list of numbers corresponding
                    with the ordinality of the
                    characters in x>},
        default {false},
    endcase,
endfun;

explodenum ("xyz")& -> (120 121 122)
explodenum ("a b")& -> (97 32 98)
explodenum ("...")& -> ()
explodenum ("()")& -> (102 97 106 115 101)

The function explodenum has the advantage over explode that it
avoids the creation objects on the oblist which correspond to the
characters in the print name. It should be noted however that this
difference is not too important as the MoDL garbage collector
reclaims all space used by unreferenced names which are in their
initial state (i.e. which are reconstructed in the same state if they
are re-introduced into the system).

concat (list), returns the name whose print-name is composed of
the single character print-names in the argument-list.

    function concat x,
        implode (x),
    endfun

concat ('x', 'y', 'z')& -> xyz
concat ('a', 'b')& -> "a b"

gensym (name), returns the name composed of the first character
of <name>, followed by 6 characters which represent the integer
value of name. The integer value of <name> is incremented by
gensym.

gensym: 0$ -> 0
othersym: 7$ -> 7
gensym (gensym)& -> g000001
gensym (gensym)& -> g000002
gensym (othersym)& -> o000008
gensym (gensym)& -> g000003

ascii (n), returns the name whose print-name is composed of the
single character from the ascii alphabet with ordinality n.
function ascii (n),
case
  fixnum (n) and (0 <= n) and (n <= 255)
  {<the name with print-name ascii (n)>},
  default {""},
endcase,
endfun;

ascii (32)& -> " "

ascii (13)& -> <the ascii carriage-return character>
ord (char), returns integer which corresponds with the ordinality of the first character in the print-name of <char>.

function ord (x),
case
  x == "" {-1},
  default {<the ordinality of the first ascii character of x>},
endcase,
endfun;

ord (""")& -> 32
ord ("M")& -> 77
ord ("MoDL")& -> 77
II.6 Vector Operations

MoDL vectors can be used to speed-up the access to the data structures in an algorithm. Any MoDL data structure can contain an arbitrary number of vectors. The fields of the vector can reference arbitrary MoDL data structures, including vectors. The basic recognizer functions include a function which treats vectors:

\[ \text{vector}(x) \text{ returns } \begin{cases} \text{true} & \text{if } \langle x \rangle \text{ is a vector,} \\ \text{false} & \text{otherwise.} \end{cases} \]

function vector (x),
  case
    \langle x \rangle \text{ is a vector} \rightarrow \text{true},
    \text{default} \rightarrow \text{false},
  endcase,
endfun;

Vectors are accessed with an index which may run from 0 to n, i.e. any vector with an allocated length of n has n+1 elements. Vectors can either be read between double angle brackets as indicated, or allocated by the MoDL function allocvec:

allocvec (n, v), allocates a vector of \langle n+1 \rangle elements, ranging from 0 up to and including n, with initial values \langle v \rangle. The vector itself is returned.

function allocvec (n, v),
  \langle allocate a vector with \langle n \rangle elements with value \langle v \rangle \rangle,
endfun;

vec: allocvec (3, false); -> \langle false, false, false, false \rangle

Note: The maximum length of a vector is restricted to 2040 elements for efficiency reasons. Longer vectors can be easily implemented in MoDL with a set of user written functions: alloclongvec, setlongvec and getlongvec.

setvec (vec, n, v), assigns the value \langle v \rangle to element \langle n \rangle of \langle vec \rangle and returns \langle v \rangle.
function setvec (vec, n, v),
    <assign the value <v> to element <n> of <vec>>,
    v,
endfun;

setvec (vec, 2, 'a);            -> a
vec;                          -> <false, false, a, false>

getvec (vec, n), returns element <n> of <vec>.

function getvec (vec, n),
    <element <n> of <vec>>,
endfun;

getvec (vec, 1);             -> false
getvec (vec, 2);             -> a
II.7 Arithmetic Operations

\( n_1 + n_2 + \ldots + n_n, \) returns the sum of the numbers \( n_1, n_2, \ldots, n_n. \) The '+' operator can be used in its unary form also like in +7 or +x, however the MoDL parser removes such redundant occurrences of this operator.

\[
function + (x_1, x_2),
\]
\[
case
number (x_1) and number (x_2),
{<the sum of x_1 and x_2>},
default
{break (list (+, x_1, x_2))},
endcase,
endfun;
\]

\( n_1 - n_2, \) returns the difference between the numbers \( n_1 \) and \( n_2. \) The MoDL read function converts -7 and -x to 0-7 and 0-x respectively.

\[
function - (x_1, x_2),
\]
\[
case
empty (x_2)
{0 - x_1},
number (x_1) and number (x_2),
{<the difference between x_1 and x_2>},
default
{break (list (-, x_1, x_2))},
endcase,
endfun;
\]

\( n_1 \cdot n_2 \cdot \ldots \cdot n_n, \) returns the product of the numbers \( n_1, n_2, \ldots, n_n. \)

\[
function \cdot (x_1, x_2),
\]
\[
case
number (x_1) and number (x_2),
{<the product of x_1 and x_2>},
default
{break (list (*, x_1, x_2))},
endcase,
endfun;
\]

\( \text{divide} \ (n_1, n_2), \) returns a two-element list comprising of the quotient and rest of the division of \( n_1 \) by \( n_2. \)

\( n_1 \div n_2, \) returns the quotient of the division of \( n_1 \) by \( n_2. \)

\( \text{mod} \ (n_1, n_2), \) returns the rest of division of \( n_1 \) by \( n_2. \)
n1 ~ n2, returns n1 to the power n2.

n1 << n2 returns the integer n1 left-shifted over n2 bits.

n1 >> n2 returns the integer n1 right-shifted over n2 bits.

abs(n), returns -<n> if <n> is negative and <n> otherwise.

    function abs (n),
        case n<0
            [-n],
        default  { n},
        endcase,
    endfun;

max (n1, n2, ...nn), returns the largest of the numbers <n1>, <n2>, ... <nn>.

    function max x,
        local (max),
        max: pop (x),
        loop
            exitif not (pair (x)) {max},
            case max<first (x) {max: first (x)} endcase,
            z: rest (x),
        endloop,
    endfun;

min (n1, n2), returns the minimum of the numbers <n1> and <n2>.

    function min x,
        local (min),
        min: pop (x),
        loop
            exitif not (pair (x)) {min},
            case first (x)<min {min: first (x)} endcase,
            z: rest (x),
        endloop,
    endfun;

sin (n), returns a flonum representing the sine of the flonum representation of <n>.

cos (n), returns a flonum representing the cosine of the flonum representation of <n>.

tan (n), returns a flonum representing the tangent of the flonum representation of <n>.
representation of $n$.

**asin** ($n$), returns a flonum representing the arcsine of the flonum representation of $n$.

**acos** ($n$), returns a flonum representing the arccosine of the flonum representation of $n$.

**atan** ($n$), returns a flonum representing the arctangent of the flonum representation of $n$.

**log** ($n$), returns a flonum representing the natural logarithm of the flonum representation of $n$.

**nlog** ($n$, $n2$), returns a flonum representing the base-$n$ logarithm of the flonum representation of $n$.

**sqrt** ($n$), returns a flonum representing the square root of the flonum representation of $n$.
II.8 IO Functions

MoDL supports files and ports for its basic IO functions. The files are used to identify a file-object in the MoDL environment. The ports, which are refinements of a truename, are used to keep track of the current activity of the read or print function on the open file. This concept makes it possible to have multiple ports open for a single file.

<table>
<thead>
<tr>
<th>portname</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>infile(filename)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outfile(filename)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>close(portname)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdout</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stderr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;defined_port&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE II.8.1

Table II.8.1 shows how MoDL identifiers can be assigned proper port values. The first two examples show how new ports can be opened for input and output. MoDL sets the function pointer of the newly created port name in these first two cases to a function which controls all access to the named file. The close function can be used to return the port which was open when the port in the argument list of the close was opened. Assignments of system ports (stdin, stdout, stderr) result always in a proper port value, as the value of the system ports cannot be changed. The last way to change the value of a port variable is to assign the value of another variable, which itself also holds a port value.

<table>
<thead>
<tr>
<th>PORT</th>
<th>variable</th>
<th>default</th>
<th>proper value</th>
<th>used by:</th>
<th>modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>infile</td>
<td>stdin</td>
<td>port</td>
<td>reader</td>
<td>user</td>
</tr>
<tr>
<td></td>
<td>outfile</td>
<td>stdout</td>
<td>port printer</td>
<td>user</td>
<td></td>
</tr>
<tr>
<td></td>
<td>errfile</td>
<td>stderr</td>
<td>port</td>
<td>break</td>
<td>user</td>
</tr>
<tr>
<td></td>
<td>inecho</td>
<td>false</td>
<td>port/false</td>
<td>reader</td>
<td>user</td>
</tr>
<tr>
<td></td>
<td>outecho</td>
<td>false</td>
<td>port/false</td>
<td>printer</td>
<td>user</td>
</tr>
</tbody>
</table>

TABLE II.8.2

Table II.8.2 shows the most important relations between standard MoDL port-identifiers and their usage in the MoDL system. The first column shows the default settings for the MoDL port-identifiers. All MoDL port-identifiers should hold proper port values. Only the port identifiers inecho and outecho may have a value false to indicate that the system should not echo its input or output. The main usage of the MoDL port-identifiers is given in column four. All
MoDL Reference Manual (II)  

10 Functions

port-identifiers are under direct control of the user, none is changed by the MoDL system. The port identifiers inecho and outecho can be controlled by the user for various purposes.

`inecho : stdout;`

Echoes all input read to the standard output, normally the user's terminal. This setting is very useful if a file with MoDL programs is read which may still contain syntax errors.

`inecho : outfile (...);`

Echoes all input read to the specified output file. This setting is useful if one wants to keep a file with all text typed by the user.

`outecho : outfile (...);`

Echoes all input output produced by the MoDL print functions to the specified port returned by outfile.

`outecho : inecho : outfile (...);`

Echoes all input and output to a single output file. This setting is ideal to obtain a hardcopy of a complete MoDL session.

`port (x) returns true if <x> is a port, otherwise false.`

```lisp
function port (x),
case
  truename (x) and <x is a port> {true},
  default {false},
endcase,
endfun;
```

11.8.1 10port

`infile (file), opens <file> for input and returns the corresponding port. The name of the <file> and the current value of infile are memorized for the close function. The function forces a break if the <file> is not a truename, or the file cannot be opened.

page II-8-2

6.92
function infile (file),
local (port),
case
truename (file)
|case
| (port: fopen (file, 'r'))
| {put ('close, port, infile),
| port},
| default {print ('cannot open:'),
| break (list ('infile, file'))},
endcase
| default {break (list ('infile, file'))},
endcase,
endfun;

outfile (file), opens <file> for output and returns the corresponding port. The name of the <file> and the current value of outfile are memorized for the close function. The function forces a break if the <file> is not a truename, or the file cannot be opened.

function outfile (file),
local (port),
case
truename (file)
|case
| (port: fopen (file, 'w'))
| {put ('close, port, outfile),
| port},
| default {print ('cannot open:'),
| break (list ('outfile, file'))},
endcase
| default {break (list ('outfile, file'))},
endcase,
endfun;

close (port), closes an open <port>, and returns the port memorized by the function infile or outfile. The function forces a break if the <port> is not open or if any of the ports stdin, stdout, stderr is used.
function close (port),
    local (oldport),
    case
        oldport: remprop (‘close, port)
            { fclose (port),
              oldport}
        default
            {break (list (‘close, port))},
    endcase,
endfun;

charscnt (port), returns the read- or write-pointer of an open <port>. A break is forced if the file is not open.

function charcnt (port),
    case
        get (‘close, port)
            {<the value of the file-pointer>},
        default
            {break (list (‘charscnt, port))},
    endcase,
endfun;

Example: redirection of output.

function redirect_out (file),
    outfile: outfile (file),
    <Here follows the program which needs redirection of output>,
    outfile: close (outfile),
endfun;

redirect_out (“"/usr/cadmwdl/<file>“);
redirect_out (“’.../<file>“);

II.8.2 Reader

The MoDL reader can easily be configured to read according to different syntax rules. We distinguish here:

2. The token-syntax: readatom (), preadchar(), printatom (), pprintatom () .
3. The list-syntax: readlist (), preadlist (), printlist (), pprintlist () .
4. The MoDL-syntax: read (), print ()

II.8.2.1 Character IO

readchar (), reads and returns a character from the port assigned to the variable infile. A throw to the label read is made when the end of the file is encountered during reading.

preadchar (port), is identical to readchar, but reads from the specified port instead of the standard input.

The function copyfile defined below will copy a file from in to out.

```
function copyfile (in, out),
  infile: infile (in),
  outfile: outfile (out),
catch ('read,
    loop
      printatom (readchar ()),
    endloop),
  infile: close (in),
  outfile: close (out),
en dfun;
```

II.8.2.2 The reader: token-syntax

readatom (), reads and returns an atom from the port assigned to the variable infile. A throw to the label read is made when the end of the file is encountered during reading.

preadatom (port), is identical to readatom but reads from the specified port instead of the standard input.

The function dictionary defined below will collect all atoms found in the infile in the list dict.
function dictionary (in),
    local (word, dict),
    infile: infile (in),
    catch ('read,
        loop
            word: readatam ()
        case
            member (word, dict) {, push (word, dict) },
        default,
        endcase,
        endloop),
    infile: close (infile),
    dict,
endfun;

What is considered to be a word depends on the syntax rules assigned to the characters. This issue, which is highly flexible in MoDL, will be considered next.

First of all there is a possibility to include comment in a file, a single character is usually selected as a start of comment character. The reader will skip all text until the next newline after reading a comment character, i.e. a character marked with the COMMENTFUN character-read-macro. The percent character: % is used for this purpose in MoDL.

Any character preceded by a single escape character, !, i.e. marked as S_ESCAPE, will be considered to be a normal character for the reader, the escape-character itself will be ignored in this process.

Multiple characters can be escaped from the normal MoDL token syntax by including them in string quotes, also called multiple escape characters. The characters " or ], can be used for this purpose, they are marked as M_ESCAPE in the character-read-table.

The actual setting of a character from one type to another can easily be inspected with the setsyntax and getsyntax, respectively setmacro and getmacro functions. It is for instance not needed to support two M_ESCAPE characters, as the print function can use only one of them.

Tokens are being separated by white space characters, i.e. characters marked WHITESPACE in the character-read-table, or
broken on terminating characters, marked with TERMAC in the table below. What remains are characters which build up the token names, the CONSTITUENTS and the ILLEGAL characters, which issue a throw to the label read.

Seven types are associated with all characters read by the MoDL reader, these are:

<table>
<thead>
<tr>
<th>CharType</th>
<th>Character(s) initially of the given type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ILLEGAL</td>
<td>all remaining characters</td>
</tr>
<tr>
<td>2. CONSTITUENT</td>
<td>&quot;0123456789@abcdefghijklmnopqrstuvwxyz&quot;</td>
</tr>
<tr>
<td>3. WHITESPACE</td>
<td>&lt;tab&gt;, &lt;vtab&gt;, &lt;lf&gt;, &lt;cr&gt;, &quot; &quot;</td>
</tr>
<tr>
<td>4. TERMAC</td>
<td>&quot;$&amp;*(+:-[:&lt;]~`</td>
</tr>
<tr>
<td>5. NONTERMAC</td>
<td>&quot;</td>
</tr>
<tr>
<td>6. M_ESCAPE</td>
<td>&quot;</td>
</tr>
<tr>
<td>7. S_ESCAPE</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

MacroType  Character(s) initially of the given type:
1. IDENTITYFN | "$&(*+;[:<]~`|] " |
2. COMMENTFN   | "| "            |
3. NOFUNCTION  | all remaining characters                                                     |
4. MINUSFN     | -                |

The settings of character types of individual characters are kept in the reader_table of the MoDL system. The initial settings of the reader_table are given in the figure, where:

\[\begin{array}{ll}
<bs> & \text{denotes the backspace character } \quad \text{ascii (8)}.\\
<tab> & \text{the tab character } \quad \text{ascii (9)}.\\
<lf> & \text{the linefeed character } \quad \text{ascii (10)}.\\
<cr> & \text{the carriage-return character and } \quad \text{ascii (13)}.\\
\text{""} & \text{the space } \quad \text{ascii (32)}.
\end{array}\]

Single character tokens are those which are marked with TERMAC and IDENTITYFN, i.e.: $ & \ast + \; - [ ] \sim \{ \} .

Such single character tokens are always read as a single character token if they are not escaped, no matter what the surrounding characters are.

The MoDL token recognition has an additional syntax for the recognition of numbers, integers as well as floating point, which is identical to the syntax found in common programming languages. Let us, as an example, apply the dictionary function shown, on a
The reader-character-table can be altered with the getsyntax and setsyntax functions:

**getsyntax** (char), returns the value of the entry of char in the reader-character-table. The character <char> can either be specified as an integer or as the first character of a name.

```modl
function getsyntax (char),
local (n),
case
  empty (n: readerindex (char)) [false],
default ["the content of readertable (n)"],
endcase,
endfun;
```

```modl
<begin of file>

% This is a test-file to show how tokens are read...

function copyfile (in, out),
infile: infile (in),
outfile: outfile (out),
catch ('read,
  loop
    printatom (readchar ()),
  endloop),
infile: close (infile),
outfile: close (outfile),
endfun;

<end of file>
```

```modl
dictionary ("copyfile")&
  -> (endfun close endloop readchar printatom loop
      read '"' catch outfile infile "") out "." in
     "(" copyfile function)
```

The reader-character-table can be altered with the getsyntax and setsyntax functions:

**getsyntax** (char), returns the value of the entry of char in the reader-character-table. The character <char> can either be specified as an integer or as the first character of a name.
function readerindex (char),
case
  name (char)
    {char: first (explodeunm (char))},
  endcase,
case
  fixnum (char) and (0<=char) and (char<128)
    {n},
  default
    {false},
  endcase,
endfun;
sclsynlax (char, x), changes the value of the entry of <char> in the reader-character-table to <x>. The character <char> can either be specified as an integer or as the first character of a name. The return value is ...
function setsyntax (char, x),
local (n),
case
  empty (n: readerindex (char))
    {false},
  member (x, 
   '("ILLEGAL" "CONSTITUENT"
     "WHITESPACE" "TERMAC" "NONTERMAC"
     "M_ESCAPE" "S_ESCAPE"'))
    {<assign <x> to readertable (n)>},
  default
    {false},
  endcase,
endfun;
The reader-macro-table can be altered with the getmacro and setmacro functions:

getmacro (char), returns the value of the entry for char in the reader-macro-table. The character <char> can either be specified as an integer or as the first character of a name.
function getmacro (char),
    local (n),
    case
      empty (n: readerindex (char))
      {false},
      default
      {the content of macro-table (n)},
    endcase,
endfun;

setmacro (char, x), changes the value of the entry of char in the
reader-macro-table to <x>. The character char can either be
specified as an integer or as the first character of a name. The
return value is ...

function setmacro (char, x),
    local (n),
    case
      empty (n: readerindex (char))
      {false},
      member (x,
      {"IDENTITYFN" "COMMENTFN" 
      "NOFUNCTION"})
      {assign <x> to macro-table (n)},
      default
      {false},
    endcase,
endfun;

11.8.2.3 The reader: list-syntax
The MoDL list level is LISP-alike, i.e. constants and lists can be
processed in a way very similar to several LISP implementations.

readlist (), reads a complete list data-item from the port assigned
to the variable infile, returning the equivalent data structure. A
throw ("read, false), is performed if the end of the file is
encountered. A break is made if an error occurs. The function
readlist will either read an atom following the token syntax of
readatom, or a (nested) linked list. The last element of a list has in
general a false pointer, unless there is a dotted pair included.

Lists have the general form: (x1 x1 ... xn). Any element xi can be a
list again, or a dotted pair of the form: x1. x2

preadlist (port), is identical to readlist, but reads from the
specified port instead of the standard input.

```
function readlist (),
  <read a data-item in list-notation
   from the infile returning
    its list-notation>,
endfun;
```

The following example shows readlist in action reading a file:

```
function readfile (in),
  infile: infile (in);
  catch ('read,
    loop
      printlist (readlist (), newline),
    endloop),
  infile: close (infile),
endfun;
readfile ('copyfile')
```

```
function copyfile
  (in ".", out)
  "",
  infile
  (in)
  "",
and so on until end-of-file.
```

11.8.2.4 The reader: MoDL-syntax

The MoDL language level assumes the use of the MoDL syntax, which is read by the function read.

```
read (), reads a complete MoDL language construct from the infile,
returning the equivalent data structure. A throw ('read, false), is
performed if the end of the file is encountered. A break is made if
an error occurs. The MoDL syntax is defined elsewhere in this
manual.
```

```
function read ()
  <read a MoDL language construct returning
   its list representation>,
endfun;
```
II.8.3 Output

printlist (x1, x2, ... xn), prints x1, x2, ... xn in list-format, surrounding escaped names with escape characters where needed and returns xn.

```
function printlist x,
   foreach el in x do
      printlistl (el, false)
   endfor,
endfun;
```

printatom (x1, x2, ... xn), prints x1, x2, ... xn in list-format and printing nonprintable names without escape characters. The function returns xn.

```
function printatom x,
   foreach el in x do
      printlistl (el, true)
   endfor,
endfun;
```
function printlist1 (x, flag),
local (y),
case
  empty (x)
    | printstring ('()'),
  fixnum (x)
    | printfomat ('%ld', C_Fixnum (x)),
      if x == 'newln
        | newln (),
  ftonum (x)
    | printfomat ('%g', C_Flonum (x)),
  bignum (x)
    | printbignum (x),
  true-name (x)
    | case
      flag & printstring (x),
    default & printname (x),
  endcase,
  pair (x)
    | printstring ('('),
  loop
    | printlist (pop (x)),
    exitif empty (x) {},
    exitif atom (x)
      | printstring (''),
    printlist1 (x, flag),
  endloop,
    printf (''),
  vector (x)  | printvec (x),
  case
    y: get ('close, x)
      | printlist1 (first (y), flag),
    endcase,
  endcase,
  x,
endfun;
function printbignum (x),
   <print x in integer format>
endfun;

function printpname (x),
   <print <x> with single_escape and multiple_escape characters where needed>
endfun;

newln (), prints the newln character and returns it.

function newln (),
   printatmn (newln),
endfun;

newln: ascii (10)$

pprintlist (port, x1, x2, ... xn), prints x1, x2, ... xn in list-format on
the specified port, surrounding escaped names with escape
characters where needed and returns xn.

ppprintatom (port, x1, x2, ... xn), prints x1, x2, ... xn in list-format
on the specified port, printing nonprintable names without escape
characters. The function returns xn.

pnewln (port), prints the newln character on the specified port
and returns it.

II.8.4 The loader

The function load acts as an acceptor, it reads a file taking a
certain input-syntax into account. The function transload acts as
a transducer, it reads a file taking an input-syntax into account
and produces at the same time an output-file with a different
syntax.

load (file, in_syntax), loads the specified <file> with the specified
<in_syntax>, returning false. The input filename is:
<file>.<in_syntax>, i.e. <in_syntax> is used as the extension of
the filename.
function load (fname, in).
case
    member (in, 'modl list 0))
        {infile: infile (concat (fname, dot, in)),
         driver (in, 'modl),
         infile: close (infile)},
    default
        {break (list ('load, fname, in))},
endcase,
endfun;

transload (file, in_syntax, out_syntax), loads the specified <file> with the specified <in_syntax> and produces at the same time an output-file with the specified <out_syntax>. The function transload returns false. The input filename is: <file><in_syntax>, i.e. <in_syntax> is used as the extension of the filename. The output filename is: <file><out_syntax>, i.e. <out_syntax> is used as the extension of the filename.

function transload (fname, in, out),
case
    member (out, get ('transload, in))
        {infile: infile (concat (fname, dot, in)),
         outfile: outfile (concat (fname, dot, out)),
         driver (in, out),
         infile: close (infile),
         outfile: close (outfile)},
    default
        {break (list ('transload, fname, in, out))},
endcase,
endfun;

The allowed <in_syntax>, <out_syntax> combinations are defined as:

put ('transload, 'modl, '(list 0))$
put ('transload, 'list, '(modl 0))$
II.9 Program Execution Primitives

II.9.1 Evaluation

'x returns <x> without evaluation.

```
subroutine 'x,
  x,
endsub;
```

`x (x y z)`

```
'x& => x

x => (x y z)
```

```
eval (x), evaluates x and returns the evaluated result. The table below shows the result of the evaluation process for all MoDL data types:
```

<table>
<thead>
<tr>
<th>type</th>
<th>argument of eval</th>
<th>value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>fixnum</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>pair</td>
<td>&quot;('+' 2 3)&quot;</td>
<td>5</td>
</tr>
<tr>
<td>bignum</td>
<td>1234567890</td>
<td>1234567890</td>
</tr>
<tr>
<td>flonum</td>
<td>3.14</td>
<td>3.14</td>
</tr>
<tr>
<td>truename</td>
<td>'cat'</td>
<td>'cat'</td>
</tr>
</tbody>
</table>

Note: eval is actually called twice on the data objects: "(\'+\' 2 3)" and object. This notation shows for the list "(\'+\' 2 3)" that the operator is always in its first position, irrespective of its appearance in the MoDL surface language. It shows for the truename cat that the value of cat is not meant.
function eval (x).

  case
    empty (x) or fixnum (x)
      {x},
    pair (x)
      {apply (first (x), rest (x))},
    name (x)
      {case
          bound (x) {value (x)},
        default {break (list ('eval, x))},
        endcase},
    default {x},
  endcase,
endfun;

Eval has been written in the C language directly for efficiency reasons, as it is used to execute MoDL programs interactively. Eval can execute compiled MoDL code too. It delegates this task however to apply, a general function which can execute compiled and interpretative MoDL code.

apply (fn, args), executes the function <fn> with as actual arguments <args>, in accordance to the discipline of the function <fn>.

function apply (fn, args),
        (discipline (fn)) (fn, args),
endfun;

The notation (f) (x, y), which is not part of the MoDL syntax, is used here to denote the application of a compiled function f on the arguments (x, y).

II.9.1.1 Evaluation disciplines

There are three basic function types: functions, subroutines and macros. All these function types can either be invoked with a variable number of arguments or with a fixed number of arguments, which may range from 1 to 10. The syntax and corresponding types are already given in section 5.3. under getdef. The most widely used are functions with a fixed number of arguments:

page II-9-2
function fun (x1, x2, ..., xn),
    <function body>,
endfun;

apply ('fun, list (a1, a2, ..., an)),
apply will handle this case to the corresponding discipline handler:
function CompiledFUN1..10 (fn, args),
    (fn) (eval (a1), eval (a2), ..., eval (an)),
endfun;
which evaluates the actual arguments of the function and then
invokes the function, returning the function value. The operation
for an interpretative function is similar. This will however be
discussed later on.

x: 'a&
y: 'b&
apply ('cons, '(x y))&  -> a & b
which is equivalent with:
    cons (x, y),
Less frequently used are functions with a variable number of
arguments:
function fun x,
    <function body>,
endfun;
apply ('fun, list (a1, a2, ..., an)),
apply will handle this case to the corresponding discipline handler:
function CompiledFUN (fn, args),
    (fn) (foreach arg in args collect
eval (arg) endfor),
endfun;
which evaluates the actual arguments of the function and then
invokes the function, returning the function value. The operation
for an interpretative function is similar. This will however be
discussed later on.
Subroutines allow the programmer to get the actual arguments passed as they are, i.e. without evaluation. There are subroutines with a fixed number of arguments:

```plaintext
subroutine sub (x1, x2, ... xn),
  <subroutine body>,
endsub;

apply ('sub, list (a1, a2, ... an)),
```

Apply will handle this case to the corresponding discipline handler:

```plaintext
function CompiledSUB1..10 (fn, args),
  (fn) (a1, a2, ... an),
endfun;
```

which passes the actual arguments without evaluation and then invokes the subroutine, returning the subroutine's value.

```plaintext
x: 'a&
y: 'b&
z: 'c&
apply ('concat, '(x y z))&
-> abc i.e. concat (x, y, z).
```

The subroutines `and` and `or` take a variable number of arguments:

```plaintext
subroutine sub x,
  <subroutine body>,
endsub;

apply ('sub, list (a1, a2, ... an)),
```

The corresponding discipline handler just passes control to the desired code, the arguments are not touched at all!

```plaintext
function CompiledSUB (fn, args),
  (fn) (args),
endfun;
```

Macros are treated in much the same way as subroutines, however...
their result is evaluated by eval, c.q. apply. Macro's are used for text substitution purposes. Their discipline handlers are:

```lisp
function CompiledMAC1..10 (fn, args),
  eval ( (fn) (a1, a2, ... an)),
endfun;

function CompiledMAC (fn, args),
  eval ( (fn) (args)),
endfun;
```

```lisp
example:
```

Next we show the use of a macro to expand the n-ary and operator into the binary and2 operator:

```lisp
macro and x,
  case
    empty (rest (rest (x)))
    {cons ('and2, x),
    default
    {cons ('and2, pop (x), cons ('and, x))},
  endcase,
endmac;
```

One discipline handler treats all undefined functions:

```lisp
function undefined (fn, args),
  case
    valid_first_rest (fn)
    {<define the first-rest combination>,
      apply (fn, args)},
    default
    {break (cons ('fn, args))},
  endcase,
endfun;
```

The recognizer valid_first_rest will detect an undefined function of the form frrrest (x) by inspecting the print-name of the undefined function, which stands for the fourth element of a list. The action <define the first-rest combination> will then define a function:

```lisp
function frrrest (x),
  first (rest (rest (rest (x))));
endfun;
```

See also section 3.2 under generalized selector functions.
If MoDL cannot recover from the fact that a function isn't defined it will cause a throw to the label _eval_, with the form which cannot be applied as return value.

### 11.9.1.2 Execution of interpretative MoDL code

The body of an interpretative function is a list data-structure with references to the formal arguments and local variables. The function _def_ takes care of the fact that the formal arguments and local variables are really local. To this extend it creates new names for the formal-arguments and local-variables with exactly the same print-name at a physically different address, i.e. such that \( x == \text{local}(x) \) is always false. These new names are then substituted into the body in order to guarantee the same semantic actions the interpreter as in the compiled code.

The function-body will be evaluated by the discipline handler using _eval_. This suggests that the values of the actual-arguments should be assigned to the formal-arguments, eventually after application of _eval_, in exactly the same manner as in the compiled case, while saving the old values of the formal-arguments on the stack, to allow recursion. All local-variables get an initial value _false_, while saving the previous values on the stack.

The value of the formal-arguments is restored at the return of the evaluation of the interpretative body, and the return value is passed to the caller.

### 11.9.2 Evaluation of MoDL language constructs

**case** (list), is the basic conditional statement in the MoDL system. The MoDL syntax for this statement is:

```moodel
  case
e10 \{ e11, ... e1n \},
e20 \{ e21, ... e2n \},
em0 \{ em1, ... emn \},
endcase,
```

The internal representation of the case construct is:

```moodel
'(case \( e10 \ e11 \ldots \ e1n \) 
\( e20 \ e21 \ldots \ e2n \)
\ldots
\( em0 \ em1 \ldots \ emn \) )
```

---

6.111
Eval evaluates the eiO for i:1..m until any of them becomes non-
false. If any has been found then the evaluation proceeds with the
eiO .. ein. The return value of the case expression is the last
evaluated expression. This implies that eiO is the return value if
the i-th clause of the case had been written as:

\[
eiO \{!\}
\]

So mind that:
\[
eiO \{!\}
\]
differs from:
\[
eiO \{false\}
\]

loop (el, e2 ...em), evaluates the expressions in list iteratively until
an exitif has been executed. The return value of the loop is the
last expression executed in the exitif language construct.

Syntax:

\[
\text{loop} \\
e1, \\
e2, \\
... \\
em, \\
\text{endloop},
\]

Internal representation:

\[
'(\text{loop } e1 e2 \ldots em) \\
\text{subroutine loop } x, \\
\text{local } (x0), \\
x0: x, \\
\text{loop} \\
\text{case} \\
\text{not } (\text{pair } (x)) \{x: x0\}, \\
\text{endcase}, \\
\text{eval } (\text{pop } (x)), \\
\text{exitif retval } \{\}, \\
\text{endloop,} \\
\text{endsub};
\]

exitif (e0, e1 ...em), evaluates e0. The remaining expressions are, if
any, also processed if the value of e0 is non-true. The exitif
construct causes any loop in which the exitif is embedded to
return with the value of the last expression evaluated.
Syntax:

```
exitif e0 \{ e1 e2 ... en \},
```

Internal representation:

```
'(exitif e0 e1 e2 ... en)
```

```
subroutine exitif x,
  case
    retval : eval (pop (x)),
    {loop
      exitif not (pair (x)) {{},
      retval : eval (pop (x)),
    endloop}
  endloop.
endsub;
```

Note that the given definitions of the subroutines loop and exitif cannot be interpreted as given, as their bodies contain a loop and an exitif construct too. The compiled versions of these subroutines can however be executed without any problem.

```
foreach el in list do ex1, ex2, ...exn endfor;
```

```
foreach el in list collect ex1, ex2, ...exn endfor;
```

```
foreach el in list collectif ex1, ex2, ...exn endfor;
```

```
for x:x1 step x2 until x3 do ex1, ex2, ...exn endfor;
```

```
forxv ex1 repeat ex2 until ex3 do ex1, ex2, ...exn endfor;
```

```
catch (label, body), evaluates <body> as if it was a function body. Any invocation of throw within a function reached through the <body> of the catch with a label identical to <label>, will revert control to the catch with a return value determined by the throw. The label false catches all nested throws.
```

```
throw (label, v), reverts control to the matching catch, which will return with the value <v> specified in the catch.
```

```
trace (l1, l2 ... fn), trace sets an environment to print all arguments of an interpretative MoDL function before invocation of that function, and to print the return value when the function is left. The arguments to trace are either truenames or quoted true names. The return value is the list of successfully traced.
```
functions.

\textbf{untrace} (f1, f2 ... fn), \texttt{untrace} removes the trace-environment of a traced function. It returns the list of successfully untraced functions.

**II.9.3 Memory management**

\texttt{gc} (), causes an explicit re-collection operation for all MoDL data-objects. All data objects which can be accessed through the oblist or the runtime stack are marked. The remaining objects are reclaimed to the free lists associated with the data spaces. An additional condition for truenames on the object-list is that their value should corresponds to their initial value, their plist should be empty and their function pointer should be false false, before the truename gets reclaimed. The special function \texttt{removename} set all these conditions.

\texttt{removename} (truename), sets all additional conditions needed to reclaim a truename from the oblist. Note that the object will remain in the system if it is still referenced from any of the data spaces.

**II.9.4 Driver**

\texttt{toplevel driver} (), is the top-level driver of the MoDL system, it is compiled and \textit{not} accessible to the user. The function accepts user-specific initialisations from the file:

\texttt{.initmodl.list} from the defined path.

This file is obtained from a transload of the MoDL file:

\texttt{.initmodl.modl}.

The function then invokes driver ('modl, 'modl), i.e. it starts the interaction with the user assuming MoDL-input syntax and MoDL-output syntax.

The user can terminate its interaction with the driver by typing \texttt{~D}, i.e. ascii (5). The list-driver will respond with "quit ?" if control is passed back to the toplevel_driver as a result of a \texttt{~D} in the input stream. A single character y or Y will indeed terminate the current MoDL session, any other response however will continue it.
function toplevel_driver (),
  local (breakmsg),
  load (".initmdl, 'list),
  loop
    breakmsg: catch (false,
      loop
        driver ('modl, 'modl),
        foreach port in close do close (port)
        endfor,
      endloop),
    case
      breakmsg == ascii (4)
      {print ('"quit ?"'),
        case
          member (readchar (), 'Y')) {quit ()},
        endcase},
    endcase,
  endloop,
endfun;

driver (readSyntax, printSyntax), reads from the given infile with the specified <readSyntax> and prints on the given outfile with the specified <printSyntax>. A throw (false, ascii (4)) reverts control to the calling function with the last input read as the driver-result.

The ports in echo and outecho determine the way in which the input and output are echoed (see section II-8.1. for details). The driver issues a prompt if the infile is stdin, i.e. in the case of an interactive session.

The user can choose the <print_syntax> in an interactive session, i.e. if infile == stdin, by typing an appropriate terminator character if the <read_syntax> == 'modl.

<table>
<thead>
<tr>
<th>Terminator</th>
<th>Print-syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>modl</td>
</tr>
<tr>
<td>&amp;</td>
<td>list</td>
</tr>
<tr>
<td>$</td>
<td>noPrintout..</td>
</tr>
<tr>
<td>~D</td>
<td>modl</td>
</tr>
</tbody>
</table>

page II-9-10
declare (special (read, print, driver))$

function driver (read, print),
  local (breakmsg, prompt, driver),
  prompt: concat (driver: driver + 1,
  case
    get (read, 'prompt') {}
  default
    {'?'}
  endcase),
loop
breakmsg: catch (false,
  case
    read == 'modl
    {loop
      case
        infile == stdin
        {printatom (prompt, "")}
      endcase,
      set (prompt, eval (read ()),
      endloop},
    read == 'list
    {case
      print == 'modl
      {loop
        case
          infile == stdin
          {printatom (prompt, "")}
        endcase,
        print (set (prompt,
          eval (readlist ()))),
        printatom (', newln),
      endloop},
      print == 'list
      {case
        infile == stdin
        {loop
          printatom (prompt, "")},
          printlist (set (prompt,
            eval (readlist ()))),
          printatom (', newln),
        endloop},
        default
        {loop
          printlist (set (prompt,

page II-9-11
eval (readlist ())).

endloop
endcase

read == 'o

{<load the machine dependent object-file>},
endcase.

exitif breakmsg == ascii (4) {prompt},
printatom ('"run-time error: "'),
print (breakmsg, newln).
endloop.
endfun;

put ('modl, 'prompt, '.');
put ('list, 'prompt, '>');

function ; (x),
case
 (print == 'list) and (infile == stdin)
 {printlist (x)},
default
 {print (x)},
endcase.
endfun;

function & (x),
case
 (print == 'modl) and (infile == stdin)
 {print (x)},
default
 {printlist (x)},
endcase.
endfun;

function $ (x),
x,
endfun;
function \(D\ (x)\),
case
  (print == 'list) and (infil == stdin)
    {printlist (x)},
  default
    {print (x)},
endcase,
throw (false, ascii (4)),
endfun;

break (msg), prompts with "break: "<msg> to show the cause of an error, and starts a new driver for interaction with the user. A user prompt followed by \(D\) (ascii (4)) gives a substitute value to be used at the place of error, a user prompt followed by \(E\) (ascii (5)) gives control to the next higher driver.

function break (msg),
  printatam ("break", "", ""),
  print (msg, newln),
  driver ('modl', 'modl),
endfun;

eval (a);

break: eval (a) -- comment: --
  1: bound (a); -> false cause of the error
  1: 'a<\~D> -> a repair value typed by user
  0: a MoDL prompts repair value

rest ('a);

break: rest (a) -- comment: --
  1: atom (a) -> true rest of atom is undefined
  1: false<\~D> -> false repair value typed by user
  0: false MoDL prompts repair value

II.9.5 Environment

shell (), creates a new unix driver (shell).
function shell (),
  <create a new unix driver (shell)>,
endfun;

shell ():          -- comment: --
>ls               list the content of the directory
unix response
<-D>              leave the unix shell
*:                 back in the running MoDL system.

quit (), calls the C_exit routine with argument 0, and hence terminates the running MoDL process.

function quit (),
  exit (0),
endfun;
quit ():           -- comment: --
>ls                leave the running MoDL system
*:                 back in unix, ls command

time (), returns a list of four integers:

(usetime delta-usetime gctime delta-gctime)

Each integer in the list corresponds to the time units on the running machine, which amounts frequently to 1/60 or 1/50 of a second. The variable time is used to hold the number of time units on the running machine which corresponds to a second of CPU-time.

time ()&
time&
60

system (string), make a call to the unix 'system' function with the given <string> as argument.

function system (string),
  case
    trueame (string)
    {system (string)},
default
    {break (list ('system, string))},
endcase,
endfun;

kill (pid, signal), call unix to 'kill' the process with the integer
process identifier <pid> and the integer signal <signal>.

\[\text{function kill (pid, signal).}\]
\[\quad \text{case}\]
\[\quad \quad \text{fixnum (pid) and fixnum (signal)}\]
\[\quad \quad \quad \text{kill (pid, signal)}\]
\[\quad \quad \text{default}\]
\[\quad \quad \quad \text{throw (false, list ("kill, pid, signal"))}\]
\[\quad \text{endcase}\]
\[\text{endfun;}\]

\text{fork ()}. \]

\text{exec (path, a1, a2, ...an)}.

\text{signal (n, function)}.\]
III.1 The Hardware Description Language.

Figure III.1.1 gives a survey of the MoDL implementation. It shows how parallel algorithms and hardware descriptions are mapped to the MoDL list-level by means of the MoDL read and print functions. This second part of the reference manual defines the precise details of the MoDL parallel language and hardware description system. Remind that all functions, with the exception of the evaluation functions, defined in part II of the reference manual are also part of the MoDL parallel language, c.q hardware description system.

Figure III.1.1. A survey of the MoDL implementation

This paper was produced as part of Task VIII of the ICD project which has received partial support by the Commission of the EEC under the MR-09 contract.
III.2 Model Definition and Syntax

III.2.1 Model Definition

A model defines a (sub)network composed of interconnect, which carries the operands, and functions which act upon the operands.

Models are defined with the subroutine modeldef in much the same way as functions are defined with def. The definition of a model can be retrieved with getmodeldef.

```
subroutine modeldef x,
.
endsub;
```

```
function getmodeldef (x),
get (x, modeldef),
endfun;
```

Model definitions are never compiled, as this internal representation is needed during the model-expansion phase, which interconnects models and optimizes their representation, to form a complete model-instance.

III.2.2 Model Definition Syntax

```
model f (x1, x2, ... xn),
.
endmod;
```

Defines a (computational) subnetwork with external terminals, or formal arguments, x1, x2, ... xn. A default integer-arithmetic domain is used for the body of a model. See III-2.4 for arithmetic domains.

```
dsmodel f (x1, x2, ... xn),
.
endmod;
```

Defines a (computational) subnetwork with external terminals, or formal arguments, x1, x2, ... xn. A default user-defined-arithmetic (fractional) domain is used for the body of a dsmodel. The acronym dsp stands for digital signal processing, where fractional arithmetic is very common. The reader is referred to section III-2.4 for arithmetic domains.
III.3 Execution of Parallel Algorithms

' x, returns <x> without evaluation. This construct is used to represent a symbolic value in a computational network. Model-instances with symbolic values propagate the expression <x> in its literal form. This operator is frequently used in examples to avoid problems during model-instantiation, as models without bound-values are not accepted by the MoDL instantiation toplevel.

label : x, associates the expression <x> with the label <label> and returns <x>. The combinatorial association between the <label> and its corresponding expression <x> is unique and does not depend on time or execution order. This is an important semantic difference with the assignment in the MoDL programming environment, which accepts multiple assignments which may override previous values, depending on order of execution. It is strongly advised that the MoDL user views its parallel algorithms as a computational network, in which labels can be used to reference a signal, or variable, at several places. This view makes it quite clear that parallel languages can be characterized in terms of the terminology of traditional programming languages as single assignment languages. MoDL signals an error if a single label is used for multiple expressions.

It should be noted that a labeling can be qualified with a condition. It is in this case that the parallel algorithm or hardware description may appear to have multiple assignments, however the qualification of the assignments should be such that the corresponding conditions are mutually exclusive, this guarantees the uniqueness of the labeling. A special MoDL module can be used to prove or disprove mutual exclusivity of such qualifications.

It is not allowed either that the value of <label> can depend in a combinatorial way on its own value, i.e. a combinatorial path through the computational network may not reference <label>. Conditional paths as described with the case language constructs are (to be) tested for mutual exclusion.

case (list), is the basic conditional language construct in the MoDL system. Its syntax is:
Each case subexpression:

\[ e_{10}, \ldots , e_{in} \], represents a unique case-context:

which activates the context-expressions \( e_{11}, \ldots , e_{in} \), i.e. any subnetwork represented by the context-expressions is present if the case-condition \( e_{10} \) is logically true and absent if it is logically false. Case-conditions \( e_{10} \) which depend on a value propagated by its own context-expression are not allowed and hence marked as an error condition during the model-instantiation phase.

The value propagated by the case-context is the value of the last context-expression, i.e. the value of \( e_{in} \) or in the special case

MoDL implements two aliases for the case construct, the demandcase and the datacase. These language constructs which differ only in name from the case construct are used to define the execution view which should be supported during the timing analysis of a parallel algorithm, either demand-driven or data-driven as described in section III-7.
This construct requires that the case-conditions:
\( e_{10}, \ldots, e_{m0} \) are mutually exclusive.

Example: a 3-input multiplexer:
```plaintext
x: case
  n=1 \{x1\},
  n=2 \{x2\},
  n=3 \{x3\},
endcase,
```

Example: a 3-output demultiplexer:
So in contrast with the usage of the case statement in the MoDL programming environment we assume full parallel evaluation of the case language construct for hardware description purposes.

```plaintext
    case
        n=1 \{ x1: x \},
        n=2 \{ x2: x \},
        n=3 \{ x3: x \},
    endcase,
```

activates the `loop` expressions `e_1`, ..., `e_m`, i.e. all subnetworks represented by the specified loop-expressions are present for iterative calculation.

The loop language construct propagates also a value, which is actually seldom used, in a way similar to the usage of the loop in the MoDL programming environment. This value is here too given by the value of the last expression present in the `exitif` language construct.

The loop construct drives all master-slave updates for the `registers` defined within the loop, as it will be described in the next section. Loops can only be nested within each other to represent systems with multiple clocks. The `exitif` construct determines the relative rates of the nested clocks.

The `exitif` expression:

```plaintext
    exitif (e_0) \{ e_1, e_2, ..., e_m \},
```

represents a unique exit-context:

```
\[
\begin{array}{ccc}
  e_0 & \rightarrow & (e_1, \ldots, e_m), \\
\end{array}
\]
```

which activates the exit-expressions \{ `e_1`, ..., `e_m` \}, i.e. any subnetwork represented by the context-expressions is present if the case-condition `e_{10}` is logically true and absent if it if-condition
e₀ is always active when the loop itself is active.

The register expression:

\( \text{label} := \text{x}, \) defines <label> as a register variable, which holds a current master-value, either derived from a master-slave update in the previous loop iteration (clock cycle), or given as initial value in the first pass through the loop. The slave value of the register is determined by the expression \( \text{x} \).

The clocked association between the <label> and its corresponding expression \( \text{x} \) is unique and does not depend on time or execution order. This is an important semantic difference with the assignment in the MoDL programming environment, which accepts multiple assignments which may override previous values, depending on order of execution. It is strongly advised that the MoDL user views its parallel algorithms as a computational network, in which labels can be used to reference a signal, or variable, at several places. This view makes it quite clear that parallel languages can be characterized in terms of the terminology of traditional programming languages as single assignment languages. MoDL signals an error if a single label is used for multiple expressions.

It should be noted that a labeling can be qualified with a condition. It is in this case that the parallel algorithm or hardware description may appear to have multiple assignments, however the qualification of the assignments should be such that the corresponding conditions are mutually exclusive, this guarantees the uniqueness of the labeling. A special MoDL module can be used to prove or disprove mutual exclusivity of such qualifications.

Conditional paths as described with the case language constructs are (to be) tested for mutual exclusion.

**Note:** \( \text{master} := \text{x}, \) is equivalent with:

\( \text{master} := \text{slave} \cdot \text{x}. \)

It can never happen that the value of <master> can depend in a combinatorial way on its own value, as the register expression is always of a clocked nature, it is for this reason that the combinatorial expression \( \text{x} \cdot \text{x}+1 \), is not allowed, whereas its clocked variant \( \text{x} := \text{x}+1 \) is.
III.4 Support Functions

MoDL gives the user a rich set of programming and hardware description tools, as all language constructs from part I of this manual can be used in a hardware description. However there is a need to distinguish between support functions, such as invocations of a print function to support simulation or declarative statements to support a certain design view, and the actual hardware which is being qualified by these support functions. It is the purpose of this section to give a survey of all support constructs currently defined. The actual implementation of support functions for specific views is discussed in separate sections.

III.4.1 Additional Specifications for Simulation

support (ex1, ex2, ...exn), returns false, in case of a MoDL simulation. The whole support expression is neglected for all other views.

    subroutine support x,
        loop
            exit if empty (x) {false},
            eval (pop (x))
        endloop,
    endsub;

III.4.2 The Description of Wiring

pins (truenname, list), is used to indicate that <truenname> is connected to the pins specified in <list>. The model expansion of this support function returns modelexp (<truenname>). The actual content and processing done on the specified <list> depends strongly on the application. It is for instance recommended discrete designs using 7400 like "packages" include the package instance name, i.e the modelexp of the package name and the actual pins which hold the signal <truenname> in the specified <list>.

III.4.3 The Specification of Timing

delay (fun, n), is a MoDL programming construct used to specify the time-delay dt(<fun>) introduced by the specified primitive function. i.e. the output of the function is assumed to be stable at:

\[ t = \max (t_{x1}, t_{x2}, \ldots t_{xn}) + dt (fun) \]

With t_{x1}, t_{x2},... t_{xn} the moments at which x1, x2, xn are stable, and dt (fun) the delay associated with the specified function.
Delay specifications have a global nature and should not be used in a model definition.

The function returns \( n \).

\[ \text{dt} (\text{expr}, n) \]

is used to indicate that \(<\text{expr}>\) has an additional delay of \(<n>\) time-units. It is used to include local delay specifications within model definitions. I.e the output of \( \text{dt} (\langle\text{expr}\rangle, n) \) is assumed to be stable at:

\[ t = t_{\text{expr}} + n \]

With \( t_{\text{expr}} \) the moment at which \(<\text{expr}>\) is stable, and \( n \) the extra delay specified.

The model expansion of this support function returns \( \text{modelexp} (\langle\text{expr}\rangle) \).

\[ \text{demandcase} \ (\text{list}) \]

is used to indicate that the case expression given in \(<\text{list}>\) is demand driven. The demandcase expression executes in exactly the same way as the normal case expression and has, apart from the name differences the same syntax.

\[ \text{datacase} \ (\text{list}) \]

This support function indicates that the case expression given in \(<\text{list}>\) is data-driven, an important feature for the timing analysis of an algorithm.

The syntax used with the demandcase and datacase expressions is similar to the one of the case ... endcase statement.

\section*{III.4.4 The Description of Electrical Effects}

\[ \text{capacitance} \ (\text{truename}, n) \]

is used to indicate that the model expansion of \(<\text{truename}>\) has a capacitance to ground with value \(<n>\). The model expansion of this support function returns \( \text{modelexp} (\langle\text{truename}\rangle) \).

\[ \text{strength} \ (\text{truename}, n) \]

is used to indicate that the model expansion of \(<\text{truename}>\) has a driving strength of \(<n>\) units. A user supplied function is used to calculate the extra-delay as a function of capacitive loading and (increased) driving force for the signal \(<\text{truename}>\). The model expansion of this support function returns \( \text{modelexp} (\langle\text{truename}\rangle) \).
III.5 Additional Primitive Functions

The functions described in this section extend the primitives covered by part II of the reference manual. They are included in this section either because of their special syntactic support like for instance the arithmetic domains or their special relevance for hardware descriptions.

III.5.1 Arithmetic Domains

The user can use three types of arithmetic domains within a design:

1. Integer arithmetic,
2. User-defined arithmetic,
3. Overflow protected user defined arithmetic.

By default installed is a fractional arithmetic package, which has a variable precision, controlled by the variable one.

\[
[ \text{ex}_1, \text{ex}_2, \ldots, \text{ex}_n ] \text{ applies integer arithmetic to the expressions } \text{ex}_1, \text{ex}_2, \ldots, \text{ex}_n \text{ and returns } \text{ex}_n.
\]

\[
\{ \text{ex}_1, \text{ex}_2, \ldots, \text{ex}_n \} \text{ applies fractional or user-defined arithmetic to the expressions } \text{ex}_1, \text{ex}_2, \ldots, \text{ex}_n \text{ and returns } \text{ex}_n.
\]

\[
\text{ovl} \{ \text{ex}_1, \text{ex}_2, \ldots, \text{ex}_n \} \text{ applies overflow protected fractional or overflow protected user-defined arithmetic to the expressions } \text{ex}_1, \text{ex}_2, \ldots, \text{ex}_n \text{ and returns } \text{ex}_n.
\]

The mentioned overflow protection changes the 2-complement value after overflow to a value with the same sign as the product of the operands, but with a (much) smaller magnitude to minimize the effect of the error.

III.5.1.1 Arithmetic Domains for +

The table below shows how the mentioned domains influence the representation of the + operator. The MoDL printlist function will show the presence of the "+" "+" and the "ovl+" operators. The print function hides the presence of these specific operators in the underlying list-notation, by the use of the + operator and the corresponding domain brackets.
Additional Primitive Functions

\[
\begin{align*}
\text{print} (\text{"[+" a b c\text{"]\}}) & \rightarrow \{a + b + c\} \\
\text{delay} (\text{"[+", 150)} & \rightarrow \text{delay specification for timing analysis.}
\end{align*}
\]

III.5.1.2 Arithmetic Domains for *

The table below shows how the mentioned domains influence the representation of the * operator. The MoDL printlist function will show the presence of the "[+" "[\" and the "ov[+" operators. The print function hides the presence of these specific operators in the underlying list-notation, by the use of the * operator and the corresponding domain brackets.

\[
\begin{align*}
\text{print} (\text{"n1 + n2 + .. + nn\}}) & \rightarrow \{n1 + n2 + .. + nn\} \\
\text{ov[+" n1 + n2 + .. + nn\}} & \rightarrow \text{ov[+" n1 + n2 + .. + nn\}}
\end{align*}
\]

III.5.2 Representation of Arithmetic and Word-oriented Data

The user should be aware of the fact that he can choose his data to be represented in signed format or alternatively 2-complement form. The user-defined arithmetic has a default fractional implementation which uses a signed number representation. The word operations are implemented on unsigned 2-complement representations as well as on signed number representations. Two auxiliary functions are available to make conversions between both representations.

\[
\begin{align*}
\text{decode (x), returns <x> in 2-complement notation. The variable one is used as a reference to the range of the number system.} \\
\text{encode (x), returns <x> in sign-encoded format. The variable one is used as a reference to the range of the number system.}
\end{align*}
\]

III.5.3 Boolean Functions

The logical functions described in part I of the reference manual take false as the unique representation for logical-false and any other value as a representation for logical-true. Hardware designers however tend to use boolean values represented by the numbers 0 and 1 when they describe low level logic.
The evaluation of the boolean functions defined here is also different. The boolean functions evaluate all their operands, no matter whether the output value is already determined by an already evaluated argument, whereas the logical functions stop with the evaluation of their arguments as soon as the remaining inputs do not influence the computed result.

Special attention has been paid to the aspect of hardware debugging. All inputs are checked to represent values in the boolean domain, and the argument lists are checked to be well formed. The implementation of these functions as subroutines makes the break handling very user friendly: Any input signal to these functions which does not hold a boolean value generates a MoDL-break, showing as part of the break response, either the symbolic name of the signal or the subexpression which formed the signal.

_or (x1, x2, ... xn), tests for correct arguments and returns the boolean or.

_nor (x1, x2, ... xn), tests for correct arguments and returns the boolean nor.

_and (x1, x2, ... xn), tests for correct arguments and returns the boolean and.

_nand (x1, x2, ... xn), tests for correct arguments and returns the boolean nand.

_xor (x1, x2, ... xn), tests for correct arguments and returns the boolean xor.

_not (x), tests for a correct argument and returns the boolean not.

Only the MoDL definition of _or is given here as all boolean functions have a similar implementation.
subroutine _or xx,
  local (x, result),
  case
    atom (xx)
      {break (cons ('_or, xx))},
    NotBoolean (result: eval {first (xx)})
      {break (cons ('_or, xx))},
    default
      {loop
        exitif (empty (xx: rest (xx)):
          result),
        exitif (atom (xx))
          {break (cons ('_or, xx))},
        x: eval {first (xx)},
        exitif (NotBoolean (x))
          {break (cons ('_or, cons (result, xx)))},
        result: _or (result, x),
      endloop},
  endcase,
endsub;

Example:
  x: 0$
  y: 1$
  z: 'a$
  _or (x, y, z)$ -> break: _or (1, z)

The function _test is automatically inserted in the control branch of a case .. endcase if a boolean function is used in this position, as the boolean function on its own would always execute as a true-case as 0 and 1 both represent logical true.

function _test (x),
  case
    x=0 {false},
    default {x},
  endcase,
endfun;
The MoDL print function will suppress the _test function.

print ('(_test 1))$ -> 1
III.5.4 Operations on Words

Sixteen different boolean operations can be performed in parallel on the bits of a word with the function bool. A mask is used to specify the bits upon which the operation acts.

`bool (x1, x2, op, mask)`, returns the result of the bitwise-boolean operation `<op>` for all bits specified in `<mask>` on the 2-complement operands `<x1>`, `<x2>.

<table>
<thead>
<tr>
<th>op</th>
<th>bitwise result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>_and(x1, x2)</td>
</tr>
<tr>
<td>2</td>
<td>_and(_not(x1), x2)</td>
</tr>
<tr>
<td>3</td>
<td>x2</td>
</tr>
<tr>
<td>4</td>
<td>_and(x1, _not(x2))</td>
</tr>
<tr>
<td>5</td>
<td>x1</td>
</tr>
<tr>
<td>6</td>
<td>_xor(x1, x2)</td>
</tr>
<tr>
<td>7</td>
<td>_or(x1, x2)</td>
</tr>
<tr>
<td>8</td>
<td>_nor(x1, x2)</td>
</tr>
<tr>
<td>9</td>
<td>_not(_xor(x1, x2))</td>
</tr>
<tr>
<td>10</td>
<td>_not(x1)</td>
</tr>
<tr>
<td>11</td>
<td>_or(_not(x1), x2)</td>
</tr>
<tr>
<td>12</td>
<td>_not(x2)</td>
</tr>
<tr>
<td>13</td>
<td>_or(x1, _not(x2))</td>
</tr>
<tr>
<td>14</td>
<td>_or(x1, _not(x2))</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

It is left as an exercise for the reader to (re)write the bool function with the case ... endcase language construct, i.e. in terms of multiplexers.

`threebits: 7;`

```c
% 3-bits _and (x1, x2):
bool (2, 3, 1, threebits); -> 2
% 3-bits _or (x1, x3):
bool (2, 3, 7, threebits); -> 3
% 3-bits _not (x1):
bool (2, 0, 10, threebits); -> 5
```

`singedbool (x1, x2, op, mask)`, returns the result of the bitwise-boolean operation `<op>` for all bits specified in `<mask>` on the signed operands `<x1>`, `<x2>`.

page III-5-5
function signedbool (x1, x2, op, mask),
    encode (bool (decode (x1), decode (x2), op, mask)),
endfun;

III.5.5 Distributed Logic
MoDL has a special mechanism for dealing with single functions that are realized from parts defined in separate models. These functions are known as wired-or and related functions.

pulldown (out, in1, in2, ... , ink), specifies k parallel transistors between the ground and the "out" pin inside the model in which it is declared. These transistors are controlled by the signals "in1", " ... " ink" connected to their gates. A similar declaration within one or more other models will make that MoDL will merge the two (or possibly more) pull-down devices with the same output names into a single wired-nor device. During this merging a single pull-up device (e.g. a depletion load in nMOS technology) will be added symbolically between "out" and the power to complete the nor gate.

Example:
model ml(..., mlout),
    pulldown(mlout, mllocal)
endmod:
model m2(..., m2out),
    pulldown(m2out, m2local)
endmod:
model supermod(...),
    ml(..., superlocal),
    m2(..., superlocal),
endmod:

In "supermod" the two pull-down outputs get the same name (i.e. they are made to coincide), which is an indication for the MoDL model expansion program that they should be merged. The result is a single nor gate in the expanded model description having two outputs: "mllocal" and "m2local".

pullup (out, in1, in2, ... , ink), is analogous to the "pulldown"
specification. However, the parallel transistors are now located between the "out" pin and the power instead of the ground.
III.6 Layout Construction by Abutment

III.6.1 User Interface

The top-level user interface consists of the keyword abut, followed by the name of the model to be formed, and an abutment expression.

\[ \text{abut} \text{name, exp}; \] returns the \(<\text{name}>\) of a new model which is formed in accordance to the abutment specification in \(<\text{expr}>\).

MoDL supports a modified argument list syntax to specify the layout of a cell, for instance the BitAdder:

\[
\begin{array}{c}
\text{SI} \\
\text{Cout} \leftrightarrow \text{Cin} \\
\text{A1, B1} \\
\end{array}
\]

is specified as:

\[
\text{model BitAdder } ((A1, B1), (Cin), (S1), (Cout)).
\]

\[
\text{endmod;}
\]

to indicate the fact that \((A1, B1)\) are on the south side, Cin is on the east side, S1 is on the north side and Cout is on the west side of the BitAdder. Parameters which do not represent wiring, such as values of wiring capacitances and the like, can be specified either with global MoDL data or in the remaining parameter list.

\[
\text{model BitAdder } ((A1, B1), (Cin), (S1), (Cout), \cap3, r5).
\]

\[
\text{endmod;}
\]

This layout description can be used in conjunction with an Abutment algebra, which allows the user to define new model descriptions using \(<\text{model identifier}>\)s as operands of the abutment operators - and +.
The abutment algebra bridges the gap between an algebraic description language and a layout based system. For instance the toplevel expression:

\[
\text{abut ThreeBitAdder, BitAdder - BitAdder - BitAdder;}
\]

defines the layout and the algebraic structure of the ThreeBitAdder all at once. The newly generated ThreeBitAdder is equivalent with:

\[
\text{model ThreeBitAdder ((A3, B3, A2, B2, A1, B1), (Cin), (S3, S2, S1), BitAdder (A3, B3, Carry2, S3, Cout), BitAdder (A2, B2, Carry1, S2, Carry2); BitAdder (A1, B1, Cin, S1, Carry2), endmod;}
\]

i.e. it describes an algorithmic structure with exactly the same instantiation. It is, however, likely that internal names, which are irrelevant anyhow, are different.

A repetition factor can be used as the first operand of the abutment operator:

\[
\text{abut ThreeBitAdder, 3 - BitAdder;}
\]

is equivalent to the ThreeBitAdder described before. This implies that a 16Bit WordAdder can be described as:
Even complicated two dimensional structures are easily described with the abutment algebra, for instance the Object:

```
Object:

        Top
        |   |
        |   |
        |   |
        |   |
        Cell
        |   |
        |   |
        |   |
        Bottom
```

is described with the simple top-level expression:

```
about Object, Bottom + ( 4 + ( 6 - Cell)) + Top;
```
### III.7 Timing Analysis

#### III.7.1 User Interface

The top-level user interface consists of two model-instantiation functions, which start the timing analysis process and a function which defines the delay of an individual operator.

**timing** [Mexpr]; perform a model expansion assuming an integer arithmetic domain, a demand-analysis and a timing analysis of the given model-expression <Mexpr>.

The function prints the delay values of all the user-defined variables in the model expression <Mexpr> and the critical path for every loop expression. It returns `true`.

**timing** {Mexpr}; perform a model expansion assuming a user defined arithmetic domain, a demand-analysis and a timing analysis of the given model-expression <Mexpr>. The function prints the delay values of all the user defined variables in the model expression <Mexpr> and the critical path of every loop expression. It returns `true`.

**delay** (run, n), is a MoDL programming construct used to specify the time-delay \( dt(\text{<fun>}) \) introduced by the specified primitive function.

Delay specifications have a global nature and should not be used in a model definition. The function returns `n`.

```mo dl
function delay (op, n),
put (op, `delay, n),
endfun;

delay(F, 10);  --> 10
```

#### III.7.2 Definition of the Delay of a Function

A function \( F \), with operands \( x_1, x_2, \ldots, x_n \), which become stable at time instances \( t_1, t_2, \ldots, t_n \), has a stable output at:

\[
\tau : \max (t_1, t_2, \ldots, t_n) + dt(F),
\]

where \( dt(F) \) is the delay of \( F \).

Example:
model example1 (x, y, z).
  a : F (x, y),
  b : G (a, z),
endmod$

delay (x, 2)$
delay (y, 5)$
delay (z, 0)$
delay (F, 7)$
delay (G, 3)$

Given tx: 2, ty: 5, tz: 0, dt(F): 7, dt(G): 3 one finds:

\[ t_a = \max (2, 5) + 7 = 12 \]
\[ t_b = \max (12, 0) + 3 = 15 \]

\[ \text{timing [example1 ('x', 'y', 'z)]}: \]
\[
\begin{array}{ll}
  \text{signal} & \text{delay} \\
  a@1 & 12 \\
  b@1 & 15 \\
\end{array}
\]

\[ \text{critical path} = (15 \ y \ F \ a@1 \ G \ b@1) \]

All remaining specification for timing analysis should be done within models, e.g. dspmodels as they qualify the model expansion locally.

III.7.3 The Introduction of Local Delay

dt (expr, n), is used to indicate that <expr> has an additional delay of <n> time-units. It is used to include local delay specifications within model definitions. I.e the output of dt (expr, n) is assumed to be stable at:

\[ t = t_{expr} + n \]

With texpr the moment at which <expr> is stable, and <n> the extra delay specified.
The model expansion of this support function returns `modelexp (<expr>)`.

```plaintext
function dt (n, expr),
    expr,
endfun;

Example:

delay (_nand, 7)$

model example2 (a, b, c),
    z: dt (3, _nand (a, b, c)),
    y: _nand (a, b, c),
endmod$
```

An extra delay of 3 time units is added to the delay of the _nand with output x, no extra delay is given for y the _nand with output y, this latter gate still has the default delay of 7 time units.

### III.7.4 Timing Analysis of Conditional Expressions

The parallel calculation of results within a case subexpression can be performed in one of two different ways:

1. The functions which are subject to the condition are active irrespective of the value of this condition. The function values however, are only passed when the condition is true. This will be referenced as the data-driven timing view, which is specified with the `datacase ... endcase` syntax:

   ```plaintext
datacase
   t_1 (x) \{e_{11, \ldots, e_{1n}}\},
   t_2 (x) \{e_{21, \ldots, e_{2n}}\},
   \ldots
   t_m (x) \{e_{m1, \ldots, e_{mn}}\},
endcase,
```

2. The functions which are subject to the condition are only active if the condition demands the functions to be so. This will be referenced as the demand-driven timing view, which is specified with the `demandcase ... endcase` syntax:
Timing Analysis

III.7.4.1 Definition of Timing View of the Data Driven Case Construct

A case subexpression Cond\(j\) with a stable condition Cond at \(t_{\text{Cond}}\) and stable internal outputs \(x_1, x_2, \ldots x_n\), at time instances \(t_{01}, t_{02}, \ldots t_{0n}\), has stable external outputs at time instances \(t_1, t_2, \ldots t_n\), with:

\[ t_i = \max(t_{0i}, t_{\text{Cond}}) + \text{dt(Data)}, \]

where \(\text{dt(Data)}\) is the data driven delay of the case expression.

Example of data driven timing analysis:

```plaintext
model example3 (a, b, Cond),
data case
  Cond \{ x : F(a),
          y : G(x, b) \}
end case,
end mod$

delay (a, 2)$
delay (b, 0)$
delay (Cond, 9)$
delay (F, 3)$
delay (G, 6)$
```
The value of \( x \) is stable at \( t_x \):
\[
t_{\text{Cond}} + d_t(\text{Data}) = 9 + 2: 11.
\]

The value of \( y \) is stable at \( t_y \):
\[
t_a + d_t(F) + d_t(G) + d_t(\text{Data}) = 2 + 3 + 6 + 2: 13
\]

Timing \{example3 (\( 'a', 'b', '\text{Cond}' \))\}:

\[
\begin{align*}
\text{signal} & \quad \text{delay} \\
x & \quad 11 \\
y & \quad 13
\end{align*}
\]

Critical path = \((13 \ a \ F \ x\#1 \ G \ y\#1)\)

### III.7.4.2 Definition of Timing View of the Demand Driven Case Construct

A case subexpression \( \text{Cond}j \), with a stable condition \( \text{Cond} \) at \( t_{\text{Cond}} \) and stable external inputs \( x_1, x_2, \ldots, x_n \), at time instances \( t_{01}, t_{02}, \ldots, t_{0n} \), has stable internal inputs at time instances \( t_1, t_2, \ldots, t_n \), with:

\[
t_i = \max (t_{0i}, t_{\text{Cond}}) + d_t(\text{Demand})
\]

where \( d_t(\text{Demand}) \) is the demand driven delay of the case expression.

Example of demand driven timing analysis:
model example4 (a, b, Cond).

demandcase
    Cond \{ x : F (a), y : G (x, b) \},
endcase,
endmod$

delay (a, 2)$
delay (b, 0)$
delay (Cond, 9)$
delay (F, 3)$
delay (G, 6)$

The value of $x$ is stable at $t_x$:
\[ t_{\text{Cond}+\text{dt}(F)}: 11+3:14. \]

The value of $y$ is stable at $t_y$:
\[ t_{\text{Cond}+\text{dt}(F)+\text{dt}(G)}: 11+3+6:20 \]

timing [example4 ('a', 'b', 'Cond)]:

\[ \text{signal delay } x\#1 14 \]
\[ y\#1 20 \]

\[ \text{critical path} = (20\text{ COND}\#1 F x\#1 G y\#1) \]

III.7.5 Definition of Timing View of the loop ... endloop language
construct

The MoDL timing analysis module performs its analysis with respect to every loop ... endloop within a given model-instance.

page III-7-6
Clocked variables, i.e. the variables occurring in the left hand side of the register expression:

\[ \text{var} := \text{expression}, \]

are by definition stable at \( t=0 \).

In the result of a timing analysis however, the delay of a clocked variable represents the delay of the expression in the right hand sight of the respective register expression.

**Example:**

```model example5 ()
  loop
    y1 := F1 (y2),
    y2 := F2 (y1),
    y3 := F3 (y2),
  endloop,
  endmod$
end
```

```delay (F1, 10)$
delay (F2, 15)$
delay (F3, 20)$
```

The clocked elements are connected to the same clock signal.
Loops can be nested within each other to represent systems with multiple clocks. The exitif construct determines the relative rate of the nested clocks. The calculated delay values of variables declared within a certain loop expression are relative to the clock which drives that loop. Input variables of a loop expression, i.e. variables declared outside the loop context, are considered as clocked variables. They are assumed to be stable when the loop is activated.

The expression within the exitif construct are skipped during a timing analysis.

Example:

```model example6 (a),
loop
  y1: F1 (a),
  y2: F2 (y1),
loop
  y3: F3 (y1),
  y4: F4 (y3),
endloop,
  y5: F5 (y1, y4),
endloop,
endmod$

delay (F1, 10)$
 delay (F2, 20)$
 delay (F3, 30)$
 delay (F4, 40)$
```

---

```
timing [example5 ()]:
  --> signal delay
  y1#1 10
  y2#1 25
  y3#1 30

  critical path = (30 y2#1 F3 y3#1)
```

6.148
Results for the nested loop:

\[ ty_3 = dt (F3) + ty_1 = 30 + 0 = 30 \]

\( y_1 \) is declared outside the nested loop
\[ --> ty_1 = 0 \]

\[ ty_4 = dt (F4) + ty_3 = 40 + 30 = 70 \]

Results for the main loop:

\[ ty_1 = dt (F1) + ta = 10 + 0 = 10 \]

\( a \) is declared outside the main loop
\[ --> ta = 0 \]

\[ ty_2 = dt (F2) + ty_1 = 20 + 10 = 30 \]

\[ ty_5 = dt (F5) + max (ty_1, t' y_4), \]

where \( t' y_4 = ty_4 + dt \) (offset)

The delay value of \( y_1 \) is relative to the clock of the nested loop. An offset value must therefore be added to make it relative to the clock at the main loop. In this example, the nested loop is activated after signal \( y_2 \) becomes stable:

\[ --> dt \) (offset) = 30 \]

\[ t' y_4 = 70 + 30 = 100 \]

\[ ty_5 = 50 + max (10, 100) = 150 \]

**timing [example6 (a)]:**

\[ --> signal \) delay \]

\| y1#1 | 10 |
| y2#1 | 30 |
| y3#1 | 30 |
| y4#1 | 70 |
| y5#1 | 150 |

**LOOP#1**

critical path = (150 y4#1 F5 y5#1)

**LOOP#2**

critical path = (70 y1#1 F3 y3#1 F4 y4#1)

page III-7-9
III.7.6 Definition of Timing View of the Combinatorial Labeling

An expression:

```
var: expression,
```

can be qualified with electrical parameters, such as parasitic capacitances which load the named label, var, and driving strengths of outputs which drive it (see section III-8). A predefined function:

```
function "delay:" (var),
0,
endfun;
```

is used to substitute a zero delay for the combinatorial labeling. The user can change this function definition to reflect electrical parameters:

```
function capacitance (var),
case
  get (var, 'capacitance) {1},
  default {0},
endcase,
endfun;
function "delay:" (var),
capacitance (var) ^ 2,
endfun;
```

For instance the above definition assumes a quadratic delay, which takes the parasitic capacitance in an integrated circuit as well as the parasitic resistance (assuming a fixed line width) into account.

```
function strength (var),
case
  get (var, 'strength) {1},
  default {10},
endcase,
endfun;
function "delay:" (var),
capacitance (var) / strength (var),
endfun;
```

The above delay model makes it possible to reduce the actual amount of delay through inclusion of stronger driver transistors.
III.7.7 Critical Path Analysis

The MoDL timing module calculates the critical path for every loop expression in the model expression `<Mexpr>`. A critical path is a path within a computational network which has the maximum delay. The critical path is represented by a list of the form:

\[(\text{delay path})\]

or

\[(\text{delay (path1) (path2) . . .})\]

if there are parallel paths.

Each critical path contains all variables and function on the path. Critical subpaths with the same delay are enclosed in brackets.

Example:

```plaintext
model example7 (x),
   y1: F1 (x),
   y2: F2 (y1),
   y3: F3 (y1),
   y4: F4 (y2, y3),
endmod$

delay (F1, 10)$
delay (F2, 20)$
delay (F4, 40)$
timing [example7 ('x1)]$
```

![Diagram of the model example](image)

Page III-7-11
dt(F3) < 20 :
critical path =
(70 'z1 F1 y1#1 F2 y2#1 F4 y4#1)

dt(F3) = 20 :
critical path =
(70 'z1 F1 y1#1 (F2 y2#1) (F3 y3#1) F4 y4#1)

dt(F3) > 20 = 30 :
critical path =
(80 'z1 F1 y1#1 F3 y3#1 F4 y4#1)

If an operator has no delay value specified, a delay value 0 is assumed and the operator is excluded from the critical path.

Example:

```
model example7 (x),
y1: F1 (x),
y2: F2 (y1),
y3: F3 (y1),
y4: F4 (y2, y3),
endmod$
```

delay (F2, 20)$
delay (F3, 30)$
timing [example7 ('x1)]$

```
--> signal delay
   y1#1 0
   y2#1 20
   y3#1 30
   y4#1 30
```

critical path =
(30 'z1 y1#1 F3 y3#1 y4#1)

The critical path within a conditional expression is stored on the property list of the atom COND#n which uniquely identifies the conditional expression. The critical path of the condition can be retrieved by get('COND#n,'cpcond).

Example:

The critical path of model example4 (a, b, Cond) defined in section page III-7-12
III.7.4.2 is represented by the list:

```
critical path = (20 COND#1 F x#1 G y#1)
get ('COND#1, 'cpcond) --> (9 Cond)
```
IV.1 The MoDL-to-C Compiler

Figure IV-1-1 gives a survey of the MoDL implementation. It highlights the MoDL to C compiler which is used to obtain a portable machine code for the MoDL system, which will be described in short in this third part of the manual. Future versions of the manual will have a more detailed text on the compiler.

![Diagram of MoDL implementation](image)

**Figure IV-1-1**

**IV.1.1 Motivation for a New MoDL Implementation**

Early implementations of the Modelling and Design language, MoDL, were based on a LISP implementation. Lots of problems which were the consequence of the LISP implementation (and the availability of LISP on various machines) have made us decide to...
write an own MoDL implementation based on C. This version has a
list-level comparable to LISP as an underlying data structure for
MoDL. Our compiler translates list-level data structures (especially
functions) into C after which the C-compiler is activated.

Advantages of the new MoDL implementation compared to the old
one are:

1. The MoDL list-level and MoDL language-level use all the same
symbols for their operators. This makes the implementation
much more clean, and it eliminates the many synonyms, such
as less for < and so on, which merely confused our users.

2. Our C-based kernel proves to be rather portable, as it runs on
quite different machine architectures, such as register
machines as well as on stack machines.

3. The compilation of the MoDL sources used in the system
kernel, greatly improved the speed of the basic MoDL system.

4. List-level MoDL code can be loaded very fast due to the fast
implementation of the readlist function, which takes
advantage of the hashed object list organisation, providing a
convenient and fast storage for partially compiled user
programs.

5. Macros now serve to support the compiler exclusively,
subroutines implement all special cases which are needed for
the interpreter. The old implementation did (have to) use
macro expansion during execution of the interpreter, which
took away a considerable part of the available processing
capacity due to redundant invocations of cons.

6. MoDL can compile functions, subroutines and macros with a
fixed or a variable number of arguments.

7. MoDL takes considerably less space for its compiled kernel,
leaving more space for interpretative programs and data. This
in turn results in a decreased amount of invocations of the
garbage collector, and hence in a faster execution.

8. Linkage of programs written in another language is relatively
easy, as MoDL is totally C-based.
IV.2 Calling the Compiler

Calling the compiler is very simple: it is activated with the function `transload` described in part II of the manual, specifying o (for object code) as the extension of the output file. Both files in MoDL syntax and list syntax can be compiled with the commands:

```mondl
transload(<file name>, 'modl', 'o);
transload(<file name>, 'list', 'o');
```

Compiled files can be loaded in a run-time system with the `load` command described in part II.
IV.3 Compiler Directives

For the correct compilation of MoDL programs and for reasons of efficiency it is sometimes necessary to provide the compiler with additional information. This information is contained in the arguments of the pseudo-function:

declare (<arg1>, ..., <argn>), which has no effect when evaluated by the interpreter. The arguments are one of the following pseudo-function calls, also denoted compiler directives (all of them have a variable number of arguments):

1. special (<var1>, ..., <varn>), has arguments of the type truename. The directive tells the compiler, that the truenames seen as variables have to have a dynamic scope after compilation. This is exactly the scope of variables in an interpreted environment. If a variable is not declared special, it has a static scope, which is the usual scope of variables in languages as C and Pascal. A static scope leads to more efficient code.

2. FUN (<fun1>, ..., <funn>),
   SUB1..10 (<fun1>, ..., <funn>),
   SUB (<fun1>, ..., <funn>),
   MAC1..10 (<fun1>, ..., <funn>),
   MAC (<fun1>, ..., <funn>), have again arguments of the type truename. The directive is necessary because every call to a function not declared inside the module being compiled is assumed to be of one of the types FUN1..10. Other types have to be declared explicitly, such that they can be compiled correctly.

3. localf (<fun1>, ..., <funn>), has arguments of the type truename. The directive tells the compiler that the names seen as function names (of functions defined inside the module) need not be known outside the compiled module. All references to the functions listed will be made directly; local functions can only be referenced from inside the module being compiled. No truename will be created for the function name when the compiled code is loaded. This directive is not essential from the point of view of correctness; however efficiency of execution can be increased by it.

4. compiled (<fun1>, ..., <funn>), has arguments of the type truename. The directive tells the compiler that the functions mentioned will be available in compiled form inside the runtime system. This means that the compiler can call such a function directly via the function field of the truename.
instead of calling *apply*, which is done in the general case. Kernel functions (all functions mentioned in part II of the manual) are known to the compiler and the user need not to bother with them. So, this directive is meant for user-written functions compiled in a separate module. Clearly, this directive only increases efficiency.
V. Index for the Parts II, III and IV

<table>
<thead>
<tr>
<th>function name</th>
<th>page</th>
<th>function name</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'</code></td>
<td>II-9-1</td>
<td><code>bignum</code></td>
<td>II-2-1</td>
</tr>
<tr>
<td><code>.</code></td>
<td>II-9-1</td>
<td><code>bool</code></td>
<td>III-5-6</td>
</tr>
<tr>
<td><code>*</code></td>
<td>III-3-1</td>
<td><code>bound</code></td>
<td>II-5-2</td>
</tr>
<tr>
<td><code>+</code></td>
<td>II-7-1</td>
<td><code>break</code></td>
<td>II-9-13</td>
</tr>
<tr>
<td><code>-</code></td>
<td>II-7-1</td>
<td><code>capacitance</code></td>
<td>III-4-2</td>
</tr>
<tr>
<td><code>/</code></td>
<td>II-7-1</td>
<td><code>case</code></td>
<td>II-9-6</td>
</tr>
<tr>
<td><code>:</code></td>
<td>II-5-2</td>
<td><code>case</code></td>
<td>III-3-1</td>
</tr>
<tr>
<td><code>:=</code></td>
<td>III-3-1</td>
<td><code>catch</code></td>
<td>II-9-8</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>II-4-2</td>
<td><code>charcnt</code></td>
<td>II-6-4</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>II-7-2</td>
<td><code>close</code></td>
<td>II-6-3</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>II-4-3</td>
<td><code>concat</code></td>
<td>IV-3-1</td>
</tr>
<tr>
<td><code>=</code></td>
<td>II-4-1</td>
<td><code>cons</code></td>
<td>II-5-9</td>
</tr>
<tr>
<td><code>==</code></td>
<td>II-4-1</td>
<td><code>copy</code></td>
<td>II-3-3</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>II-4-2</td>
<td><code>cos</code></td>
<td>II-7-2</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>II-4-2</td>
<td><code>datacase</code></td>
<td>III-4-2</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>II-7-2</td>
<td><code>datatype</code></td>
<td>II-2-3</td>
</tr>
<tr>
<td><code>and</code></td>
<td>III-5-3</td>
<td><code>declare</code></td>
<td>IV-3-1</td>
</tr>
<tr>
<td><code>_nand</code></td>
<td>III-5-3</td>
<td><code>def</code></td>
<td>II-5-6</td>
</tr>
<tr>
<td><code>nor</code></td>
<td>III-5-3</td>
<td><code>delay</code></td>
<td>II-4-1</td>
</tr>
<tr>
<td><code>not</code></td>
<td>III-5-3</td>
<td><code>delay</code></td>
<td>III-7-1</td>
</tr>
<tr>
<td><code>or</code></td>
<td>III-5-3</td>
<td><code>demandcase</code></td>
<td>III-4-2</td>
</tr>
<tr>
<td><code>_test</code></td>
<td>III-5-4</td>
<td><code>divide</code></td>
<td>II-7-1</td>
</tr>
<tr>
<td><code>_xor</code></td>
<td>III-5-3</td>
<td><code>driver</code></td>
<td>II-9-10</td>
</tr>
<tr>
<td><code>abs</code></td>
<td>II-7-2</td>
<td><code>dt</code></td>
<td>II-4-2</td>
</tr>
<tr>
<td><code>abut</code></td>
<td>III-6-1</td>
<td><code>dt</code></td>
<td>III-7-2</td>
</tr>
<tr>
<td><code>acos</code></td>
<td>II-7-3</td>
<td><code>empty</code></td>
<td>II-2-1</td>
</tr>
<tr>
<td><code>allocvec</code></td>
<td>II-6-1</td>
<td><code>encode</code></td>
<td>III-5-2</td>
</tr>
<tr>
<td><code>and</code></td>
<td>II-6-3</td>
<td><code>exec</code></td>
<td>II-9-15</td>
</tr>
<tr>
<td><code>append</code></td>
<td>II-3-3</td>
<td><code>exitif</code></td>
<td>II-9-7</td>
</tr>
<tr>
<td><code>apply</code></td>
<td>II-9-2</td>
<td><code>explode</code></td>
<td>III-3-4</td>
</tr>
<tr>
<td><code>aseii</code></td>
<td>II-6-9</td>
<td><code>explodenum</code></td>
<td>II-5-7</td>
</tr>
<tr>
<td><code>asin</code></td>
<td>II-7-3</td>
<td><code>first</code></td>
<td>II-5-6</td>
</tr>
<tr>
<td><code>assoc</code></td>
<td>II-3-5</td>
<td><code>fixnum</code></td>
<td>II-3-4</td>
</tr>
<tr>
<td><code>assq</code></td>
<td>II-3-5</td>
<td><code>flonum</code></td>
<td>II-2-1</td>
</tr>
<tr>
<td><code>atan</code></td>
<td>II-7-3</td>
<td><code>for</code></td>
<td>II-2-2</td>
</tr>
<tr>
<td><code>atom</code></td>
<td>II-2-2</td>
<td><code>foreach</code></td>
<td>II-9-8</td>
</tr>
</tbody>
</table>

page V-0-1
Index for the Parts II, III and IV

<table>
<thead>
<tr>
<th>function name</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>forexp</td>
<td>II-9-8</td>
</tr>
<tr>
<td>fork</td>
<td>II-9-15</td>
</tr>
<tr>
<td>fourth</td>
<td>II-3-5</td>
</tr>
<tr>
<td>gc</td>
<td>II-9-9</td>
</tr>
<tr>
<td>gensym</td>
<td>II-5-9</td>
</tr>
<tr>
<td>get</td>
<td>II-5-5</td>
</tr>
<tr>
<td>getdef</td>
<td>II-5-6</td>
</tr>
<tr>
<td>getmacro</td>
<td>II-8-9</td>
</tr>
<tr>
<td>getmodeldef</td>
<td>III-2-1</td>
</tr>
<tr>
<td>getsyntax</td>
<td>II-8-8</td>
</tr>
<tr>
<td>getvec</td>
<td>II-6-2</td>
</tr>
<tr>
<td>implode</td>
<td>II-5-8</td>
</tr>
<tr>
<td>inecho</td>
<td>II-8-2</td>
</tr>
<tr>
<td>infile</td>
<td>II-8-2</td>
</tr>
<tr>
<td>inplacereverse</td>
<td>II-3-9</td>
</tr>
<tr>
<td>integer</td>
<td>II-2-2</td>
</tr>
<tr>
<td>kill</td>
<td>II-9-14</td>
</tr>
<tr>
<td>length</td>
<td>II-2-4</td>
</tr>
<tr>
<td>list</td>
<td>II-3-1</td>
</tr>
<tr>
<td>load</td>
<td>II-8-14</td>
</tr>
<tr>
<td>localf</td>
<td>IV-3-1</td>
</tr>
<tr>
<td>log</td>
<td>II-7-3</td>
</tr>
<tr>
<td>loop</td>
<td>II-9-7</td>
</tr>
<tr>
<td>loop</td>
<td>III-3-4</td>
</tr>
<tr>
<td>max</td>
<td>II-7-2</td>
</tr>
<tr>
<td>member</td>
<td>II-2-3</td>
</tr>
<tr>
<td>memq</td>
<td>II-2-3</td>
</tr>
<tr>
<td>min</td>
<td>II-7-2</td>
</tr>
<tr>
<td>mod</td>
<td>II-7-1</td>
</tr>
<tr>
<td>modeldef</td>
<td>III-2-1</td>
</tr>
<tr>
<td>name</td>
<td>II-2-3</td>
</tr>
<tr>
<td>newln</td>
<td>II-8-14</td>
</tr>
<tr>
<td>nlog</td>
<td>II-7-3</td>
</tr>
<tr>
<td>not</td>
<td>II-4-3</td>
</tr>
<tr>
<td>nthnode</td>
<td>II-3-6</td>
</tr>
<tr>
<td>number</td>
<td>II-2-3</td>
</tr>
<tr>
<td>oblist</td>
<td>II-3-2</td>
</tr>
<tr>
<td>or</td>
<td>II-4-4</td>
</tr>
<tr>
<td>ord</td>
<td>II-5-10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function name</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>outecho</td>
<td>II-8-2</td>
</tr>
<tr>
<td>outfile</td>
<td>II-8-3</td>
</tr>
<tr>
<td>pair</td>
<td>II-2-2</td>
</tr>
<tr>
<td>pins</td>
<td>III-4-1</td>
</tr>
<tr>
<td>plist</td>
<td>II-5-5</td>
</tr>
<tr>
<td>pnewln</td>
<td>II-8-14</td>
</tr>
<tr>
<td>pop</td>
<td>II-5-3</td>
</tr>
<tr>
<td>port</td>
<td>II-8-2</td>
</tr>
<tr>
<td>pprintatom</td>
<td>II-8-14</td>
</tr>
<tr>
<td>pprintlist</td>
<td>II-8-14</td>
</tr>
<tr>
<td>preadatom</td>
<td>II-8-5</td>
</tr>
<tr>
<td>preadchar</td>
<td>II-8-5</td>
</tr>
<tr>
<td>preadlist</td>
<td>II-8-10</td>
</tr>
<tr>
<td>printatom</td>
<td>II-8-12</td>
</tr>
<tr>
<td>printlist</td>
<td>II-8-12</td>
</tr>
<tr>
<td>pulldown</td>
<td>III-5-6</td>
</tr>
<tr>
<td>pullup</td>
<td>III-5-6</td>
</tr>
<tr>
<td>push</td>
<td>II-5-3</td>
</tr>
<tr>
<td>put</td>
<td>II-5-4</td>
</tr>
<tr>
<td>quit</td>
<td>II-9-14</td>
</tr>
<tr>
<td>read</td>
<td>II-8-11</td>
</tr>
<tr>
<td>readatom</td>
<td>II-8-5</td>
</tr>
<tr>
<td>readchar</td>
<td>II-8-5</td>
</tr>
<tr>
<td>reader_table</td>
<td>II-8-7</td>
</tr>
<tr>
<td>readlist</td>
<td>II-8-10</td>
</tr>
<tr>
<td>removename</td>
<td>II-9-9</td>
</tr>
<tr>
<td>remprop</td>
<td>II-5-6</td>
</tr>
<tr>
<td>replacefirst</td>
<td>II-3-7</td>
</tr>
<tr>
<td>replacerest</td>
<td>II-3-6</td>
</tr>
<tr>
<td>rest</td>
<td>II-3-4</td>
</tr>
<tr>
<td>second</td>
<td>II-3-4</td>
</tr>
<tr>
<td>set</td>
<td>II-5-1</td>
</tr>
<tr>
<td>setmacro</td>
<td>II-8-10</td>
</tr>
<tr>
<td>setplist</td>
<td>II-5-6</td>
</tr>
<tr>
<td>setprotect</td>
<td>II-6-4</td>
</tr>
<tr>
<td>setsyntax</td>
<td>II-8-9</td>
</tr>
<tr>
<td>setunbound</td>
<td>II-5-2</td>
</tr>
<tr>
<td>setvec</td>
<td>II-6-1</td>
</tr>
<tr>
<td>shell</td>
<td>II-8-13</td>
</tr>
</tbody>
</table>

page V-0-2

6.160
Index for the Parts II, III and IV

<table>
<thead>
<tr>
<th>function name</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal</td>
<td>II-9-15</td>
</tr>
<tr>
<td>signedbool</td>
<td>III-5-5</td>
</tr>
<tr>
<td>sin</td>
<td>II-7-2</td>
</tr>
<tr>
<td>special</td>
<td>IV-3-1</td>
</tr>
<tr>
<td>sqrt</td>
<td>II-7-3</td>
</tr>
<tr>
<td>strength</td>
<td>III-4-2</td>
</tr>
<tr>
<td>support</td>
<td>III-4-1</td>
</tr>
<tr>
<td>system</td>
<td>II-9-14</td>
</tr>
<tr>
<td>tan</td>
<td>II-7-2</td>
</tr>
<tr>
<td>third</td>
<td>II-3-4</td>
</tr>
<tr>
<td>throw</td>
<td>II-9-8</td>
</tr>
<tr>
<td>time</td>
<td>II-9-14</td>
</tr>
<tr>
<td>timing</td>
<td>III-7-1</td>
</tr>
<tr>
<td>toplevel_driver</td>
<td>II-9-9</td>
</tr>
<tr>
<td>trace</td>
<td>II-9-8</td>
</tr>
<tr>
<td>transload</td>
<td>II-8-14</td>
</tr>
<tr>
<td>truename</td>
<td>II-2-2</td>
</tr>
<tr>
<td>untrace</td>
<td>II-9-9</td>
</tr>
<tr>
<td>vector</td>
<td>II-6-1</td>
</tr>
</tbody>
</table>
VI. The Description of the NEC uPD7720 in MoDL

VI.1 Introduction

To show the power of MoDL as a hardware description language on a non-trivial design, an existing micro-processor architecture has been modelled in MoDL. The target architecture has been the NEC uPD 7720, which is a signal processor interface. This chip is mainly intended for telecommunication applications. All information used for the description has been derived from the verbal product description supplied by the manufacturer.

The description level regarded here is the high level only. Several aspects motivated this choice:

- A high level description results in a precise design definition and gives the opportunity for an early evaluation of the product.
- In general, microcode designers prefer to "play" with such complicated designs as the uPD 7720 in order to find design flaws (e.g., bus conflicts) or performance bottle necks (critical paths) in an early stage of the design process.
- Hardware design, considered as a hierarchical process, starts with a high level product definition.
The NEC uPD 7720 is a complete 16-bit microcomputer on a single chip. While the on-chip RAM may be used for temporary data/coefficients and results, computational power is provided by a 16-bit Arithmetic Logic Unit (ALU) and a separate 16*16-bit fully parallel multiplier.

Some features of the NEC uPD 7720 are:
- fast instruction execution (250 ns / 8 MHz clock)
- 16-bit datawords
- multi-operation instructions for optimizing program execution
- instruction ROM (512 * 23 bits)
- data/coefficient ROM (510 * 13 bits)
- data RAM (128 * 16 bits)
- fast 16 * 16 to 31-bit parallel multiplier
- dual accumulators.

A block diagram of the processor is figure VI.2.1.

Figure VI.2.1. The architecture of the uPD 7720.

The top-level description of the chip given below, has a close correspondence to the block diagram.
The description consists of two parts, one part is activated by the "reset" signal, the other by its negation. The latter part is the most essential: in it one can distinguish the submodels in which the processor has been divided. Some of these submodels will be discussed in detail in the next section. Note also, the loop ... endloop construct, that provides the registers in the circuit with a clock.
VI.3 Some Submodels

In this section the description of some of the submodels of the signal processor circuit will be presented.

VI.3.1 The Multiplier

Below the MoDL description of the multiplier has been given:

```
MODEL multiply (k, l, m, n),
% This MODEL describes the multiplier and the m and n registers of the processor. The result of the multiplication (31 bits)
% is stored in the m and n registers.
% input signals : k, l.
% output signals : m, n.
% local signals : mm.
% support functions: multiplier, high16b, low16b.

mn: multiplier(k, l),
m: high16b(mn),
n: low16b(mn).
ENDMOD;
```

This description is still a high level one; therefore the main actions, the multiplication itself and the selection of the highest and lowest 16 bits of the result are represented by functions.

VI.3.2 The K-register

Below the MoDL description of the K-register has been given:

```
MODEL kreg (src, dst, ramout, k, idb),
% This MODEL describes the data transport between the k register and the internal data bus (idb).
% control signals: src, dst.
% input signals : ramout, k, idb.
% output signals : k, idb.

idb : CASE src = 'K
           {k},
        ENDCASE,

k := CASE dst = 'KLM
          member(dst, '(K KLR))
          {ramout},
        ENDCASE,
ENDMOD;
```

This example features the possibility of having two-directional signals in MoDL. Depending on the value of control signals, the model "kreg" either writes a new value on the internal data bus ("idb") or reads from it.
VI.3.3 The ALU and peripherals

A block diagram of the ALU is shown in figure VI.3.1.

Figure VI.3.1. Block diagram of the ALU and peripherals

This circuit can be described with the following MoDL code:
The Description of the NEC uP7720

Some Submodels

MODEL ALU (asl, alu, dst, src, p, mpzout, acca, accb, idb, flaga, flagb, sgn).

% This model describes the arithmetic logic unit of the processor. It describes the alu operations to be carried out (controlled by the signal "alu"). The result is stored in acca or accb. Data transport between the accumulators and the internal data bus is described and controlled here (note: the alu operation is ignored when dst=asl). Finally the flags of the accumulators are updated.

% control signals : alu, dst, src, asl.
% input signals : p, mpzout, idb, sgn.
% output signals : acca, accb, idb, flaga, flagb.
% local signals : q, cond, aluout.
% support functions : Or, And, Xor, Plus, Min, sbb, adc.
% dec, inc, cmp, updateflag.

q, mpzout.

CASE cond
CASE dst = 'AccA' {acca := idb},
dst = 'AccB' {accb := idb},
ENDCASE
not cond
aluout : CASE alu = 'Or' {Or(p, q)}.
alu = 'And' {And(p, q)}.
alu = 'Xor' {Xor(p, q)}.
alu = '+' {Plus(p, q)}.
alu = 'Min' {Min(p, q)}.
alu = 'Sbb' {sbb(p, q)}.
alu = 'Add' {ade(p, q)}.
alu = 'Dec' {dec(q)}.
alu = 'Inc' {inc(q)}.
alu = 'Cmp' {cmp(q)}.
member(alu, 
'SH1 SH2 SH4 XCHG'))
ENDCASE.
CASE asl = 'AccA' {acca := aluout, 
updateflag{flaga}},
asl = 'AccB' {accb := aluout, 
updateflag{flagb}},
ENDCASE.

ENDCASE.
CASE src = 'AccA' {acca},
src = 'AccB' {accb},
ENDCASE.
ENMOD.

One of the interesting possibilities of MoDL has been shown in this description, viz. the use of symbolic values for the signal "alu". Its values can select the function to be performed in the ALU. If the
use of symbolic values had not been possible, the values would have to be encoded in an integer. This would decrease the readability of the description and increase the possibility of introducing errors.
VI.4 The Presentation of Results

For simulation purposes, a user interface has been implemented, interactively showing the simulation results within the (symbolic) floor plan of the chip (figure VI.4.1). The hardware description has been tested and verified with a 7th order digital low pass filter.

![Symbolic floor plan of the uPD 7720 Signal Processor chip](image)

On it all essential signals on the chip are visible; this makes it very easy for the user to follow the signal flow in subsequent clock periods.
VI.5 Remarks and Conclusions

The most important conclusions were the following:

- **MoDL as a hardware description language.** MoDL is found to be a very user-friendly and powerful hardware description language. As far as hardware description is concerned, it is evident that MoDL gives a far more precise definition of a design than a verbal description.

- **Simulation results.** The MoDL description of the target architecture can easily be evaluated in the application domain. The model executes symbolic instruction codes, leaving open all possible control signal coding on bit level. The results were exactly correct and the simulation was quite fast, even in interpretative mode.

- **Simulation Environment.** It has been recommended to implement the simulation support functions as a standard support function library in the MoDL design environment.

- **Future Research.** Many HDL's have been developed in the past, and surely others will be in the future. MoDL has to satisfy two main constraints in order to become a success:
  
  - Apart from being a powerful description language, MoDL must include user-friendly and convenient simulation I/O features and accessible hardware design structures.
  - MoDL must be provided with indispensable hardware design features which are not provided in competitive HDL's (e.g., expert system facilities and formal verification tools such as a mutual exclusion prover for conditional signal labelings).

Particularly this second aspect is of vital importance and therefore demands much attention. Much research has already been done in this area and will be continued in the future.
Chapter 7
Gate Array Design
The architecture of an open design system for gate arrays.

J.A.C. Jess.

Eindhoven University of Technology.

1. Introduction.

Gate arrays are semicustomized chips intended for the realization of digital systems. The term "semicustomized" means that in principle the structure and arrangements of the various logic and interface components on a chip are fixed for all digital systems realized by a family of gate arrays. The realization of a particular digital network is obtained by designing the wiring between the various components.

Nowadays the main advantage of the use of gate arrays is a quick turn around that can be obtained during fabrication. Since the majority of masks for a gate array exhibits fixed patterns wafers can be prefabricated. Per design only the wiring tracks must be individually fabricated. Also the number of fabrication steps that must be completed between submittal of the wiring pattern and the final completion of the chip fabrication is small compared to the fabrication of a completely customized integrated circuit.

The design of the wiring patterns for gate arrays is, contrary to what many people think, fairly complex. The reason is the rigidity of the chip pattern. Any wiring channel has a fixed number of connections it can accommodate. In custom integrated circuits we match the width of channels to the number of tracks required. In gate array design whenever the number of tracks is not sufficient in some channel area we must use complex rerouting strategies to resolve such congestion problems. Since gate arrays are relatively small as compared to VLSI custom chips digital systems must be distributed over various chips. Since every designer wants as few as possible chips the temptation

* This paper has been produced in the context of TASK IX of the project MR-09 supported by the Commission of the EEC, under the 3744/81 Microelectronics program. Partial support by the Ministry of Economic Affairs of the Netherlands under the NELSIS program is hereby also acknowledged.
exists to squeeze too many functions onto one gate array chip. This results in endless design cycles be they manual or automatic. On the other hand using too many chips for one system implies waste of real estate on silicon. This situation caused a lot of research concerning wirability predictions. Nonetheless gate arrays are generally not too economic with respect to real estate usage.

2. Gate array families.

Gate arrays are offered in a wide variety. On one hand they differ a lot as to the logic families they implement. There are gate array types based on all kinds of logic families like I2L, (cascode-) ECL, Schottky TTL, static NMOS and CMOS. A second feature of a gate array is its so called image. The image is composed from the prefabricated gate structures, the possible locations of wires on various polysilicon, silicide or metal wiring layers and the possible locations of via holes establishing connections between the various wiring layers. The spectrum of images reaches from the one metal layer-fixed-poly underpath-static CMOS type with a row oriented arrangement of gates to the two-metal-layer-programrnable via-Schottky TTL type with a block arrangement of gates. Examples of those images are given in the subsequent contributions of this chapter.

The problem that arises for the designer or Silicon broker is that any family of gate arrays seems to require its own specific design software. Logic synthesis must be taylored to the logic family of a gate array. Any image requires its personal place and rout algorithms.

A further dimension of complexity is introduced by the use of family specific macros. Those macros are larger functions like specific high fan-in or fan-out gates, various types of flip-flops, shift registers and counters. They are characterized by fixed standardized wiring patterns ("stamps"). By this token there are limitations to the legal positions of macros on the gate array chip. Also their use implies a still more reduced economy of silicon area usage. However, their use is often preferred by designers because it saves design time. Moreover manufacturers often guarantee timing and power features of those macros.

Of course established vendors of gate arrays have the advantages of this situation. They bind their client to their gate array type in particular if the client decided to invest much into the family specific design automation technology. Also new vendors are kept out of the market because clients stick
to the established products, which cost them so much to get operational for themselves. This whole situation is particularly threatening for European designers and silicon brokers.

3. 'Open' design systems.

The project we are reporting about here is a first step to alleviate the situation described above by providing an "open" design system for gate arrays. Basically such a system consists of a frame of data structures called "grids" in our jargon. The grids are used to store the design information at any stage of the design process. Another data structure contains the so-called "routing primitives".

We call the system "open" because of three essential features enhancing the flexibility of the system. Firstly the system can be adapted to any image in the following way: a language is defined in order to describe the image and the rules for routing on that image. An extended form of the language can be made powerful enough to also handle macros. A compiler is provided to translate the image, routing rules and macro descriptions into a grid, a set of routing primitives and a macro library.

The information present in those data structures can be used by any design program for placement and routing. The system may contain a large variety of those programs. This variety can go through a long history of evolution until a sufficiently powerful configuration is obtained. Thus in the second place the system is open in the sense that it permits evolution in design methodology.

A third aspect of design is the logic synthesis. The success of any gate array design depends on the tools for logic design and in particular on their flexibility. In this project, however, we have limited our scope to the first two aspects of design namely the adaptivity to various images and the construction of an initial set of place and route tools. The logic synthesis part is a subject that can be studied independently of the place and route problem. Moreover it is a very difficult problem to tackle and requires a couple of manyears at the least in addition to what is spent on this project.

4. The structure of the ICD-system GAS.

The architecture of our system is given in Fig. 1. Basically the system has four different grids. "GRID" is generated by COMPILER1 and is in fact a complete description of the unpersonalized gate array, complete enough to steer all place and route actions, but compact enough to fit into the work memory of
a large workstation. "GRID*" and "GRID**" store intermediate results of place
and route. "GRID***" finally contains the complete information to generate the wiring and via hole masks. The "ROUTING PRIMITIVES" control the local routing and in a much stronger way than the global routing. In principle the "ROUTING PRIMITIVES" together with the information in "GRID" make it possible to construct and evaluate all possible extensions of a net at any arbitrary grid point. Nets may be extended in all three co-ordinates depending on the structure of the image and the occupation of neighbouring gridpoints by other nets or obstacles of various kinds. Nets may also be extended by polyunderpaths be they fixed or programmable. Whichever is the case is part of the stored information in "GRID".

In the system it makes a difference whether the options for possible extensions are checked or whether a net is actually extended along a certain gridline. Also updating processes are invoked in the event that two subtrees of a certain net are merged by a net extension. On the other hand traces of a net can be extinguished. If by this process a tree is split into various subtrees the identification of the subtrees is properly adjusted. Actually the routing primitives are nothing but a set of commands which are conditioned by COMPILER1 so as to match the image to be processed. Any routing procedure can use these primitives. A list of those routing primitives is given in the next figure.
<table>
<thead>
<tr>
<th>Rout primitive</th>
<th>arguments</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEH(i,j,k)</td>
<td>i,j,k = grid coordinates</td>
<td>returns a list of possible extensions of a net from location i,j within layer k together with the cost of those extensions</td>
</tr>
<tr>
<td>TEV(i,j,k)</td>
<td>i,j,k = grid coordinates</td>
<td>the same as TEH however in vertical direction</td>
</tr>
<tr>
<td>EH(i,j,k,dir)</td>
<td>i,j,k = grid coordinates, dir = north,east,south,west</td>
<td>performs an actual extension of a net into the desired direction</td>
</tr>
<tr>
<td>EV(i,j,k,dir)</td>
<td>i,j,k = grid coordinates, dir = down,up</td>
<td>the same as EH however in vertical direction</td>
</tr>
<tr>
<td>DEH(i,j,k,dir)</td>
<td>as EH</td>
<td>Delete dangling edge in layer k from subtree</td>
</tr>
<tr>
<td>DEV(i,j,k,dir)</td>
<td>as EV</td>
<td>Delete dangling via from subtree</td>
</tr>
<tr>
<td>DIH(i,j,k,dir)</td>
<td>as EH</td>
<td>Delete inner edge from subtree (creates new subtree)</td>
</tr>
<tr>
<td>DIV(i,j,k,dir)</td>
<td>as EV</td>
<td>Delete inner via from subtree (creates new subtree)</td>
</tr>
<tr>
<td>DST(i,j,k,dir)</td>
<td>i,j,k = grid coordinates, dir = north,east,south,west,down,up</td>
<td>Deletes subtree starting at i,j,k in direction dir</td>
</tr>
<tr>
<td>FLS(i,j,k)</td>
<td>as TEH</td>
<td>Finds leaves of subtree occurring location i,j,k</td>
</tr>
<tr>
<td>TEHS(i,j,k)</td>
<td>as TEH</td>
<td>same as TEH, however reports if a different subtree of the same net is encountered</td>
</tr>
<tr>
<td>TEVS(i,j,k)</td>
<td>as TEV</td>
<td>same as TEV, however reports if a different subtree of the same net is encountered</td>
</tr>
<tr>
<td>EHS(i,j,k,dir)</td>
<td>as EH</td>
<td>same as EH except if a different subtree of the same net is encountered then the two subtrees are joined</td>
</tr>
<tr>
<td>EVS(i,j,k,dir)</td>
<td>as EV</td>
<td>corresponds to EHS for extensions by down edges</td>
</tr>
</tbody>
</table>

Figure 2. List of routing primitives.

At the time of writing the "MACRO LIBRARY" is still an item under study. A module description language has tentatively been defined. However, "COMPILER2" and structure of "MACRO LIBRARY" have still to be established.

As has been pointed out earlier the question of logic synthesis has not yet been approached in this project, although we have developed logic synthesis systems for custom IC's. In the design system existing now we assume that a netlist will have been constructed before the personalization of the gate array.
will be started. This netlist is a list of signal nets. For any signal net the set of macros it contacts is given (including the identifications of the pins the net is connected to).

At the time of writing the following program modules have been incorporated into the system:

1. placement for equal macros by simulated annealing;
2. placement of unequal macros by simulated annealing;
3. global routing by Burstein's "2x2" integer programming method and his "2xN" dynamic programming method;
4. local routing by a variant of the Lee routing algorithm.

A combination of the min-cut-placement and Burstein's global routing method approaches its completion rapidly. Interesting enough this last method bypasses "GRID*" and establishes "GRID**" directly.

5. Conclusion.

The contributions that follow give more details about the systems. Also some performance data are given although these have a very tentative character. Being an "open" system and subject to evolution the system will achieve increasing performance in the first years of practical use. However the flexibility we claim is already achieved now. In a day of preparation the system can be conditioned to handle a new gate array family. Although the speed of placement and routing may not be optimum right after grid compilation, many months of programming or equivalently many thousands of dollars are saved and a point to be able to design a new family is reached almost instantaneously. The results obtained now prove the worthiness of the approach and the potential usefulness to European designers and silicon brokers.
1. Introduction

Gate arrays have been used in IC designs for fast turnaround time for many years. The great interest in gate arrays is actually a recent phenomenon due to the rapid progress both in the microelectronics and in IC applications. The numbers of gate array manufactures and therefore the types of gate arrays have also increased considerably. Gate arrays are designed in many technologies (NMOS, CMOS, TTL, IIL etc.). CAD systems for gate arrays have to face the new problems caused by the rapid growth of the gate array market. One problem is the gate array entry to a system (supposing that the system can handle different type of gate arrays). Although there are several gate array placement and routing systems commercially available, they require a lot of work to enter a given gate array in the system, or they may remove a lot of efficiency from the router [1]. There is still a great need for a flexible and easy way to enter a gate array into a CAD system.

To make our gate array CAD system [2] applicable to a great variety of gate arrays, we have developed a gate array description language GADSL to fulfill the task. GADSL describes the structure and the properties of the gate arrays that will be used by placement and routing programs. As shown in the following sections, both the structure and the properties of gate arrays can be properly characterized by "personalized" grids. The features provided by a gate array that are important for routers are covered by grid lines and/or grid points such as terminals, vias and connections.

GADSL is developed in parallel with a data structure for gate array routing and a description language for describing numerical gate array design rules. The data structure and design rule description language will not be mentioned in this report. The basic idea is to translate a gate array image and the associated design rules into the data structure which is then used by routers. The result obtained appears to be quite satisfactory. A large class of gate arrays and their features can be described in GADSL. A prototype compiler has

---

* This paper has been produced in the context of TASK IX of the project MR-09 supported by the Commission of the EEC, under the 3744/81 Microelectronics program. Partial support by the Ministry of Economic Affairs of the Netherlands under the NELSIS program is hereby also acknowledged.
been written for GADSL which generates gate array documentations from an input
description in the form of a data base, which will be then used by several
automatic tools.

In section 2, we model the gate arrays which GADSL should be capable to
describe. Section 3 gives the main features of GADSL. In GADSL, gate arrays
are described in terms of grids. Since the gate arrays have generally simple
structures, the concepts "array" and "repetition" are used rather than
"hierarchy". An example is given at the end of the section. As one can see
gridding and describing a gate array in GADSL is a fairly easy job. Finally,
appendix A gives the definition of the data structure used for each image grid.
The syntax of GADSL is listed in appendix B.

2. The Structure and Features of Gate Arrays

2.1 The overall structure

In principle a gate array consists of a regular arrangement of basic cells. Each
cell can be personalized by metalization to perform logic functions like
NAND, OR or even Flip-flop. These cells, usually referred to as function­
boxes, gate-boxes, active area etc., are placed in rows and then in a column
with pre-estimated wiring spaces between the rows and cells. The wiring spaces
between the rows are usually called channels or routing areas. Channels can
generally be divided into basic cells called connection-boxes (or junction­
boxes, channel cells). A connection-box can vary in complexity from just some
vertical and horizontal tracks for intercellular connections to a complex
structure with prefabricated connection paths (cross-unders) with fixed vias.

The area formed by regular repetition of gate-cells and connection-boxes is
mostly referred to as the core of a gate array image. In Figure 1, the image
core is shown as area I. Routing and placement are principally done in this
area. This area has a simple structure and hence easy to describe. In area
III shown in Figure 1, there are only bonding pads and input/output buffers
called I/O cells. The details of this area are not interesting for us except
for the connection possibilities to the I/O cells. This area is ignored in
GADSL descriptions except for the terminal positions. The area between the I/O
cells and image core, shown symbolically as area II in Figure 1, is used for
interconnections between the terminals within the image core and I/O cells.
This part of an image (when it is taken into account) should be treated
carefully since it does not have to be regular in structure.
2.2 The connection-boxes and gate-boxes

A rough sketch of the gate array structure is given above. Let us examine the cells in an image core. The connection-boxes have generally simple structures. Some typical connection-boxes shown in Figure 2 will give an indication of the complexity and structure of this kind of cells. The gate-cells of gate arrays are relative more complex since many trade-offs are made in these cell designs [3]. Depending on design philosophy, process technology, I/O capacities, and (most importantly) routability of the gate array, gate-cells may (and do) have quite different structures and properties. Wiring patterns in these cells are also more complicated. Two examples of CMOS gate-cells are given in Figure 3.
Figure 1.

Figure 2.
Although the (dedicated) gate arrays have different structures and are designed in different technologies, they have many points in common, even for those gate arrays which have not gate-cells and routing channels explicitly.

1. The intercellular connections can only be made vertically and/or horizontally on certain pre-defined tracks. From this, it is easy to define a grid where all the terminals and vias are positioned on grid points.

2. When "non-orthogonal" wires are permitted in gate-cells for realizing logical functions (the macros), they are either provided by gate array foundries or routed manually.

The fact that only those features of an image cell that are involved with interconnections have to be described simplifies the task significantly. Using a grid will be an appropriate method to do this. A grid is a set of vertical and horizontal lines. The grid lines do not need to be equi-distant. The grid should be defined in such way that:

a. any possible via and terminal position is covered by a grid point, and
b. (programmable) wires will be running along the grid lines.

On a grid, it is enough to characterize an image cell by specifying

1. The position of the fixed vias.
2. The fixed wire patterns (a set of grid edges).
3. The forbidden via positions.
4. The grid edges which are blocked for wiring.
5. For the pre-diffused non-orthogonal wire patterns (with the condition that the terminal positions are on grid points) we specify only their "terminal relations" rather than the exact wire patterns. For example, the exact wire patterns in the cells shown in Figure 3, can be ignored except the terminals. These terminals are related actually by an "electrical equivalence relation". A signal assignment (or in the term of routing, net assignment) to a terminal on a grid point propagates automatically to all the other grid points that are electrical equivalent with it. This is the important fact for routers to know. In this way, the fixed but not necessarily "simple" paths in an image cell can be specified.

2.3 Conclusion

An appropriate way to describe a gate array image is a grid. Gridding an image should be done in such way that all the features important for routings are covered by grid lines and/or grid points. The "equivalence relations" defined on a (sub)set of grid points can be used to describe the non-orthogonal features of the images. The concept "repetition", rather than "hierarchy" fits better to the gate arrays due to their simple structures. The area II in Figure 1 of a gate array, when it is taken into account, should be gridded carefully, since otherwise the irregular structure of this part may introduce a lot of "dummy" grid lines.

It should be emphasized that the whole study on gate array images is rather empirical. It is difficult to give an exact description and definition of the gate array images. We have developed the GADSL which is in principle capable to describe the gate arrays having the following properties.

1. The programmable interconnections can only be made in X- or Y-directions.
2. The features that are important for the routing of the gate array can be described with a grid.

In the next section, we shall discuss the GADSL in details.
3. The main features of GADSL

3.1 An outline of GADSL

GADSL describes gate arrays in terms of cells. Each cell description starts with a grid definition. Then there follow the detailed specifications. A two level cell description is implemented in GADSL. The cell at higher level should start with the keyword `IMAGE` and all the other cells should start with `IMCELL`. The two level refinement structure of GADSL is shown symbolically in Figure 4. The grid defined in an `IMAGE`-cell can be further specified by placing other detailed grids (IMCELL-cells) within it. From our experience, the complexity of a gate array image can be easily controlled by this two level description. GADSL encourages the users to map the structure of an image onto the structure of its GADSL description. Since when an image is properly described, its `IMAGE`-cell grid can be used directly for placing the macros.

![Figure 4.](image)

3.2 The cell definitions

In GADSL, an image description consists of a list of cell descriptions, starting with an "image cell description". In BNF form we define it as:

```
<cell list> := <image cell description> <cell description list>
<cell description list> ::= <empty> | <cell description list> <imcell description>
<image cell description> ::= "IMAGE" <cell identifier> <dimension> <layer declaration> <grid distance definition list> <statement list> "END"
<cell identifier> ::= <identifier>
<dimension> ::= "DIMENSION" "="
```

7.14
<number of columns> "*" <number of rows> ";"

<number of columns> := <integer>
<number of rows> := <integer>
<empty> :=

where <identifier> is a string of letters and digits starting with a letter.
<integer> is an integer.

The <imcell description> has the same syntax as the <image cell description> except that it starts with the keyword "IMCELL" instead of "IMAGE". The "cell placement statement" (see later) is not implemented in the <statement list> of IMCELL-cells. This means that the grids defined in IMCELL-cells are final grids.

3.3 The grid definition

After the keyword "IMAGE"/"IMCELL" and an user specified cell identifier, a grid should be defined and specified. The non-terminals <dimension>, <layer declaration>, and <grid distance definition list> in the above syntax are related to this part of specification. (see Appendix B for their definitions).

A grid definition starts with a dimension specification by giving the number of columns and rows of the grid. For example, the following statement

    dimension = 8 * 10;

defines a grid consisting of ten horizontal lines and eight vertical lines shown in Figure 5

![Figure 5](image)

The <layer declaration> declares all the (routable) layers. A layer is declared by giving the name of the layer together with the layer
"initialization". The layers should be declared in the same order as they appear on an image. The syntax of <layer declaration> is defined in BNF form as:

\[
\text{<layer declaration>} := \text{"LAYER" <layer list> ";"}
\]

\[
\text{<layer list>} := \text{<layer specifier>}
\]
\[
\text{<layer specifier>} := \text{<layer name> "("}
\]
\[
\text{"VIA" ":" <via type> ","}
\]
\[
\text{"WIRE" ":" <wire type> ")"}
\]

\[
\text{<layer name>} := \text{<identifier>}
\]
\[
\text{<via type>} := \text{"FIX"}
\]
\[
\text{"PROG"}
\]
\[
\text{<wire type>} := \text{"FIX"}
\]
\[
\text{"PROG"}
\]

All the layers have the same dimension as defined by <dimension>. When a layer is declared, each grid edge is a potential wire segment and each grid point of that layer is a potential via connection position. A via can be used for a connection to the layer above (if there is any) or the layer below (if there is any). Depending on the <via type> of a layer, which is either "FIX" or "PROG", the grid points of the layer are initialized to the state "INHIBIT" or "PROGRAMMABLE". The edges of the layer are specified by <wire type>. When a grid point is in the state "INHIBIT", it means that a via can not be made from that position to the layer below it. For an edge the state "INHIBIT" means that the edge may not be used for interconnection. Statements in the <statement list> of a cell description may change the state of a grid point or a grid edge (see later). When a cell description is completed, via/wire can or can not be made during routing on a grid point/edge depending on whether its state is "PROGRAMMABLE" or "INHIBIT".

The <grid distance definition list> is a list of statements specifying the distances between the grid lines. The user is responsible for knowing the unit in which the distances are specified (in microns, Lambdas etc.). The distance between two vertical/horizontal lines is specified by giving the x/y coordinate of a vertical/horizontal grid line and an integer indicating the separation to the next grid line (which has as coordinate x/y increased by 1). Repetitions on the coordinates by fixed jumps are possible. This part may be omitted in an
IMAGE-cell description when the separation of the grid lines are implied in the underlying IMCELL-cell descriptions. All the spacings should be specified in the IMCELL-cell specifications.

3.4 The statement list

The "statement list" of a cell description is a list of primitive statements used for grid refinements and describing a cell in details. The primitive statements are:

1. cell placement statement;
2. wire statement;
3. wire inhibit statement;
4. via statement;
5. via inhibit statement;
6. (electrical) equivalence statement.

(See Appendix B).

3.4.1 The cell placement statement

This statement can be used in an IMAGE-cell description to place a sub-cell(s) (IMCELL-cells) in the IMAGE-cell grid while the sub-cell itself is defined elsewhere.

Syntax:

\[
\begin{align*}
\text{<cell placement statement>} & := \\
& "\text{AT"}\ "(" \text{<x coord}>"," \text{<y coord}>")" \\
& ;" \text{<cell placement}> ";" \\
\text{<cell placement>} & := \text{<cell identifier> \\
& \text{x-cell size} \text{<y-cell size> \\
& \text{<copy attributer> \\
& \text{<copy attributer>} & := \text{<empty> \\
& \mid \text{<copy x>} \\
& \mid \text{<copy y>} \\
& \mid \text{<copy x>} \text{<copy y>} \\
& \text{<copy x>} := \\
& "\text{CX"}\ "(" \text{<delta x>}"," \text{<number of repetitions}>")"
\end{align*}
\]
The \texttt{<copy attributer>} part of the statement is optional. The keywords "\texttt{cx}" and "\texttt{cy}" specify the repetition of the cell in x- and y-direction respectively. The \texttt{<delta x>} specifies the displacement of a cell in x-direction and \texttt{<delta y>} specifies the displacement in y-direction. The \texttt{<delta x>} and \texttt{<delta y>} are specified in terms of IMAGE-cell gridlines.

The use of the statement can be simply explained by the following example. A statement like:

\begin{verbatim}
  at( 1,2 ) : imagecell 2 3 cx( 1,2 ) cy( 2,1 );
\end{verbatim}

specifies a matrix of cells called "imagecell" in an IMAGE-cell grid starting at position (1,2) (see Figure 6). The "imagecell" in this statement has dimension 2*3 in the term of the IMAGE-cell grids.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{Figure6.png}
\caption{Figure 6.}
\end{figure}

A possible extension of the \texttt{<cell placement>} is to specify the "rotation" and/or "mirror" of a cell (see Appendix B). But at the moment they are not implemented.

3.4.2 wire statement

A wire statement specifies a prel-diffused electrical conducting path consisting of grid edges. A wire statement starts with the keyword "wire". Then in turn, the layer where it is found, the starting coordinate, the
positions where the connection bends and its end coordinates. For example, the statement

```
wire are specified (metall, (10,20), (40,20), (40,50));
```

describes a fixed connection path running from (10,20) to (40,20) and then to (40,50) on the layer called "metall". Note that the layer name should be declared in the <layer declaration> part.

### 3.4.3 wire inhibit statement

This statement is the counter-part of the <wire statement>. It is used to forbid the specified polygon consisting of grid edges used for interconnection. A <wire inhibit statement> starts with a keyword "nwire". For example, the statement

```
nwire( metall, (10,20), (10,30));
```

specifies the edges from (10,20) to (10,30) cannot be used for interconnection(s). The coordinate lists in <wire statement>'s and <wire inhibit statement>'s should describe orthogonal polygons running along grid lines on a specific layer.

### 3.4.4 via statement and via inhibit statement

These two kinds of statements are used to specify the positions of the fixed vias and the forbidden positions for the vias on an image. A <via statement> starts with the keyword "via" and a <via inhibit statement> starts with the keyword "nvia". In these statements, if two layer names are given, the vias or forbidden vias are found between the specified layers. If there is only one layer name being specified, then the vias or forbidden vias are found between the specified layer and the layer below it. The position of the vias are specified simply by giving their x- and y- coordinates within the cell. For example

```
via( metall.meta12 (10,20), (40,20));
via( metall (10,20), (40,20));
```

The first statement specifies two fixed vias between a layer called "metall" and a layer called "meta12" at position (10,20) and (40,20). If the layer "meta12" is the layer under the layer "metall", then the second statement will be identical to the first one. The names in the statements should be declared in the <layer declaration> part.
3.4.5 (electrical) equivalence statement

With this statement, a fixed connection path which is not running along the grid lines while the terminals are positioned at grid points can be described. As an example, the following statement describes the path shown in Figure 7. The dashed grid lines in Figure 7 are forbidden for the interconnections.

```plaintext
# Figure 7.

equivalent( (metal,3,11), (metal,6,11) );
nwire(metal, (3,11), (6,11) );
nwire(metal, (5,12), (5,10) );
nwire(metal, (4,12), (4,10) );
equivalent( (metal,3,10), (metal,4,10),
            (metal,5,10), (metal,6,10) );
wire(metal, (4,10), (5,10) );
nwire(metal, (3,10), (4,10) );
nwire(metal, (5,10), (6,10) );
```

If the edges of the layer are initialized to the state "INHIBIT", the "nwire" statements can be omitted.

A prototype compiler is written for GADSL. We store in principle a whole gate array as a three dimensional array of "vertices". The definition of a vertex is given in Appendix A. Each vertex of the array is corresponding to a grid point of the input gate array image. The <equivalence statement> is implemented by an additional table called "EQ-table". The "EQ-table" contains lists of (relative) coordinate pairs. Each list can be accessed by an integer index. In the output data structure, each vertex has a six bits field called "eq-index". This field is used to store the index to the "EQ-table". If the "eq-index" of a vertex say (x,y), is not zero, all the vertices that are equivalent with (x,y) can be found by adding the displacements stored in the
"EQ-table" to the grid point (x,y) (see Figure 8).

<table>
<thead>
<tr>
<th>index</th>
<th>number of items</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
<td>*ptr1</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>*ptr2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>...</td>
<td>*ptri</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N</td>
<td>...</td>
<td>*ptrn</td>
</tr>
</tbody>
</table>

ptri ---> | delta-x | delta-y | next | ---> ...

Figure 8.

4. An example

As an example, we give in Figure 10 a GADSL description of the image cell shown in Figure 3(a) in section 2. Let us call the cell "image_cell". Figure 9 shows the grid model of the "image_cell". The integers given between the grid lines indicate the distances between the grid lines in Lambdas. The grid points connected by the dashed lines are electrically equivalent. They are specified by <equivalence statement>’s in Figure 10. The power lines are omitted since they are not allowed to be used by routers. The edges to the vias of the power lines are inhibited. The "image_cell" has two layers called "m1" and "m2" respectively. The wiring patterns on the layer "m2" are fixed. Routers can only program the first layer "m1".

The compiler has a special feature called "layer reduction". For a gate array, when there are no programmable vias to the last layer, and all the wire patterns on the layer are fixed, in this case, the layer is "reduced". The fixed patterns are substituted by electrical equivalence relations determined by the (fixed) vias to the layer. In the example of Figure 9, the output data structure is a two dimensional array for the first layer, although there are two layers declared. This feature reduces the amount of data of a gate array significantly.
mage image_cell

dimension = 10 * 7;
layer ml( via: fix, wire: prog ),
m2( via: fix, wire: fix );

bdx( 1: ( 3 ), 2: ( 4 ), 3: ( 5 ), 4: ( 5 ), 5: ( 10 ),
       6: ( 10 ), 7: ( 5 ), 8: ( 4 ), 9: ( 3 )
);

bdy( 2: ( 2 ), 3: ( 4 ), defdist: ( 5 ) );

via( m2.m1, ( 1,4 ), cx( 9,1 ), cy( 1,3 ) );
via( m2.m1, ( 1,2 ), ( 3,1 ), ( 3,4 ), ( 3, 6 ) );
via( m2.m1, ( 4, 2 ), cx( 1, 3 ) );
via( m2.m1, ( 4, 5 ), ( 4, 7 ), ( 5, 6 ) );
via( m2.m1, ( 6, 4 ), ( 6, 5 ), ( 6, 7 ) );
via( m2.m1, ( 7, 5 ), ( 7, 7 ), ( 8, 1 ) );
via( m2.m1, ( 8, 4 ), ( 8, 6 ), ( 10, 2 ) );
equivalent( ( m2, 1, 7 ), ( m2, 6, 7 ), ( m2, 10, 7 ) );
equivalent( ( m2, 1, 6 ), ( m2, 5, 6 ), ( m2, 10, 6 ) );
equivalent( ( m2, 1, 4 ), ( m2, 6, 2 ), ( m2, 10, 2 ) );
equivalent( ( m2, 1, 2 ), ( m2, 5, 2 ) );
equivalent( ( m2, 6, 4 ), ( m2, 10, 4 ) );
equivalent( ( m2, 1, 5 ), ( m2, 6, 5 ), ( m2, 10, 5 ) );

nwire( ml, ( 2, 2 ), ( 2, 4 ) );
nwire( ml, ( 1, 3 ), ( 3, 3 ) );
nwire( ml, ( 8, 3 ), ( 10, 3 ) );
nwire( ml, ( 9, 2 ), ( 9, 4 ) );

end

Figure 10.

5. References


6. Appendix A : The Definition of The Data Structure VERTEX

The definition of a "vertex" is given below. Each "vertex" needs totally 48 bits to code the information.

The Definition of a "vertex"

<table>
<thead>
<tr>
<th>field</th>
<th>number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>test bit</td>
<td>1</td>
</tr>
<tr>
<td>net number</td>
<td>14</td>
</tr>
<tr>
<td>sub-tree number</td>
<td>6</td>
</tr>
<tr>
<td>design rule index</td>
<td>8</td>
</tr>
<tr>
<td>equivalent index</td>
<td>6</td>
</tr>
<tr>
<td>edge-south</td>
<td>1</td>
</tr>
<tr>
<td>edge-west</td>
<td>1</td>
</tr>
<tr>
<td>edge-up</td>
<td>1</td>
</tr>
</tbody>
</table>

7.23
7. Appendix B: The Syntax of GADSL

The syntax of GADSL.

<cell list> :- <image cell description>
    <cell description list>

<image cell description> :-
    IMAGE <cell identifier>
    <dimension>
    <layer declaration>
    <grid distance definition list>
    <statement list>
    END

<cell description list> :- <empty>
    | <cell description list>
    <imcell description>

<imcell description> :-
    IMCELL <cell identifier>
    <dimension>
    <layer declaration>
    <grid distance definition list>
    <statement list>
    END

<dimension> :-
    DIM(ENSION)? = <number of columns> * <number of rows> ;

<layer declaration> :- LAYER <layer list> ;

<layer list> :- <layer specifier>
    | <layer list>, <layer specifier>

7.24
<layer specifier> :=
<layer name> ( VIA : <via type> , WIRE : <wire type> )

<grid distance definition list> := <empty>
| <grid distance definition list>
  <grid distance definition>

<statement list> := <empty>
  | <statement list> <statement>

<grid distance definition list> :=
  BDX ( <gd statement list> ) ;
  | BDY ( <gd statement list> ) ;

<gd statement list> := <gd statement>
  | <gd statement list> , <gd statement>

<gd statement> := <row/column number> : ( <grid distance> )
  | <row/column number> : ( <grid distance> ,
    <translation specifier> )
  | DEFDIST : ( <grid distance> )

<translation specifier> :=
  REPEAT ( <delta x/y> , <numbers of repetitions> )

<statement list> := <wire statement> ;
  | <wire inhibit statement> ;
  | <via statement> ;
  | <via inhibit statement> ;
  | <equivalent statement> ;
  | <cell placement statement> ;

<wire statement> :=
  WIRE ( <layer name> , <position variable> ,
    <position list> <copy part> )

<wire inhibit statement> :=
  NWIRE ( <layer name> , <position variable> ,
    <position list> <copy part> )

<via statement> :=
  VIA ( <via layer specifier> , <position list>
    <copy part> )
<via inhibit statement> :=
NVIA ( <via layer specifier> , <position list>
       <copy part> )

<equivalent statement> :=
EQ(UIVALENT)? ( <eq. variable list> )

<cell placement statement> :=
AT ( <x-coord>, <y-coord> ) : <cell placement>

<via layer specifier> := <layer name> . <layer name>
| <layer name>

<position list> := <position variable>
| <position list> , <position variable>

<position variable> := ( <x-coord>, <y-coord> )

<eq. variable list> := <eq. variable>
| <eq. variable list> , <eq. variable>

<eq. variable> := ( <layer name>, <x-coord>, <y-coord> )

<cell placement> := <cell identifier>
   <x-cell size> <y-cell size>
   <mirror>
   <rotation>
   <copy attributer>

<mirror> := <empty>
| MX
| MY
| MX MY

<rotation> := <empty>
| R3
| R6
| R9

<copy attributer> := <empty>
| <copy x>
| <copy y>
| <copy x> <copy y>

<copy part> := <empty>

7.26
\[
\begin{align*}
\text{cell identifier} & := \text{identifier} \\
\text{number of rows} & := \text{integer} \\
\text{number of columns} & := \text{integer} \\
\text{row/column number} & := \text{integer} \\
\text{layer name} & := \text{identifier} \\
\text{via type} & := \text{FIX} \\
& \quad | \quad \text{PROG} \\
\text{wire type} & := \text{FIX} \\
& \quad | \quad \text{PROG} \\
\text{grid distance} & := \text{integer} \\
\text{delta x/y} & := \text{integer} \\
\text{numbers of repetitions} & := \text{integer} \\
\text{x- coord} & := \text{integer} \\
\text{y- coord} & := \text{integer} \\
\text{x- cell size} & := \text{integer} \\
\text{y- cell size} & := \text{integer} \\
\text{delta x} & := \text{integer} \\
\text{delta y} & := \text{integer} \\
\text{number} & := \text{integer} \\
\text{empty} & := \text{integer} := [0-9]+ \\
\text{identifier} & := [a-zA-Z]([a-zA-Z0-9])* 
\end{align*}
\]
1. Introduction.

Simulated Annealing is a stochastic optimization method. This method is based on the existence of a connection between combinatorial optimization and statistical mechanics. The framework for this optimization is provided by a detailed analogy with annealing in solids. Simulated annealing is particularly useful as an optimization method. An algorithm is used for appropriate numerical simulation of the behaviour of a many-body system at a finite temperature. This algorithm provides a natural tool for bringing the techniques of statistical mechanics to bear on optimization.

The model in the statistical mechanics, which is used as an analogue for optimization, consists of a mixture of a number of inhomogeneous liquids. This mixture has to be cooled (annealed) until it becomes a mono crystalline structure, associated with a ground state of matter. With other words a single crystal is growing from a melt by lowering the temperature. To lower the temperature is not a sufficient condition for finding the ground states of matter. The procedure which on the other hand guarantees the sufficient condition consists of careful annealing, first melting the substances, then lowering the temperature slowly and spending a long time at temperatures in the vicinity of the freezing point. If this procedure is not followed exactly, the substance may get out of equilibrium. The resulting crystal will have many defects, or the substance may form a glass with no crystalline order and only metastable, locally optimal structures.

2. The analogue.

In the section above we already talked about the physical model consisting of a mixture of inhomogeneous liquids, which will be used as an example for showing the annealing process in statistical mechanics. On the other hand we have the gate array system with a placement cost function that has to be optimized by an

* This paper has been produced in the context of TASK IX of the project MB-09 supported by the Commission of the EEC, under the 3744/81 Microelectronics program. Partial support by the Ministry of Economic Affairs of the Netherlands under the NELSIS program is hereby also acknowledged.
optimization method (simulated annealing). In the following it will be shown how the physical model can be used as an analogue for the placement of gate arrays.

In the physical model we know the following terms: atoms, movement and collision of atoms, temperature, energy of the system, cooling and heating, single crystal and glass. These terms can be compared with the terms of the gate array placement model.

Figure 1. Comparison of annealing in solids and simulated annealing.

There is a certain disagreement between the placement and the mixture of liquids. The movement and collision is a typical parallel process i.e. at the same time many (almost all) atoms or molecules are moving through the system. The interchanges of modules are generally done sequentially. Only recently parallel organization schemes have been developed for simulated annealing. Mostly, however, the parallel motion of the atoms is simulated by interchanging two random modules. This simulation associates one time step in the physical model with a fixed number of interchanges. The consequence is that the control function which is time dependent has to be adapted to this change of the time variable. All this will have the effect that the time variable in both cases is equal to a certain number of movements or interchanges. The movement of an atom and the collision of two atoms is simulated by an interchange of two, randomly chosen modules of the gate array.

3. Metropolis.

Iterative improvement, commonly applied to optimization problems, is much like the microscopic rearrangement processes modeled by statistical mechanics, with the cost function playing the role of energy. However, only accepting
rearrangements which lower the cost function of the system is like extremely rapid quenching from high temperatures to the temperature zero. Thus it should not be surprising that resulting solutions are usually metastable. The Metropolis procedure derived from statistical mechanics provides a generalization of iterative improvement in which controlled uphill steps can also be incorporated in the search for a better solution.

Metropolis in the earliest days of scientific computing introduced a simple algorithm which can be used to provide an efficient simulation of a collection of atoms in equilibrium at a given temperature. In each step of this algorithm, an atom is given a small random displacement, and the resulting change, $E$ (delta energy), in the energy of the system is computed. If $E \leq 0$, the displacement is accepted, and the configuration with the displaced atom is used as the starting point of the next step. The case $E > 0$ is treated probabilistically; the probability that the configuration is accepted is $P(E) = \exp(-E/kT)$.

Random numbers uniformly distributed in the interval $(0,1)$ are a convenient means of implementing the random part of the algorithm. One such number is selected and compared with $P(E)$. If it is less than $P(E)$, the new configuration is retained, if not, the original configuration is used to start the next step.

4. Initialization.

The annealing algorithm needs to be initialized. The initialization phase assigns an initial value to the temperature and prepares the initial placement. This is read from the description the user has defined.

Before the annealing phase can start one has to be sure that all substances are in a dissolvable condition in the mixture of the analogue annealing model. This means that the placement has to grow worse, i.e. the cells have to be mixed so that the quality of the placement will decrease. The decreasing of the quality of the placement is called the heating phase because the temperature of the system increases during this phase. At one point of the heating phase the increase of the temperature is stopped and the placement created corresponds with the maximal necessary disorder placement. The maximal necessary disorder placement corresponds with the mixture containing all substances in a dissolvable condition. There is only one problem: we don't know when the point the heating phase ends is reached exactly. In practice heating is stopped when more than 99.5 percent of the moves are accepted. The end of the heating process defines the starting temperature.
The determination of the starting temperature in the heating phase is important to guarantee the algorithm ends with an optimal placement. If the placement reached after the heating phase is better than the placement corresponding with the maximal necessary disorder, we have to anneal until this placement is reached. This means a lot of computation time wasted. On the other hand, if we don’t heat enough, the annealing algorithm cannot find the optimal placement.

The starting temperature is determined by the mean absolute value of the delta-energies of a certain number of moves. This corresponds with the temperature being calculated from the fact that the exponent of the Boltzmann equation is set to one. With this method we can come fast and with a little effort to the starting point of the annealing phase.

5. The control function.

The function controlling the temperature is almost the most important part of this optimization approach. When the temperature lowers too fast the system will end in a glass, and when on the other hand the temperature lowers very slow the system will become a single crystal but the computation time to run the optimization process will be immensely huge. One thing we know is that when we cool infinitely slow the probability of reaching the global minimum of the energy function is equal to one!

Our aim is to find a control function in such a way that we know with a certain confidence that the global minimum of the cost function will be reached and the costs of this optimization process in terms of computation time will be minimal. Thus, the problem of finding a suitable control function is transformed in the problem of finding the fastest temperature lowering scheme such that reaching the global optimum is guaranteed.

In the gate array placement model there is no time variable. Thus when we want to use a time dependent temperature scheme we have two undetermined variables. We can change the number of interchanges or moves per temperature step and we can change the temperature step size.

At the time we performed our experiments no theoretical results about the temperature lowering schemes were available. Some results are available now [Otten 85]. We tried to find an appropriate scheme experimentally. For this task we designed a test example that emulates the behaviour of a gate array. The fact that the optimal placement is known is what makes this test example special.
The temperature lowering scheme we found is explained next.

When the number of cells of a gate array is equal to \( n \) then the maximum number of moves at one temperature will be \( 100n \) and the maximum number of accepted mover (interchanges) will be \( 10n \). These two figures are to be considered as stop conditions for one temperature step i.e. of one of these conditions is reached a new temperature is calculated.

![Figure 2. The course of the acceptance level in function of the temperature.](image)

For each temperature the ratio of the number of accepted moves and the total number of moves is calculated. We call this ratio the "acceptance ratio" or "acceptance level". When we compare the acceptance ratio and the energy of the system as a function of temperature we see globally the same shape except that the acceptance ratio runs much smoother than the energy. This means that we can use the acceptance ratio for controlling the temperature of the system.

The controlling of the temperature we choose to be piece wise linear. The temperature lowering step changes at certain points of the acceptance ratio curve, the break points. The determination of the breakpoints depends on the values of the acceptance level. For our scheme we used only two breakpoints, one at the acceptance level of 90% and the other one at 10%. These two breakpoints are chosen this way because the shape of the acceptance ratio curve suggests three different regions. The critical region is the most important one because the energy decreases there very strongly. Therefore we chose the
temperature steps in the three regions to be 0.7 in the upper region, 0.9 in the central region and again 0.7 in the low temperature region.

The last things to fix are the stop conditions of the annealing algorithm. One stop condition is reached if the acceptance level becomes lower than a certain value. The value of the lowest level we chose is equal to 0.5%. Necessary is also another stop condition. Consider the case that a system has a close clustered configuration which may lead to the fact that the previous stop condition is never reached. In that case we must introduce stop condition to guarantee the annealing algorithm will always finish. This stop condition is given by the maximum number of times the maximum number of moves is exceeded; it is called the freeze condition.

With this scheme we found the optimal placement of the test example rather fast. We also used it in some other examples, and it seems to be quite useful in future as an overall scheme.

6. The choice of model.

The placement will be a computational nightmare if we do not simplify the model.

In principle the simplified model must meet the following conditions:

1. an update of the placement must be very easy and cost just a few calculations.

2. The minimum of the cost function of the simplified model has to coincide with the optimal placement of the gate array.

The gate array will be considered as a two dimensional array of cells. All cells are equal in dimensions and completely interchangeable, so they will be considered as points. This is the first simplification. Normally, the cells contain two rows of pins to be connected, one on the top-side and one at the bottom-side. Each pin is connected to one or no net. When the cells become points the pins will be collapse together to one pin (in the center of the point) connected to a certain number of nets. According to this simplification we store the information of pin order and to which pin each net is connected.

Note that it is self-evident that in our model the total "movespace" has to be accessible. For our application this means that every interchange of every two cells must be possible.

The placement of a gate array involves a cost function (also called object function) which has to be minimized to a global minimum corresponding to the
optimal placement of the personalized gate array. When making no simplifications the cost function consists of the total wire length of all nets. It is very easy to see that the calculation of the total wire length after every interchange of two cells will be very tedious. Therefore, we try to make-up a cost function which is very easy to update after an interchange of cells.

We will summarize the terms contained in the object function:

1. routability (uniform distribution of nets)
2. total wire length
3. number vias
4. number feedthroughs

Now we show the structure of the simplified cost function. The cost function is the total sum of the perimeters of the surrounding boxes of all nets. The surrounding box of a net is introduced to obtain an idea of the length of a particular net in a simple way. To get to know the length of a net is a great problem because it is not routed yet. When we introduce the surrounding box of a net we know that the net must be enclosed independent of the routing of the net. If the surrounding box of a net is cut by a cross-line we know sure that the net is cut at least once by that line. Thus, the surrounding box is a measure for the total wire length. Certainly it is a lower bound for it.

The gate array is constructed of a number of columns and rows, the grid of the gate array. The smallest grid unit is equal to the dimensions of the smallest possible basic cell. All dimensions and measures in this report will be expressed in terms of these grid units. For example, a 10x12 gate array is a matrix of 12 rows containing 10 cells each.

To estimate the total wire length, the total chip is cut at each grid coordinate in both the x- and y-direction. At every cut the number of the surrounding-boxes that is hit or cut by the cut-line is determined. If we add these numbers for all cut-lines we have an estimate of the total wire length. This total sum is equal to half of the sum of the perimeters of all surrounding boxes. It is also possible to express a preference of a certain kind for the form of the surrounding boxes in the cost function. This is useful if the routing of a particular net is more difficult in one direction than in the other direction. For example it is more favourable to route a net in the "channels" than across the rows of cells, which is only allowed at feedthrough spacings. We display the cut-counts in histograms for each direction.
Now the cost function can be easily constructed using these two histograms. The area of such a histogram gives an estimate of the total wire length in one direction. Wirelength is however not a measure sufficient to evaluate the quality of a gate array design. There is also a need for providing an expression accounting for the homogeneity of the distribution of the nets. This expression will be given by the maximum column height of the two histograms. The next example will show this more in detail. Consider a personalized gate array with the following histogram:

![Figure 3. Badly placed chip.](image)

The total area of that histogram is rather small, although the associated placement is very bad, because of the high density of the nets at one place of the chip. This is for gate arrays very inconvenient since the routability can be lost with the consequence that the placement becomes worthless. Therefore, the cost function we will use to optimize the placement by simulated annealing is defined by:

\[
F = w_h^s \sum_{\text{col}} H_h^c + w_h^m \max(H_h^c) + w_v^s \sum_{\text{row}} H_v^t - 2' + w_v^m \max(H_v^t)
\]

where

\- \(w_h^s\): Horizontal sum weight factor
\- \(w_h^m\): Horizontal max weight factor.
\- \(w_v^s\): Vertical sum weight factor.
\- \(w_v^m\): Vertical max weight factor.

The weight factors are meant to control the impact of the terms of the cost function on the performance of the placement. Their mutual ratio of the weight factors is important. The ratios \(w_h^m/w_h^s\) and \(w_v^m/w_v^s\) give a measure of the preference for the vertical or horizontal direction respectively.

Note that the magnitudes of the 4 terms appearing in the cost function are totally different. This difference can be explained from the fact that the "sum" terms are formed by the sum of all columns of the histograms compared with
the "max" terms that are constructed by the size of only one column. One has to correct this by adapting the weight factors in a way that the magnitudes of the terms become about the same size.

7. Implementation aspects

Conform the number of annealing atomic actions necessary for one run, the total run time will be the bottle-neck in the implementation of this optimization algorithm. It will be clear that the run time optimization of the atomic action has got the greatest attention during the implementation phase of the annealing algorithm.

PASCAL was used for the implementation, because on the moment the annealing research on gate array placement was started the language C was not available. (C was the language chosen for the NELSIS project).

8. Data structure

Good runtimes are achieved by carefully designed data structures. In our case the most obvious data structure design would be by linked lists: one per net, containing the connected modules (cells) and one per cell, containing the connected nets. The advantage of this approach is the dynamic memory for the storage of all relevant information of the nets and modules, but its big disadvantage is the limited accessibility which results in a time consuming process when records are not directly accessible.

Insofar as we consider storage time tradeoffs for the annealing we rather trade time for memory than the other way around.

Fortunately memory is not a problem in the implementation of annealing, even though the number of cells, nets and modules grow up to several of thousands if VLSI circuits will be used. Consider the example of a 5000 gates array with 3000 nets and maximally 20 cells connected to one net, and each cell has 8 terminals. Then the data structure needs 5000*8+3000*20 + 100,000 records. So the whole data structure will claim about one million bytes supposing that each record is 10 bytes long. The memory required by using such structure is still quite acceptable.

All these considerations will point to the rather obvious use of arrays (PASCAL) for the storage of the data of the data structure. An array structure has the important characteristic that all elements are directly and separately accessible.
There exists also the problem of inserting and deleting one element which must not be time consuming. The solution to this problem, for the annealing algorithm, uses the fact that the number of elements in the "lists" is constant. This means that if an element has to be deleted another or the same element will be inserted.

The net records contain all cells connected by the net and the cell records contain all nets connecting to the terminals of that cell. That is because one has to know the nets which are connected to a moving cell. Surrounding rectangles must be determined for each of the connecting nets individually. For the determination of that surrounding rectangle the position of all terminals (cells) has to be known. When all terminal positions are known the surrounding rectangle can be determined by finding the minimum and maximum coordinates of the terminals. This is a time consuming process. To improve the performance of the implementation, one has to find a solution for a very fast calculation of a surrounding rectangle of a net.

The solution was found by using sorted arrays for the x-coordinates and for the y-coordinates of the terminals of each net. Now the minimum and the maximum coordinates are resp. the first and last element of the sorted arrays. Note that for this operation we claim that all these arrays must be sorted. Sorting arrays is a difficult action to do, but in our case it can be very easy. Because of the number of elements being constant and of the fact that only one element is changing position at the same time, sorting is accomplished by shifting some elements to the right or left.

The introduction of arrays for the implementation of the netlist has caused a speeding-up of a factor 2 or 3. The algorithm is now about 2 or 3 times faster than it would be if it had used linked lists. The speeding-up factor is of course dependent of the size of the array and the complexity of the netlists. For large arrays the speeding-up will be even larger than 2 or 3.

The next figure displays the data structures in PASCAL notation.
Type

gate_range = 0 .. max_num_of_gates;
column_range = 0 .. max_num_of_columns;
row_range = 0 .. max_num_of_rows;
cell_range = 0 .. max_num_of_cells;
net_range = 0 .. max_num_of_nets;
cell_per_net_range = 0 .. max_cell_per_net;

netlist_point = 'netlist_record;
netlist_record =
  RECORD
    netnum : net_range;
    next : netlist_point;
  END;

{ "MOOREC " }
cell_record =
  RECORD
    net_list : netlist_point; { pointer to head of netlist }
    colnr : column_range; { column index of cell }
    rownr : row_range; { row index of cell }
    leftmost : column_range; { leftmost column index of macro }
    rightmost : column_range; { rightmost column index of macro }
  END;
cell_array = ARRAY[ cell_range ] OF cell_record;
sort_array = ARRAY[ cell_per_net_range ] OF cell_number;

('METRIC'

net_record =
  RECORD
    xmod : sort_array; { cell list on x-coord. order }
    ymod : sort_array; { cell list on y coord. order }
    lastmod : cell_per_net_range; { number of cells in cell list }
    x_weight : integer; { weight factor in x direction }
    y_weight : integer; { weight factor in y direction }
  END;

net_array = ARRAY[ net_range ] OF net_record;

Figure 4. Data structure in PASCAL.

9. Program.

A distinction has to be made between a move and an interchange of cells. The change of energy -- delta energy -- is necessary for the determination of the acceptance of such an interchange. When an interchange is not accepted, the available energy and its corresponding interchange must be undone. This results in a placement which is exactly equal to the previous placement. To avoid the undoing of interchanges the moves were introduced. A move exchanges two cells virtually so as to be able to calculate the change of energy implied by the interchange. Now when a move is not accepted nothing has to be done to get the old system configuration back. This leads to a reduction of the cpu-time with a factor of almost 2. Therefore introduction of moves is very useful the more because the total number of not accepted moves is at least two times larger than the number of accepted moves (interchanges). These are results from experiments. Therefore the benefit of decreasing the computation time of a move is rather large. This benefit is also very profitable since the time necessary for accepted moves is totally wasted because it contributes nothing to the improvement of the system configuration.
The data structure design of the simulated annealing program is the most important part of the implementation concerning the speed optimization. Therefore, a lot of effort has been supplied for the database design, which has lead to an efficient implementation of the annealing algorithm. The implementation consists of a rather simple program controlling the "moves", "energy" and "temperature". During the designing phase a great emphasis was put on the "Stepwise refinement" method elaborating the rudely specified statements.

PROCEDURE SIMULATED ANNEALING;

begin:
  initialize surrounding rectangles;
  calculate initial energy;
  determine initial temperature;

  heating
  while not accept do
    begin:
      melt:
        metropolis( stop_melt_acc, stop_melt_att, accept );
    end;
  end;

  cooling
  repeat
    anneal:
      metropolis( stop_ann_acc, stop_ann_att, accept );
    if not accept then temp_not_acc := temp_not_acc + 1;
    until ( temp_not_acc > limit_count ) or
      ( accept_level < limit_level );

  print placement;
end; ( simulated annealing )

Figure 5. Pseudo code description of the annealing program.

The routines MELT and ANNEAL increase and decrease the temperature by a step according to the temperature schedule. They return a new temperature, which will be used for the Metropolis algorithm to do some interchanges. The Metropolis algorithm determines if the delta-energy of a move belongs to an accepted move or to a rejected one. This Metropolis procedure actually implements the Metropolis algorithm, which emulates the behaviour of the Boltzmann equation. The pseudo code description is shown next.
PROCEDURE METROPOLIS( stop_acc, stop_att, accept );

( count_acc := counts # accepted moves )
( count_att := counts total # moves )
( stop_acc := upper bound for # accepted moves )
( stop_att := upper bound for total # moves )
( accept := indication for successful termination of annealing at current temperature )

BEGIN ( metropolis )
REPEAT
pick two cells intended for interchanging at random;
calculate delta-energy;

IF delta_energy can be accepted THEN
BEGIN
energy := energy + delta energy;
effectuate interchange of the cells;
count_acc := count_acc + 1;
END
ELSE
IF RANDOM < EXP{ - delta_energy / temperature } THEN
BEGIN
energy := energy + delta_energy;
effectuate interchange of the cells;
count_acc := count_acc + 1;
END;

count_att := count_att + 1;
UNTIL { count_acc > stop_acc } OR { count_att > stop_att };

IF { count_acc > stop_acc } THEN
accept := TRUE;

END; ( metropolis )

Figure 6. Pseudo code of the Metropolis algorithm.

10. Experiments.

To determine the best temperature lowering scheme, one has to know what is the best possible placement, and in what time can it be generated. Therefore, the "final energy" and the computation time for putting the system in the state with that energy are in the first place important for evaluating the scheme. Of course this collected experience is of general importance and not particular for this application of the annealing algorithm. To increase the insight in what is the best lowering scheme for annealing with gate arrays, some additional information is necessary. In that case not only the final energy must be satisfying but also some grades for the routability are required. The routability factor is very difficult to determine without a routing program. In the examined experiments the routability factor was determined by observing a figure which expresses the net density.

The experiments were done on four different gate arrays:

1. The 'RANDOM' gate array.
2. The 'CHESS-BOARD' gate array.

3. The 'BARS' gate array.

4. The 'TEXAS' gate array.

10.1 'Random' gate array.

The 'RANDOM' array consists of 12 rows and 18 columns, and contains 180 occupied cells located at 216 possible positions. This corresponds with an array filling of about 84%. The name of the gate array results from the fact that the personalization of the array is based on a random netlist. The netlist, which contains 100 nets, is generated in a random manner but not totally arbitrarily. On the construction of the netlist there were some constraints. They were:

1. A net may contain only 2 to 10 cells to which it is connected.

2. A cell may only be connected to 4-12 nets.

Of course these constraints were defined empirically for creating a netlist matching practical cases.

In the first place this gate array was designed for testing and verifying the implemented annealing algorithm, and secondly it has also been used to adjust or determine a suitable temperature lowering scheme.

---

Figure 7. The start (a) and end (b) configuration of the 'RANDOM' ARRAY.
DUE TO THE RANDOM CHARACTER OF THE NETLIST, THE ANNEALING DOES NOT CONVERGE. THE ACCEPTANCE RATIO REACHES THE LOWER BOUND WHICH IS ABOUT 3%.

A study of the netlist provides an explanation of this phenomenon. When we look at the netlist we see one big "sphere" of elements with strong cohesion forces (during the experiments we have called this phenomenon the "chewing-gum" effect), i.e. every cell in the "sphere" has an equal connectivity (force) to all other cells in that "sphere".

Due to this strong connectivity effect it will be impossible to place the array in a way that the connectivity is uniquely distributed and at the same time exhibits a minimum interconnection length. In the center of the array there always will be a congestion of interconnect. This phenomenon appears in the histograms of the final placement.

10.2 'CHESS-BOARD' GATE ARRAY.

These series of experiments were of highest importance since they were used as reference for the determination and adjustment of the temperature cooling scheme. They involve the placement of the so called 'CHESS-BOARD' gate array, which is an array of 8 by 8 cells with a netlist such that the best (minimal) placement is known beforehand. Each net of the netlist connects to four neighbour cells. Hence, the 'CHESS-BOARD' consists of 64 cells and 49 nets.

![Figure 8. Net connection.](image)

The annealing algorithm belongs to the class of combinatorial optimization that is independent of the start configuration, since in the initial phase of the annealing almost an unlimited number of deteriorations are allowed by an increasing temperature until the point of maximal necessary disorder is reached. Therefore, the optimal placement can be used as initial placement.

The best placement has the following configuration.
First of all, the system is observed with the temperature decreasing step equal to 0.95 according to the literature [Kirkpatrick 83]. Especially the acceptance ratio as a function of the temperature is observed. The acceptance ratio shows in its shape three distinguished areas as already mentioned in a previous chapter. It is obvious that these areas characterize the entropy function of the system, and when the control function has a piecewise linear character the parameters of that function will be terminated by these areas. The observation of some experimental results shows that when the acceptance ratio changes much during one temperature step, the temperature decreasing step must be small, for example defined by 0.9 or 0.95. Does on the other hand the acceptance ratio change not very much then the decreasing temperature step can be larger for example defined by 0.7. This piecewise linear temperature lowering scheme will give the result that the final configuration of the optimization process comes rather close to the optimum configuration within a reasonable computation time.
Furthermore, some other faster schemes were tried out, but they would not reach the optimum. The system will be frozen in a local optimum of the objective function. Such a local optimum is characterized by a temperature too low to get out of the equilibrium. The probability to accept deteriorations of the system to get out of the equilibrium is too low.

Sofar a "reasonable" temperature lowering scheme is experimentally determined. The final result of the annealing algorithm depends not only on the shape of the temperature lowering scheme but also on the choice of the weightfactors in the objective function. Weightfactors are, in the first place, intended for controlling the effect of a particular term of the objective function on the final result. The second role of the weightfactors, which is of course of minor importance, is their influence on the smoothness of the objective function. If the weightfactor of the "max" term is an order of magnitude larger than the one of the "sum" term the objective function will have a "smoother" characteristic. The shape of the objective function is important for finding an optimal choice of the degrees of freedom of the annealing algorithm, because there exists a very tight relationship between the objective function and the temperature lowering scheme. This was also observed by Otten [Otten 84].
In case of our 'CHESS-BOARD' placement, values for 4 weightfactors have to be chosen. The weightfactors are chosen in a way such that the effects of each term is of equal order in size. This is done by eliminating any preference of one term in particular. Hence, the values are:

\[
\begin{align*}
    w_{LM} &= 8 \\
    w_{LS} &= 1 \\
    w_{SM} &= 8 \\
    w_{SS} &= 1. 
\end{align*}
\]

10.3 'BARS' gate array.

The gate array example circuit -- in this report called the 'BARS' example -- that was intended for testing the placement algorithm was not really a gate array delivered by a factory as a benchmark. The description of the gate array was extracted from a picture. The picture is shown next.
Figure 11. The 'BARS' gate array.

The shown gate array was used as benchmark for the routing system called BARS (Bath Routing System) by a research group at Bath university. The netlist description was extracted from the shown picture. The nets connected to I/O.
pads were ignored. Note that in almost all gate array images the basic cells have two rows of pins (terminals); one on the top side and one at the bottom side. Each pin in one pin row has its logic equivalent pin on the other pin row. If such logic equivalent pin is used to get one net from one channel to another channel then this connection through the cell is called a feedthrough.

Mostly the definition of feedthroughs is slightly different from the definition given here. The piece of interconnection that goes through a cell row, without having any connection to the underlaying cell is called feedthrough.

Because the feedthroughs were not indicated in the picture it could be possible that a net was split in two or more nets caused by ignoring the possible existence of feedthroughs. This incompatibility of the netlist with the original netlist might have an effect on the placement result.

After the netlist was extracted and typed in, the annealing job was started. The initial placement was also taken from the picture. In this example the starting energy is of great importance, because this is the energy of the placement they used in BARS to route.

The size of the array is 10x12 cells, whereby 110 cells were occupied by a certain set of functions. So it can be considered as a relative small example, the more when compared to the goal of our placement program: the feasibility of placing a gate array consisting of at least 1000 cells.

The filling of the array is very dense (about 92%) and the number of wires (nets) compared to the number of cells is very high (175 to 120). Thus we have a difficult example for the placement and routing programs. Nevertheless, the annealing program placed these 120 cells in about 45 minutes CPU-time (HP9000) with a total energy (cost function) about 150% smaller than the energy corresponding to the original placement used in the BARS system. The starting energy was about 3000 units and the energy at the freezing point was about 2000.
Here again the same course of the acceptance ratio is observed, which points to the usefulness of our temperature schedule.

10.4 'TEXAS' gate array

The experiments involving the 'TEXAS' chip, were examined in order to evaluate the performance of the placement program on larger arrays. So far only small to medium sized arrays for testing the annealing schedule were used. The experiments with the large array can lead to the justification of some parameters of the annealing program, and it is also used for estimating the time complexity. The second point which makes these experiments so important is expressed in the practical value of the examined placements. For example the 'RANDOM' chip reflects a bad imitation of a personalized gate array; therefore its practical value is reduced.

The 'TEXAS' array is based on an image designed by TEXAS INSTRUMENTS. The set of functions to be implemented was derived from a universal I/O controller. The circuit was designed by Stevens during a VLSI design course. Stevens has tried to implement this circuit by a STL gate array from TEXAS INSTRUMENTS. Although the chip was not realized, it is the most realistic example of a design.

The array was first placed without any preference of direction. To be able to
evaluate the result a tool was developed that compiled all net surrounding boxes and sorted them on size. The surrounding boxes turned out to be quite big, and certainly the boxes are not oblong enough. This effect is caused by the fact that the cells are about 5 times larger in one direction than the other one, but in our simplified model they are assumed to be square (or points). So we decided to enlarge the weights of the terms in the objective function for one direction. After the weights were adapted the annealing job was run, with much better results.

Figure 13. The annealing result of the 'TEXAS' array.

The effect of unequal weight for both directions results in a second observation. In the network there were some macro's mostly flip-flops. These flip-flops contain 8 cells. Since the wirability will be better in one direction, it will be clear that the existing flip-flops will be oriented towards that direction to give the best results, because the intra macro wiring is more critical than the inter macro wiring. Here again the results were better when one direction is weighted stronger in the objective function.

11. Results.

At the moment the computed placements had to be reviewed there was not gate array router available within the research group. This had the consequence that the routability of the generated placements could only be estimated.

One possibility of evaluating a placement is to compare the results from the annealing algorithm with the results from one or some other placement algorithms. But these placement algorithms were not available in the research
group at that moment.

Furthermore, some graphical tools were developed. Those tools served to observe some parameters continuously like the net density and the mean length of all nets during the run of the placement optimization program. Also some tools for displaying the placement with its cells and nets to get an idea of the correlation between the energy (value of the cost function) and the routability have been established. The nets were displayed from terminal to terminal just as a straight line.

Figure 14. Minimum spanning tree net connection.

With these tools only a very general informal discussion of the quality of the placement is possible.

In order to give an impression of the CPU-time necessary to run an annealing job, in our case a placement generator, we will not summarize the total time but the time needed for the calculation of an accepted and rejected move of the "chess-board" array on a HP9000-UNIX system in PASCAL.

- accepted move: 189 per sec.
- rejected move: 352 per sec.

This quantity in time depends of course on the implementing machine but it depends also on the size of the array, and on the number of interconnections. Assume that the array contains $N$ cells, then the array size produces a factor $N$. 

7.50
in time. Mostly the number of interconnections is in some way related to the number of cells. Therefore, the real dependency will be of the second order \( O(N^{**2}) \).

The total number of moves necessary for one run (split in accepted and rejected ones) is another concept which contributes to the determination of the total run time. The contribution will be of order \( N \), because the number of moves is set to \( 10^4N \) and \( 100^4N \) depending on the phase of annealing. Together with the previous dependency the total will be of the third order (worst case).

The routability will be probably, reasonably good. This conclusion is a result from the fact that in all examples, the uniform contribution of the nets is achieved (except for the "random" gate array). This is confirmed by the histograms of the final placements and also by the next figure which shows the number of net surrounding boxes at grid positions.

![Figure 15. Non-uniform (a) and an annealed (b) distribution of the nets.](image)

The placement program, applied here, is technology independent per definition, due to the definition of the model. The program can handle restricted area constraints, determined by the move calculation routine. Therefore, it is no problem to place pad cells, by considering them as normal cells present in a restricted area.

In color plate "Gate Array Placement" you can see a result of the placement program, each functional cell is represented by a colored box.
12. Conclusions.

In this report we presented an application of the simulated annealing algorithm as formulated by [Kirkpatrick 83] to the placement problem of gate arrays. This technique turns out to be a very powerful tool for optimization of a cost function involved in the gate array placement problems.

The advantages can be formulated as:

1. The algorithm is applicable to all types of gate arrays.
2. The algorithm is very easy to implement.
3. The result can be as "good" as desired.

The disadvantages are:

1. Automation of the overall cooling scheme is not realizable yet.
2. The required computation effort (time) may be very large (too large for an interactive process).

The result of the annealing algorithm is highly dependent on the choice of the cost function. If the global shape of the cost function is too smooth then the annealing will not work well. The reason is that the metropolis algorithm does not work because the cost function will change only with a small set of interchanges. When on the contrary the cost function has a very capricious character with not one clearly obvious global minimum then the annealing program may spend a lot of computation time in the region around the minima without making a clear progress. The temperature lowering schedule we used seems to be applicable and is still very simple. Our model is appropriate for the description of all kinds of gate array images and therefore we may assume that our placement program is fairly technology dependent.

13. References.


Hierarchical wire routing of gate arrays.
P.A.C.M. Nuijten.
Eindhoven University of Technology.

1. Introduction.

Gate array layout design consist of the following three phases: functional decomposition, placement and routing. Normally routing is performed in two stages: global routing and local routing. The reports describes the problems which arise during the global routing phase.

The global routing algorithm used is hierarchical of nature to cope with the still increasing gate array sizes and determines routes for connection single points of fixed patterns (to cope with the different shapes).

During the global routing stage the wires are routed in terms of cells. Major goals during this phase are avoid violating channel capacity restrictions (embedding more wires than permitted) and produce near minimal total wire length. The algorithm used is mainly based on the concepts developed by Burstein with further extensions for preplaced fixed wiring patterns.

2. HIERARCHICAL WIRE ROUTING

2.1 Introduction.

The purpose of global routing is to provide a loose initial routing of nets on a chip to guide the subsequent local routing stage. It is a crucial part in gate array layout design. A gate array wafer is preprocessed up to the interconnection level and custom logic functions will be realised by the final interconnections. As mentioned before the gate array layout involves the phases placement and routing. However it is quite difficult to determine whether or not the placement results are satisfactory until the routing is done. On the other hand it is also hard to say which phase needs improvement if routing fails: when a poor wiring method is used, we may not complete the interconnections for otherwise good placements.

The wiring space in gate arrays is limited. Channel widths are fixed, giving rise to the routing problem for gate arrays, that of fitting the desired

* This paper has been produced in the context of TASK IX of the project MR-09 supported by the Commission of the EEC, under the 3744/81 Microelectronics program. Partial support by the Ministry of Economic Affairs of the Netherlands under the NELSIS program is hereby also acknowledged.
connections into the available channels. In order to obtain a 100-percent wire completion, we need to distribute the wires over the chip such that no overflows (unability to make the desired interconnection of a net) occurs. This problem is not trivial if

- each wiring channel is almost saturated.

- the channel capacities and pin distribution are not uniform due to physical constraints (e.g., complicated macro design, internal blockade).

As the scale of integration grows, the interconnection problem becomes increasingly difficult if not intractable. Modern automatic wire routing methods can be classified as:

1. maze-runners or Lee-Moore type routers.
2. line probe routers.
3. channel routers.
4. pattern routers.

Channel routers require a restrictive arrangement of components and do not explore all areas technologically available for wiring. Routers based on "line propagation" are either extremely time consuming or may fail during the search of existing interconnection. This is why maze-runners are widely utilised for wiring very dense chips.

All these routers work in a one-net-at-a-time mode, i.e., all nets are wired in some sequence. Moreover, "maze-runners" and "line-propagation" routers are also dependent on internal ordering of the terminals within a net. Picking different terminals as a source for wave or line propagation may result in different interconnections.

Another disadvantage of "maze-runners" is speed. The time for interconnection of one net may be in the worst case proportional to the number of grid points of the maze. In order to speed up the maze running the wiring process is often realised in two phases: global wiring and final track allocation. At the global wiring phase each net is wired in terms of cells of the gate array. Instead of allocating the wires at final track resolution, a global route of the net through the cells of the gate array is allocated. This limits the area for the wave propagation during the final wiring phase to the cells specified by the global route. The idea of global wiring is the crucial assumption in the hierarchical approach.
Pattern routers are fairly new and their capabilities are not yet completely clear, but they seem to have the potential of overcoming the above mentioned difficulties of other routing methodologies. This approach should be viewed just as another collection of heuristic tricks for routing. Most wire routing problems (except trivial cases) are NP-complete so not much more can be expected. Uniformity of the wiring substrate is however the crucial assumption of the method. Optional is the assumption that horizontal and vertical wire segments are realised on different wiring layers and vias are introduced each time a wire changes direction. Any jogs ("wrong way" wires on one layer) are prohibited.

Within these limitations this approach is advantageous over the existing wiring methodologies.

- The final layout of wires is independent of net ordering and ordering of the pins within the nets.
- Because the hierarchical nature of this method it is inherently fast, usually by an order of magnitude faster than the routers based on wave propagation (maze-running) technique.

### 2.2 Notation and problem description.

**Definitions:**

We will be dealing with standard gate array technology: i.e. a uniform grid of cells \( G = G(i,j) \), \( i = 1,2,\ldots,p \), \( j = 1,2,\ldots,q \) called the final grid. A cell is the intersection of the horizontal and vertical channel together with the corresponding gate components or circuits.

\[
\begin{array}{ccc}
\text{wiring bay} & \text{wiring bay} & \text{wiring bay} \\
\text{XXXXX} & \text{XXXXX} & \text{XXXXX} \\
\text{XXX circuit} & \text{XXX circuit} & \text{XXX circuit} \\
\text{XXXXX} & \text{XXXXX} & \text{XXXXX} \\
\text{wiring bay} & \text{wiring bay} & \text{wiring bay}
\end{array}
\]

Figure 1. Typical Gate Array Model.

We assume two wiring layers to be available, usually both metallisation. Each cell is surrounded by North, East, South and West boundaries.

The **channel capacity** of each boundary is the estimated number of wiring tracks that can be driven through this boundary on both layers.
In order to guarantee the completion of the final routing we assign to each cell also a via capacity which is the estimated number of vias the cell can accommodate. Usually one layer is exclusively used for horizontal wire segments, another for vertical wire segments and vias are introduced for each layer change when the wire bends. Because of various technology constraints vias cannot be placed in certain configurations. Some of the most prevalent restrictions are shown in 3.

![Diagram of via placement constraints](image)

**Figure 2. Some examples of via placement constraints.**

For a given via position (0) the forbidden neighbour positions (X) for other vias are shown on the track intersections.

In general, for a given cell of \( n \) horizontal and \( m \) vertical tracks (referred to as an \( n \times m \) cell) there is a maximum number of vias \( V(n,m) \) that can be allowed for each via placement restriction. Beyond this maximum number \( V(n,m) \) no clever arrangement of vias can satisfy the restrictions.

On the other hand there is a minimum number, \( v(n,m) \), of vias that can be placed in a cell such that if any number less than \( v(n,m) \) is embedded in the cell, one can always find another legal via position, regardless of the configuration of the vias. Therefore, if the vias allocated in a cell do not exceed \( v(n,m) \), it is almost certain that exact embedding will not fail in the cell due to the via placement restrictions.

**Problem description:**

The net obtained as a result of the functional decomposition consists of a set of final grid cells of \( G \) where the pins and possible fixed wire interconnections (usually the metal of a macro instance) are located. The pins of the net are located somewhere within these cells.

The problem of global wiring is to assign a wire route for each net of the interconnection network, i.e. for each net to indicate exactly the cells and
boundaries the net passes through. There are two basic conditions that are to be satisfied during the global wiring:

1. **Channel capacity conditions.** The number of nets crossing any boundary must not exceed the channel capacity associated with this boundary.

2. **Via conditions.** The number of bending wires within a cell must not exceed the estimated number of vias than the cell can accommodate.

### 2.3 Hierarchy.

A **hierarchic system** is a system composed of interrelated subsystems, each of the latter being hierarchic in structure, until some lowest level of elementary subsystems is reached. There is a conjecture that complex systems are fare more manageable if they are of hierarchic nature than non-hierarchic systems of comparable size, and that aspects of complex systems that are not hierarchic even elude human understanding and observation. Both in nature and science many instances support this conjecture.

In our hierarchic system the elementary subsystems are final grid cells and the subsystems are **supercells** which consist of a configuration of several final grid cells with a rectangular shape. We use a **bisections hierarchy**. Each hierarchical level consists of a grid of supercells which can be mapped onto the final grid. **Bisection** is an operation which divides a rectangle into two smaller rectangles. In the grid we take a strip of supercells of height one (a row or a column) called a $1 \times N$ grid, and partition this by a cut line parallel with the longest side into a $2 \times N$ grid. The cut line is always applied in horizontal or vertical direction and the strip is always a row or a column in the grid. Each hierarchical level consists of a grid of supercells. At the next level each of these supercells generates two descendent supercells. This procedure is repeated until final grid resolution where the supercells become final grid cells.

Following the same philosophy that led to the separation of the global wiring stage from the final assignment of the wiring tracks we introduce a "super" global routing in terms of pairs of cells, then in terms of quadruples and so on, until the problem vanishes. The hierarchical wiring method is in a sense a reversed process, where we start with a trivial problem of wiring the grid of two supercells and subsequently build the wire routes for every level of the chip hierarchy. At the very first level, after the initial partition, the global wiring consists of simple net classification: we determine which nets cross the imaginary cut line (i. e. which nets have terminal cells in the
different parts) and which do not. The nets that do not cross the cut line will utilise only the chip area of one of the super cells and are not routed at this stage. At the second level, after we partitioned the chip one more time, the net classification becomes more complex; for each net we define a set of supercells (at this level there are only four supercells) that contain interconnection terminals of each net. The basic idea of the hierarchical global wiring is to perform global wiring at each level of the bisections hierarchy.

Hierarchical level I.

Figure 3. Typical wire route at certain level of hierarchy.

Let us describe the recursive step of hierarchical global wiring when we are moving from I-th level of hierarchy to the level I+1. Suppose that a net at the I-th level occupies a set of supercells denoted by $G(i,j); i = 1,2,\ldots, n, j = 1,2,\ldots, m$.

Initial route at hierarchical level I + 1.

Figure 4. Routing area at the next level of hierarchy.

Suppose also that each of these supercells is partitioned in a horizontal direction, so that each supercell $G(i,j)$ generates two descendent supercells $G-(i,j)$ and $G+(i,j)$ where $G-(i,j)$ is placed underneath cell $G+(i,j)$, resulting in a $n \times 2m$ matrix of cells. For each net of the network the following procedure is applied: Consider the supercells at the I-th level of hierarchy which are occupied by the net and consider their descendents which define the wiring area for this net at the $(I + 1)$-th level of hierarchy. Each rectangle of fig 3 can be subdivided into rectangles of height at most 2. Obviously the route has to
be located within the shaded area of fig 4. We solve the problem by routing the net in all these 2*N subgrids. So the global wiring problem is reduced to determine the route of the nets in each of the partitioned 2*N subgrids.

Route chosen at hierarchical level I + 1.

Figure 5. Wire route at the next level of hierarchy.

A typical resulting picture is fig 5. We proceed in this manner up to single cell resolution; and at the final level we come up with the complete global wiring. The main advantage of this design approach is that the wiring part of layout design is reduced to a relatively simple case: wiring a net within a 2*N grid. Wiring on the I-th level of hierarchy is dependent of the wiring on the (I - 1)-st level. As mentioned the wiring area for a net can be subdivided into rectangles of height at most 2, and the problem is reduced to independent wiring inside of these 2*N rectangles, referred to as 2*N grids.

2.4 Construction of 2*N wiring problems.

This paragraph describes the manner by which the wiring information of a net at the previous level of hierarchy is used to construct the 2*N wiring problems at the current level of hierarchy. We consider now only pin to pin connections, (fixed wire interconnections of macro instances are out of consideration at this moment). Suppose that at the previous level of hierarchy a row of supercells represented by the 1*N grid G is partitioned in horizontal direction resulting in a 2*N grid. Each 1*N supercell G[i] generates two descendant supercells G-[i] and G+[i], supercell G+ is situated above G-. We must now construct the sets A_i of terminal cells {G+[i],G-[i]} which have to be interconnected.

The 2*N wiring algorithm which will be presented later, has the following properties:

- only wire interconnections within the 2*N grid area are made.

- all terminal cells of set A are interconnected with interconnect within the 2*N grid.
Let \( \text{left} \) denote the abscissa of the leftmost terminal cell of set \( A \) and \( \text{right} \) the abscissa of the rightmost cell of set \( A \). Than the resulting wiring route will have horizontal interconnections with horizontal interconnections at all intermediate columns between \( \text{left} \) and \( \text{right} \). From the previous hierarchical level the parts having horizontal wire connections are known so we split all cells \( G[i] \) in sets with each set consisting of cells interconnecting each other within the \( l*N \) grid area.

All horizontal interconnections within the \( l*N \) grid are removed. The set of terminal cells are defined by the locations of the interconnection terminals and the existing wire interconnections to the outside area of the \( 2*N \) grid.

- If \( G[i] \) contains an interconnection terminal, meaning that one or more pins of the net are located somewhere inside this supercell, then after partitioning \( G+[i] \) or \( G-[i] \) or both will contain interconnection terminals defined by the net list.

![Figure 6. Partitioning a terminal cell.](image)

For each interconnection terminal a terminal cell is introduced because these cells have to be interconnected.

- Interconnections to the outside of the \( l*N \) grid should remain existing because only inside the \( 2*N \) grid new interconnections are made by the router. For each outgoing interconnection a terminal cell is introduced adjacent to the boundary crossed by this interconnection.

![Figure 7. Outgoing interconnections.](image)
After determination of our terminal cells assignment to the previously defined sets based on the location in the 2*N grid is performed. Now each set with more than one terminal cell defines a separate 2*N wiring problem.

Example:

- No separation in different sets would result in a superfluous horizontal interconnection between the columns 3 and 4 in the example.
- A set with one terminal cell contains always a interconnection terminal with a possible outgoing wire interconnection to the outside area of the 1*N grid.

2.5 General 2*N wiring problem.

The problem of global wiring within a 2*N grid can be described as follows:

Figure 8. Example of definition of the sets A.

Figure 9. A general 2*N grid.
The 2*N grid is a uniform array of cells. Let e(1) ... e(m) denote the set of nets which have terminals or existing wire connections in the 2*N grid. We now remove all wire connections of the previous level which lay in the current 2*N grid. Then each net e(i) is identified with a set of cells of this array, called terminal cells. If cell G(i,j) is a terminal cell of net e(k) than at least one of the terminals (pins) or a wire connection to the outside area of the 2*N grid is located somewhere within cell G(i,j).

We have also a 2*(N - 1) matrix of the horizontal channel capacities

\[
\begin{align*}
  h(1,1), & \quad h(1,2), \ldots, \quad h(1,N - 1) \\
  h(2,1), & \quad h(2,2), \ldots, \quad h(2,N - 1)
\end{align*}
\]

and an N vector of vertical channel capacities

\[
\begin{align*}
  v(1), & \quad v(2), \ldots, \quad v(N)
\end{align*}
\]

associated with our 2*N grid. Where \( h(i,j) \) denotes the estimated channel capacity between cells \( G(i,j) \) and \( G(i,j+1) \) and \( v(i) \) denotes the estimated channel capacity between cells \( G(1,i) \) and cell \( G(2,i) \).

A graph of our 2*N grid consists of 2N vertices associated with the grid cells; any two vertices associated with neighbour cells are adjacent. The lattice graph is depicted in fig 10.

A route of the net is a subtree of this graph containing all vertices associated with the terminal cells. Our problem is to determine routes for all nets which have more than two terminal cells in the current 2*N grid.

The final solution of wire routes for all nets has to satisfy both the boundary and via restrictions:

- The number of wires crossing the boundary between the cells \( G(i,j) \) and \( G(i,j+1) \) must not exceed \( h(i,j) \).
- The number of wires crossing the boundary between the cells \( G(1,i) \) and
G(2,1) must not exceed v(i).

- For each cell G(i,j), the number of wires bending (changing direction within this cell) for which this cell is not a terminal cell must not exceed the via capacity of that cell which is a function of h(i,j) and v(j).

In the following chapters we describe two different approaches to solve the wiring problem in a 2*N grid:

- The first solution is based on the exact solution of this problem in case of a 2*2 grid using an integer linear programming approach and the hierarchical extension in a "divide and conquer" fashion.

- The second solution is based on more traditional one-net-at-a-time routing within the 2*N grid embedding all nets sequentially. It turns out that the Steiner problem with arbitrary costs can be solved in linear time if the height of the grid is only 2. This solution is an extension of the dynamic programming procedure described in [1].

2.6 Determination of the supercell capacities.

Suppose that at a certain level of hierarchy our 2*N wiring grid consists of supercells each containing several final grid cells.

![Figure 11. 2*N grid supercells and a part of the final grid.](image)

The boundary of a final grid cell coincides with a supercell boundary if geometrically the boundary of the final grid forms a part of the supercell boundary. A coincident cell of a supercell boundary is a final grid cell, element of this supercell which cell boundary coincides with the supercell
boundary. The number of coincident final grid cells of a supercell boundary is
the number of coincident cells (number of final grid cells with coincident
boundaries with the supercell boundary). Now we have to determine the channel
and via capacities of our supercells. Consider the supercells depicted in fig
11. For the horizontal channel capacity $H_l$ in our supercell we could of course
choose the horizontal channel capacity $h(l,k)$ of the final grid cell boundary
which coincide with our super cell boundary. This definition of the channel
capacity may be adequate, provided that the final grid capacity distribution is
uniform. It may not be so in case of nonuniform channel capacity distribution
in the final grid because the channel capacities of the boundaries parallel to
those of our coincident boundary $h(l,k)$ at final grid solution disappear from
our consideration. It may be that those capacities are much less then those of
the coincident boundary. Consider for instance picture 11 where the horizontal
channel capacity $H_l$ has to be determined. Denote $h(i,j)$ is the horizontal
channel capacity between the final grid cells $G(i,j)$ and $G(i,j+1)$, $k$ is the
boundary of the supercell under consideration and $L$ and $R$ are the coincident
boundaries at final grid resolution of respectively the left and right
supercell boundary with respect to $k$. It may happen that the value of $h(l,k)$
is much larger than for example $h(l,k-1)$, but the procedure for solution of the
supercell wiring will not see the small value of $h(l,k-1)$; it may force a large
number of nets to cross the boundary which coincides with $h(l,k)$ and those nets
will overflow at the lower levels of hierarchy because the small value of
$h(l,k-1)$. In order to avoid this we use the following heuristic scheme for the
determination of the factorised channel capacity of the final grid cell
boundary which coincides with the boundary of a supercell.

First of all we compute the quantities $h^*(i,j)$ for all final grid cell
boundaries which are corrected with the channel capacities nearer to the
coincident boundary.

\[
\begin{align*}
  j = k & : h^*(i,k) = h(i,k) \\
  L < j < k & : h^*(j,k) = \min \{ h(i,j), h^*(i,j+1) \} \\
  k < j < R & : h^*(j,k) = \min \{ h(i,j), h^*(i,j-1) \}
\end{align*}
\]

Now the factorised horizontal capacity $H_l$ is determined by

\[
H_l = \frac{\sum_{j=L+1}^{R-1} h^*(i,j)}{R - L - 1}
\]
Consider now the vertical channel capacities. The factorised channel capacities of the coincident boundaries are not corrected. To determine the vertical channel capacity $V_I$ we could of course sum these factorised capacities $v^*(k+l)\ldots v^*(R)$ but this is only justified if the distribution of wires which cross the boundary $V_I$ is uniform otherwise several overflows are created at lower levels of hierarchy on this boundary. To take into account this fact a correction factor is introduced which is a function of the number of coincident final grid cells of this supercell boundary (in our case 3). The more final grid cells are used the more the summed calculated capacity should be decreased to compete with the likelihood of the nonuniform distribution of wires crossing this boundary at lower levels of hierarchy. This is a conservative heuristic and we use for this correction factor:

$$C(x) = \frac{2x + 1}{3x} \quad (3.5)$$

where

$x$ is the number of coincident final grid cells.

So we use actual channel capacities when our supercells become actual final grid cells at the lowest level of hierarchy, but when the number of final grid cells in a supercell is large the amount of channel capacity assigned is slightly more than two-thirds of the actual channel capacity of that boundary.

$$\begin{array}{c}
\text{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx} \\
\text{x----------x----------x} \\
\text{x| x| x| x} \\
\text{x----------x----------x} \\
\text{x| x| x| x} \\
\text{x| x| x| x} \\
\text{x---------x---------x} \\
\text{x| x| x| x} \\
\text{x---------x---------x} \\
\text{x| x| x| x} \\
\text{x---------x---------x} \\
\text{x| x| x| x} \\
\text{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
\end{array}$$

**Figure 12.** 2*N supercell with several final grid cells.

The situation depicted in fig. 12 becomes more complicated. Each 2*N grid cell contains 9 final grid cells, each 2*N cell boundary coincides with 3 final grid cell. The heuristics used to determine the factorised capacities of the 2*N supercell are:

- For each final grid cell boundary coinciding with a 2*N cell edge the factorised capacity is given by eq. (3.4).

- The channel capacity of the 2*N cell boundary is calculated by summing the calculated factorised capacities of the coincidence boundaries and
multiplying these by the correction factor in eq. (3.5).

After calculating the channel capacities of the 2*N grid the via capacity of each 2*N cell is determined by using the function $V(n,m)$ used in paragraph 3.2. If the 2*N cell contains two opposite boundaries with channel capacities $n_1$ and $n_2$ then the maximum of $n_1$ and $n_2$ is taken as the number of wiring tracks available through the cell.

3. 2*N WIRING PROBLEM, LINEAR PROGRAMMING APPROACH.

3.1 Introduction.

This chapter describes the solution to solve a general 2*N wiring problem with the linear programming approach. This solution is based on the exact solution of the wiring problem in case of a 2*2 grid and the hierarchical extension to a 2*N grid in a divide and conquer fashion. To solve the linear programming problem the simplex algorithm and the all-integer algorithm are used.

3.2 Restrictive wiring.

In the general 2*N graph we assumed that all possible tree structures were admissible for wire routes. But to be consistent with our general philosophy we also introduce a restrictive 2*N wiring where a wire route is not allowed to cross simultaneously the upper and lower part of a split vertical boundary.

- $X$: interconnection pins located in grid cell
- $*$: interconnect

(a) Wiring solution at the previous hierarchical level.

(b) Initial terminal cells at current hierarchical level.
Figure 13. Prohibited wiring solution within a 2*N grid.

We consider this restrictive case because for a net has been routed at the previous level of hierarchy we accounted only one crossing for a wire segment which crossed a vertical boundary of cell G. This cell is now partitioned into two cells G- and G+. Suppose that on the previous level of hierarchy a net was taken the route depicted in fig 13(a). All channel capacities of the boundaries the net is crossing, might be exhausted at this stage. After partitioning we get our new 2*N wiring problem as depicted in fig 13(b). If it is allowed to cross the old boundaries more than once a solution with simultaneously crossing the upper and lower vertical boundary may block subsequent wiring problems by running out of tracks.

3.3 Four cell wiring problem description.

Consider the simplest particular case of the 2*N wiring problem: wiring within a 2*2 grid. In this case it is easy to enumerate all possible types of nets that may occur, depending on the distribution of terminal cells. Clearly the nets having only one terminal cell have not to be wired at this stage so we assume that the number of terminal cells may be 2, 3 or 4:

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
<th></th>
<th>X</th>
<th>X</th>
<th></th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Types of 2-terminal nets:
types of 3-terminal nets:

+---+---+ +---+---+ +---+---+ +---+---+
| X | X | | X | X | | X | X |
| X | X | | X | X | | X | X |
| 7 | 8 | 9 | 10 |

type of 4-terminal nets:

+---+---+
| X | X |
| X | X |
| 11 |

Figure 14. Enumeration of all possible terminal configurations within a 2x2 grid.

There are only six different types of 2-terminal nets, four different types of 3-terminal nets and only one type of 4-terminal nets. All different terminal configurations are depicted in fig 14.

For each terminal configuration type there is a limited (and small) number of ways it can be routed:

TYPE 1

+---+---+
| X | X |
| X | X |

P(1,1) P(1,2) #

TYPE 2

+---+---+
| X | X |
| X | X |

P(2,1) P(2,2) #

TYPE 3

+---+---+
| X | X |
| X | X |

P(3,1) P(3,2)
TYPE 4

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P(4,1)  P(4,2)

TYPE 5

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

P(5,1)  P(5,2)

TYPE 6

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

P(6,1)  P(6,2)

TYPE 7

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P(7,1)  P(7,2)  P(7,3)

TYPE 8

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P(8,1)  P(8,2)  P(8,3)

TYPE 9

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P(9,1)  P(9,2)  P(9,3)

TYPE 10

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P(10,1)  P(10,2)  P(10,3)

7.71
Figure 15. Enumeration of all wiring possibilities of the different types.

For every net of type 1 there are only two wiring possibilities: pattern P(1,1) and pattern P(1,2) depicted in Fig 15. Similarly enumeration of all wiring possibilities for all other terminal configuration types leads to the wiring patterns depicted in Fig 15.

Let

- $K(i)$ is the number of nets of type $i$ ($i = 1, 2, \ldots, 11$).
- $X(i,j)$ is the number of nets of type $i$ which is wired in the $j$-th possibility, i.e., the number of nets routed in pattern fashion $P(i,j)$.

After determination of $K(i)$ of the $2 \times 2$ wiring problem our initial wiring problem is now reduced to the question: for $i = 1, \ldots, 11$ how many nets of type $i$ will be realised in which wiring possibility. Determination of $X(i,j)$ represents our wiring problem.

3.4 Four cell wiring.

The values of $X(i,j)$ can be determined by transformation in an integer programming problem which can be solved by the simplex algorithm or all-integer algorithm.

First of all the wiring problem may be non solvable, i.e., the solution may result in overflows, nets which can not be routed under the given restrictions. Overflowed nets are still classified by their terminal configuration types.

Let $X(i)$ denote the number of overflowed nets of type $i$ ($i = 1, 2, \ldots, 11$). We shall now derive all linear restrictions which the solution must satisfy:

admissible $i,j$ $X(i,j) \geq 0$ (4.1)
\[ i = 1, 2, \ldots, 11 \quad X(i) + \sum_{j} X(i,j) - K(i) \quad (4.2) \]

\[ \text{Figure 16.} \quad 2 \times 2 \text{ grid with the channel and via capacities.} \]

We have the following channel capacity restrictions:

Denote

- \( h_1 \) the channel capacity between cells A and B.
- \( h_2 \) the channel capacity between cells C and D.
- \( v_1 \) the channel capacity between cells A and C.
- \( v_2 \) the channel capacity between cells B and D.

We start with the enumeration of the wiring patterns which affect the boundary capacities.

Denote

- \( H_1 = \{ P(i,j) \mid P(i,j) \text{ crosses the upper vertical boundary} \} \)
- \( H_2 = \{ P(i,j) \mid P(i,j) \text{ crosses the lower vertical boundary} \} \)
- \( V_1 = \{ P(i,j) \mid P(i,j) \text{ crosses the left horizontal boundary} \} \)
- \( V_2 = \{ P(i,j) \mid P(i,j) \text{ crosses the right horizontal boundary} \} \)

then

- \( H_1 = \{ P(1,2), P(2,2), P(3,2), P(4,1), P(5,1), P(6,2), \)
  \( P(7,1), P(7,3), P(8,2), P(8,3), P(9,1), P(9,3), \)
  \( P(10,2), P(10,3), P(11,1), P(11,3), P(11,4) \} \)

- \( H_2 = \{ P(1,2), P(2,2), P(3,1), P(4,2), P(5,2), P(6,1), \)
  \( P(7,2), P(7,3), P(8,1), P(8,3), P(9,2), P(9,3), \)
  \( P(10,1), P(10,3), P(11,2), P(11,3), P(11,4) \} \)
\[ V_1 = \{ P(1,1), P(2,2), P(3,2), P(4,2), P(5,1), P(6,1), P(7,2), P(7,3), P(8,2), P(8,3), P(9,1), P(9,2), P(10,1), P(10,2), P(11,1), P(11,2), P(11,3) \} \]

\[ V_2 = \{ P(1,2), P(2,1), P(3,2), P(4,2), P(5,2), P(6,2), P(7,1), P(7,2), P(8,1), P(8,2), P(9,2), P(9,3), P(10,2), P(10,3), P(11,1), P(11,2), P(11,4) \} \]

The channel capacity restrictions requires that the number of wires crossing a boundary should be less or equal than the channel capacity at that boundary which imply the following set of inequalities:

\[
\sum_{P(i,j) \text{ elem of } H_1} X(i,j) \leq h_1
\]

\[
\sum_{P(i,j) \text{ elem of } H_2} X(i,j) \leq h_2
\] (4.3)

\[
\sum_{P(i,j) \text{ elem of } V_1} X(i,j) \leq v_1
\]

\[
\sum_{P(i,j) \text{ elem of } V_2} X(i,j) \leq v_2
\]

Here \( h_1, h_2, v_1 \) and \( v_2 \) stand for respectively \( h(2,1), h(1,1), v(1) \) and \( v(2) \) as defined in paragraph 3.6.

And the via capacity restrictions:

Denote

\[
\text{via}(1,2) \text{ denote the via capacity in cell A} \\
\text{via}(2,2) \text{ denote the via capacity in cell B} \\
\text{via}(1,1) \text{ denote the via capacity in cell C} \\
\text{via}(2,1) \text{ denote the via capacity in cell D}
\]

Enumeration of the wiring patterns which affect these via capacities implies the following sets:
Denote
A - \{ P(i,j) | P(i,j) bends in cell A with no terminal in cell A \}
B - \{ P(i,j) | P(i,j) bends in cell B with no terminal in cell B \}
C - \{ P(i,j) | P(i,j) bends in cell C with no terminal in cell C \}
D - \{ P(i,j) | P(i,j) bends in cell D with no terminal in cell D \}

then

A - \{ P(2,2), P(3,2), P(5,1), P(8,2), P(8,3) \}
B - \{ P(1,2), P(3,2), P(5,2), P(10,2), P(10,3) \}
C - \{ P(2,2), P(4,2), P(5,1), P(7,2), P(7,3) \}
D - \{ P(1,2), P(4,2), P(5,2), P(9,2), P(9,3) \}

The via restrictions requires that the number of wires bending within a cell should not exceed its via capacity which implies the following set of inequalities:

\[
\sum X(i,j) \leq \text{via}(1,2) \quad \text{P(i,j) elem of A}
\]
\[
\sum X(i,j) \leq \text{via}(2,2) \quad \text{P(i,j) elem of B}
\] \( (4.4) \)
\[
\sum X(i,j) \leq \text{via}(1,1) \quad \text{P(i,j) elem of C}
\]
\[
\sum X(i,j) \leq \text{via}(2,1) \quad \text{P(i,j) elem of D}
\]

So our final solution must satisfy the equalities (4.2) and the inequalities (4.1),(4.3),(4.4).

We now derive an integer linear programming problem which satisfies all these restrictions.

By introducing artificial or slack variables the inequalities are transformed into equalities:

7.75
Denote

- $Y(1)$ the number of the leftover wiring tracks on the $h_1$ boundary
- $Y(2)$ the number of the leftover wiring tracks on the $h_2$ boundary
- $Y(3)$ the number of the leftover wiring tracks on the $v_1$ boundary
- $Y(4)$ the number of the leftover wiring tracks on the $v_2$ boundary

Then the channel capacity restrictions are rewritten as:

\[
\sum_{(i,j)} X(i,j) + Y(1) = h_1 \\
P(i,j) \text{ elem of } H_1
\]

\[
\sum_{(i,j)} X(i,j) + Y(2) = h_2 \\
P(i,j) \text{ elem of } H_2
\] (4.5)

\[
\sum_{(i,j)} X(i,j) + Y(3) = v_1 \\
P(i,j) \text{ elem of } V_1
\]

\[
\sum_{(i,j)} X(i,j) + Y(4) = v_2 \\
P(i,j) \text{ elem of } V_2
\]

Denote

- $Z(1)$ the number of leftover vias in cell $A$
- $Z(2)$ the number of leftover vias in cell $B$
- $Z(3)$ the number of leftover vias in cell $C$
- $Z(4)$ the number of leftover vias in cell $D$

Then the via capacity restrictions are rewritten as:

\[
\sum_{(i,j)} X(i,j) + Z(1) = \text{via}(1,2) \\
P(i,j) \text{ elem of } A
\]

\[
\sum_{(i,j)} X(i,j) + Z(2) = \text{via}(2,2) \\
P(i,j) \text{ elem of } B
\] (4.6)

\[
\sum_{(i,j)} X(i,j) + Z(3) = \text{via}(1,1) \\
P(i,j) \text{ elem of } C
\]
\[ \text{Sum } X(i,j) + Z(4) = \text{via}(2,1) \]
\[ P(i,j) \text{ elem of } D \]

This forms a set with 19 equations and 47 variables.

We now have to construct the objective function.

The goals are in descending order of priority:

1. To derive a wireable solution or the least possible number of overflows under the given circumstances. (In wireable cases when no overflows occur \( X(i) \), \( i = 1, 2, \ldots, 11 \) have to be zero.)

2. The minimisation of the total wire length and an even distribution of the wire tracks left over on both horizontal boundaries. This is also the case for the vertical boundaries. Traditionally the objective function contains only the total wire length defined in some manner (usually the Manhattan distance). The minimal total wire length may create spots of congestion when wires crowd on one part of the cut line. Clearly this wiring congestion will contribute to break down the global router at lower hierarchical levels or in a later stage the local router by attempting to embedding the wires on the final tracks. By the manner of partitioning the \( 2 \times N \) grid to get a \( 2 \times 2 \) grid the numbers of the channel capacities \( v_1 \) and \( v_2 \) and the numbers of the channel capacities \( h_1 \) and \( h_2 \) have approximately the same value (see par 3.6).

3. The minimisation of the number of vias used and an even distribution of the vias leftover. The via capacities of cell A and cell C and those of cell B and cell D usually have approximately the same value.

ad 1) In wireable cases all \( X(i) \), \( i = 1, \ldots, 11 \) have to be zero, so:

\[
\begin{align*}
\text{minimise } & \text{Sum } X(i) \\
& i = 1 \ldots 11
\end{align*}
\]  

(4.7)

ad 2) A good parameter for the total unused wire length is the sum of all leftover tracks. To get an even distribution of the leftover tracks, we want that the values of \( Y(1) \) and \( Y(2) \) and those of \( Y(4) \) and \( Y(4) \) have approximately the same value. The object function must be cope with these two goals. This can be accomplished by introducing a new artificial variable and split the old variables \( Y(1) \) and \( Y(2) \) into two
variables whereby one variable is the new introduced variable, which is the minimum of \( Y(1) \) and \( Y(2) \) and serves as a coupling.

We substitute

\[
Y(1) - Y(1)' + MY
Y(2) - Y(2)' + MY
\]

Assume that the old objective function was

\[
f = c_1 \cdot Y(1) + c_2 \cdot Y(2)
\]

and the new one

\[
f = c_1 \cdot Y(1)' + c_2 \cdot Y(2)' + c_3 \cdot MY
\]

In order to guarantee that \( MY \) will be the minimum of the old variables \( Y(1) \) and \( Y(2) \) the cost coefficient \( c_3 \) have to be greater than \( c_1 + c_2 \).

The linear program maximises the object function \( f \) so an increases of one of both \( Y(1)' \) and \( Y(2)' \) will increases \( f \) by \( c_1 + c_2 \), an increase of one of \( MY \) will increase \( f \) by \( c_3 \) which is more as \( c_1 + c_2 \). So first \( MY \) is increased as much as possible and thereafter \( Y(1)' \) or \( Y(2)' \). We rewrite the eq. (4.5):

Substitute:

\[
Y(1) = Y(1)' + MH \\
Y(3) = Y(3)' + MV \\
maximise \quad c_1 \cdot Y(1)' + c_1 \cdot Y(3)' + MVC
\]

\[
Y(2) = Y(2)' + MH \\
Y(4) = Y(4)' + MV \\
maximise \quad c_1 \cdot Y(2)' + c_2 \cdot MH + c_2 \cdot MV
\]

Where \( MH \) is the mean term in \( Y(1) \) and \( Y(2) \) in horizontal direction and \( MV \) is the mean term in \( Y(3) \) and \( Y(4) \) in vertical direction. The ratio \( c_2/c_1 \) determines which goal deserves the highest priority, the even distribution or the maximisation of the unused tracks.

ad 3) Minimisation of the number of vias and an even distribution.

7.78
Substitute:
\[ Z(1) - Z(1)' + ML \quad , \quad Z(2) - Z(2)' + ML \]
\[ Z(3) - Z(3)' + MR \quad , \quad Z(4) - Z(4)' + MR \]

maximise \[ c_3 . Z(1)' + c_3 . Z(2)' + c_4 . ML \]
maximise \[ c_3 . Z(3)' + c_3 . Z(4)' + c_4 . MR \]

Where ML is the mean term in \( Z(1) \) and \( Z(2) \) in horizontal direction and MR is the mean term in \( Z(3) \) and \( Z(4) \) in vertical direction.

The total forms now a set with 19 equations and 51 variables. The object function which must be maximised:

\[
\begin{align*}
11 & \sum X(i) + 4 \sum Y(i)' + c_3 (MH + MV) + \\
& + c_4 \sum Z(i)' + c_5 (ML + MR) \\
& i = 1, \quad i = 1
\end{align*}
\]

\[(4.8)\]

where
- \( c_1 < 0, \ c_2, \ c_3, \ c_4, \ c_5 > 0 \)
- \( |c_1| >> c_2, \ c_3 > 2.c_2, \ c_2 > c_4, \ c_5 > 2.c_4 \)

Remarks:
- The most significant part of this reduction is that the size of the integer programming problem is always fixed and independent of the number of nets to be wired.
- The complexity of this solution is \( C + O(m) \) where \( m \) is the number of nets. We have to get the just terminal configuration types of all the \( m \) nets and the integer programming problem will run in a constant time \( C \).
- In the existing implementation we first substitute the integer programming problem by the corresponding linear programming problem, invoke standard solutions because of speed (Simplex algorithm). If in the final result some of the values of \( X(i,j) \) are not integer valued we solve the problem with the all-integer algorithm which yields the optimal result of the objective functions where all variables have to be integer valued.
3.5 Restrictive four cell wiring.

As mentioned before, we have to consider the restrictive 2*2 wiring problem where nets are not allowed to cross simultaneously the vertical boundaries h1 and h2. We achieve this by forcing several variables in our Integer Programming Problem of the previous section to be zero. Namely we forbid the following wiring patterns:

\[ P(1,2), P(2,2), P(7,3), P(8,3), P(9,3), P(10,3), P(11,3), P(12,3) \]

In our linear program the corresponding variables \( X(i,j) \) are forced to remain zero in the linear programming problem (denoted by '#' in fig 15). The rest of our system remains the same which yields in this case a set with 17 equations and 43 variables. In fact only the first four cell problem at the highest level of hierarchy is the only problem where all wiring patterns are permitted (no restrictive wiring).

3.6 2*N wiring algorithm in divide and conquer extension.

We now describe a heuristic algorithm to derive the solution for a general 2*N global wiring problem based on 2*2 grid solutions. This procedure is recursive and based on the "divide and conquer" approach carried out by the breadth first search technique. In a hierarchical way the 2*N grid is partitioned in several 2*2 wiring problems which are successively solved.

(a) Initial terminal cells of a net.

(b) Net in the factorised (2*2) wiring problem.

(c) Route taken by the net in the (2*2) wiring.
(d) Reduction to two (2*4) subproblems.

(e) Factorised (2*2) subproblems for these (2*4) problems.

(f) Routes taken by the pieces of the net at this level.

(g) Reduction to four (2*2) subproblems.

(h) Routes taken by the pieces of the net.

(i) Resulting route in the original (2*8) wiring problem.

Figure 17. Divide and conquer wiring example.

We start with the initial 2*N wiring problem as described in chapter 2. First we partition the 2*N grid into two more or less equal parts, generally in half and construct a "factorised 2*2 wiring problem" considering only the new introduced vertical cut line and the horizontal cut line as cell boundaries. For each net the supercells in our four cell wiring problem which contain terminal cells in the original 2*N wiring problem, are summarised. In this stage only the nets with terminals in more than one supercell have to be routed.
The boundary ad the via capacities of our factorised 2*2 wiring problem are determined by heuristics out of the 2*N boundary and via capacities in the same way as described in paragraph (3.6). This 2*2 wiring problem is solved and new terminal cells are introduced adjacent to the vertical boundary in case the net has a wire interconnection crossing this boundary.

The initial problem is now reduced to two smaller sized wiring problems: a 2*N1 wiring problem and a 2*N2 wiring problem where N1 + N2 = N. This reduction is based on the introduction of new terminal cells in the main 2*N wiring problem adjacent to the vertical cut line for every net with a wire interconnection crossing it. This procedure is repeated until our supercells in 2*2 wiring problems become actual cells of the initial 2*N wiring problem (N1-1, N2-1). At the lowest hierarchical level (when our supercells become final grid cells) our factorised boundary and via capacities are actual 2*N boundary and via capacities. An example is depicted in fig 17.

3.7 Pattern assignment and overflow handling.

The solution of the integer programming problem indicates for a given net type how many nets of this type are routed in which way. We still have to decide finally how the nets will be routed. The choice could be an heuristic attempt to make nets shorter. For example if there are K(4) nets of type 4 and X(4,1) have to routed straight and X(4,2) must be detoured, then we could select the nets having only interconnections terminals located "far above" for straight interconnection. In the current program this heuristic is not used and the pattern assignment is carried out in a straight forward manner by the number of appearance in the four cell problem table. The reasons not using the heuristics are:

- The heuristic is only powerful at a very high level of hierarchy where much wire length can be spared.

- Except the first cell wiring problem always restrictive wiring is performed, so the patterns denoted by '#' in fig 15, are not allowed to use. Of the remaining patterns several possibilities have the same wire length (For example patterns P(5,1) and P(5,2)).

- By means of the objective function we try to minimise the total wire length. In most solutions pattern possibilities with minimal number of boundary crossings occur.

- In order to derive a good heuristic, interconnections to the outside area of the four cell area should be considered too, resulting in a decent
overhead of computation time to make proper pattern assignments.

Overflows occur when under the given circumstances the router is unable to embed all wires. All wire interconnections of the overflowed net are removed, only its interconnection terminals remain resident in the data base. At the next levels of hierarchy the overflowed net is not routed further to decrease the likelihood of overflows at these levels.

Remarks:
- All existing interconnections of overflowed nets are removed because the a bisection hierarchical step cannot handle "dangling ends". A dangling end is a wire segment not ending in a terminal cell. If a supercell consists of a dangling end then at the next level of hierarchy after partitioning this supercell in two new supercells one must arbitrarily selected to be terminal cell of set A.
- Only the terminal configuration type of the overflowed net is known. An attempt to route the net would result in a complicated pattern selection and assignment rule and the likelihood of occurring overflows at the next level s of hierarchy.
- In the divide and conquer approach an overflow of a net having only terminal cells located in the left part or in the right part (type 1 or 2) do not automatically result in a real overflow at high level in the breadth first search because in that stage only horizontal wire connections are made.
- After completion of the router it is possible to embed the overflowed nets by :
  - clean up maze running technique
  - manual editing

3.8 The linear programming problem.

The problem of optimisation deals how to do things in the best possible manner. The function which describes the return of benefit from any proposed solution is called the objective function. Other names commonly used are the cost function and the performance function. For the special case when both the system model and the objective function are linear equations, we have a linear programming problem.

In most optimisation problems having physical significance, the solution must
satisfy restrictions in the model describing the physical system as well as maximising or minimising the objective function. Problems of this sort are termed constrained optimisation problems. Constraints are further classified as equality constraints or inequality constraints.

The set of all possible values which satisfy all the constraints is termed a feasible set. Any element in this set will be called a feasible point.

One class of problems where this restriction is violated is designated as integer programming. These problems require the solution be chosen from the set of integers rather than the set of real numbers.

A linear program is a problem of minimising or maximising a linear function subject to linear constraints where these constraints may both include inequalities and equalities and the unknown variables may include nonnegative variables and variables that are unrestricted in sign. In linear programming, both the objective function and the constraint equations are linear in the independent variables.

The problem of linear programming is to maximise a linear function subject to linear inequalities, i.e. we want to maximise:

\[ f = \sum_{j=1}^{n} c_j x_j \]  \hspace{1cm} (4.9)

subject to

\[ \sum_{i=1}^{m} a_{i,j} x_j \leq b_i \] \hspace{1cm} (4.10)

and

\[ x_j \geq 0 \] \hspace{1cm} (4.11)

where \( x_j \) are unknown variables and \( a_{i,j}, b_i \) and \( c_j \) are given constants.

Equivalent formulations:

- Maximising a function is equivalent to minimising the negative of that function so we can replace eq. (4.9) by minimise \( f = -\sum_{j=1}^{n} c_j x_j \).

- In an inequality constraint we can introduce a new unknown variable \( s_i \) (\( s_i \geq 0 \)), called a slack variable and convert the inequality into an equality:

\[ \sum_{i=1}^{m} a_{i,j} x_j + s_i = b_i \] \hspace{1cm} (4.12)

A system of linear inequalities is consistent if it has at least one solution.

The system is redundant if one of the inequalities is implied by the other
inequalities.

Let $A\tilde{x} \leq b$ ($\tilde{x} \geq 0$) be the system of inequalities where $A$ is an $m \times n$ matrix. These $m$ inequalities define a nonempty convex set if they are consistent.

A feasible solution $\tilde{x}^*$ is called an optimum solution if $(\tilde{c}, \tilde{x}^*) \geq (\tilde{c}, \tilde{x})$ for all feasible solutions $\tilde{x}$, and $(\tilde{c}, \tilde{x}) < \infty$. It can be shown that if there exist an optimum solution, then there exist a basic optimum solution.

By assumption, the optimum solution is finite so that an extreme point corresponds to a basic optimal solution.

The set of all feasible solutions is defined by

$$M = \{ \tilde{x} | \tilde{x} \text{ elem of } E, A\tilde{x} \leq b, \tilde{x} \geq 0 \}$$  \hspace{1cm} (4.13)

and the linear programming problem consists of finding an extreme value of $f$ on $M$. To be specific we shall consider $\max \tilde{x} \text{ elem of } M f(\tilde{x})$ in the rest of this chapter.

The linear programming problem is said to be in normal form if $b \geq 0$ in eq. (4.10) in which case $0 \text{ elem of } M$, ensuring that $M$ is nonempty. Since an arbitrary linear programming problem can always be transformed into this form, we will restrict our discussion to linear programming in the normal form.

The set of all feasible solutions can now be defined as

$$X = \{ \tilde{x} | \tilde{x} \text{ elem of } E, A\tilde{x} = b, \tilde{x} \geq 0 \}.$$  \hspace{1cm} (4.14)

The linear programming problem can be alternatively stated as $\max \tilde{x} \text{ elem of } X (\tilde{c}, \tilde{x})$.

Basic feasible solutions.

We assume that the linear inequality constraints have been transformed into the equality constraints $A\tilde{x} = b$ by means of slack variables and that the linear program is in the normal form, i.e. $b \geq 0$. The equality constraints $A\tilde{x} = b$ can be rewritten as
\begin{align*}
  a_{1,1} \cdot x_1 + \ldots + a_{1,n} \cdot x_n = b_1 \\
  \vdots \\
  a_{m,1} \cdot x_1 + \ldots + a_{m,n} \cdot x_n = b_m
\end{align*}
\text{(4.15)}

where $a_{i,j}$ denotes the $(i,j)$th element of the $m \times n$ matrix $A$ and $x_j$ and $b_j$ are the $j$-th components of respectively the vectors $\mathbf{x}$ and $\mathbf{b}$ in eq. (4.10). We have $m$ linear algebraic equations in the unknowns $x_1, \ldots, x_n$. If we set $n - m$ of the $n$ unknown variables equal to zero then from eq. (4.10) we obtain $m$ equations in $m$ unknowns. Then (4.10) can be solved to obtain the unique values of $x_1, \ldots, x_m$ provided that the column vectors

\begin{align*}
  a_{1,1} & \quad a_{1,n} \\
  \vdots & \quad \vdots \\
  a_{m,1} & \quad a_{m,n}
\end{align*}
\text{(4.16)}

are linearly independent and hence form a basis in $\mathbb{R}^m$.

Such a solution of $A$ called a \textit{basic feasible solution} and the corresponding $x_1, \ldots, x_m$ are called the \textit{basic variables}. Next we observe that a basic solution is not always a feasible solution, since we may have $x_j < 0$, $1 \leq j \leq m$. If $x_j > 0$ for all $j$, $1 \leq j \leq m$, then we have a \textit{non degenerate} solution. If one or more of the basic variables are zero, we call the solution \textit{degenerate}.

\textbf{Theorems :}

- If there exist a feasible solution, there is a basic feasible solution.

- If there is an optimum solution, there is a basic optimum solution.

3.9 \textbf{Simplex algorithm}.

Consider a linear program in standard form.
maximise \( f = c_j x_j \quad (j = 1, \ldots, n) \)

subject to
\[
\begin{align*}
    a_{i,j} x_j - b_i & \quad (i = 1, \ldots, m) \quad (m \leq n) \quad (4.17) \\
x_j & \geq 0, \quad b_i \geq 0
\end{align*}
\]

When the objective function is linear, the set \( M \) is a convex polyhedron, and an optimum solution exists, then at least one of the vertices of \( M \) will correspond to the optimal solution. So in order to get an optimum solution, we can choose \( m \) columns from the matrix \( A \) and solve the \( m \) simultaneous equations for the \( m \) variables. Therefore a linear programming problem can be solved by choosing \( m \) columns from the matrix \( A \), solve the \( m \) simultaneous equations for the \( m \) variables and compare the objective functions to locate the optimum solution. But this approach requires that \( n \) over \( m \) faculty sets of simultaneously equations must be solved. This is not practical except for very small \( n \). The most practical method is the simplex method (developed by Dantzig), which involves a very small number of steps to solve a linear program.

Remarks:

- Given a system of simultaneous equations, there is a solution space associated with it. The solution is the set of points which satisfies these simultaneous equations. When the number of variables is equal to the number of equations and the determinant of the coefficient matrix is nonsingular, the solution is a point, and there is a unique solution. When the number of variables is greater than the number of equations, the solution space is generally a line, a plane or a higher dimensional space. Two systems of simultaneous equations are said to be equivalent if they have the same solution space. Gauss elimination does not change the solution space of a system of simultaneous equations.

- If we started with a feasible solution the feasibility has to be maintained throughout the process.
Assume that \( \bar{x}^0 \) is a basic feasible solution which implies that \( x_i^0 \geq 0, \quad i = 1, \ldots, m, \quad x_i^0 = 0, \quad i = m+1, \ldots, n \) and the column vectors \( \bar{a}^i \) are linearly independent and form a basis in \( E \). Now every vector \( \bar{a}^j \) element of \( E \) can be represented as
where \( \hat{a}^i \) are used as the basis vectors. Since \( x^0 \) is a basic feasible solution of eq. (4.17)

\[
\sum_{i=1}^{m} x_i a^i = b \quad (4.19)
\]

holds and using the representation of \( \hat{b} \) and \( \hat{a}^j \):

\[
\sum_{i=1}^{m} x_i a^i + T \sum_{i=1}^{m} L_i a^i = b \quad (4.20)
\]

or

\[
\sum_{i=1}^{m} (x_i - T L_i) a^i + T \hat{a}^j = b \quad (4.21)
\]

If we slowly increase \( T \geq 0 \) while maintaining \( x^0_i - T L_i \geq 0 \) for \( i = 1, \ldots, m \) then the vector \( \hat{a}^j \) becomes increasingly significant in the representation of \( \hat{b} \). Letting \( x_k^0 / L_{k,j} = \min(x_i^0 / L_{i,j}) \) for all \( L_{i,j} > 0 \) and set \( T = x_k^0 / T_{k,j} \) in eq. (4.21) we observe that the coefficient of \( \hat{a}^k \) becomes zero, i.e. \( \hat{a}^k \) in eq. (4.21) is replaced by \( \hat{a}^j \).

If \( x^1 \) denotes the vector whose m nonnegative components are given by the coefficients in eq. (4.21) then \( x^1 \) is a new basic feasible solution obtained from \( x^0 \).

- Suppose having a system into diagonal form with respect to a set of m x's, which mean that the m*m identity matrix is a submatrix of A and correspond with the x's. Could we get an f equation with

\[
f + a_{0,m+1} x_{0,m+1} + \ldots + a_{0,n} x_{0,n} = a \quad (4.22)
\]

where

\[
a_{0,j} \geq 0, (j = m+1, \ldots, n)
\]

Note that the solution of a system of linear equations is not changed by
Gauss elimination, thus a solution satisfying eq. (4.17) is also a solution of the original system. Since all variables are nonnegative we conclude that $a_{0,0}$ is the upper bound of $f$ without considering the other equations of the with respect to $f$, $x_1, \ldots, x_m$. (Constraints cannot increase the upper bound when we want to maximise a function.) If the other equations of the diagonal form give $x_j \geq 0, j = 1, \ldots, m$ (a feasible solution) then $a_{0,0}$ is indeed the maximum value of $f$.

- Since we change from one basis to another, and there is a finite number of bases, in fact at most $n$ over $m$ faculty, this would guarantee finiteness unless a basis can be repeated. If the value of $f$ is always increasing strictly, then a different value of $f$ implies different bases, and this will guarantee finiteness. However in a problem at some stage we may have $b_i = 0$. Since the value of the objective function is changed by $c_j$ is minimum $(b_i / a_{i,j})$ to preserve the feasibility the change may be zero. This means that the value of the objective function may not change after the pivot operation. If several bases all correspond to the same value of $f$ it is possible to cycle among these bases.

- Suppose that we have chosen a variable with positive coefficients in the $f$ equation and there is no positive constant in the column corresponding to that variable. Then we can make that variable arbitrarily large without making other variables negative. This makes $f$ as positive as one wants. Therefore the program has no finite optimum solution.

Part of the simplex method is an procedure for moving from any vertex of M to a neighbouring vertex via the associated basic feasible solutions. The simplex algorithm does not evaluate the objective function at every vertex of $M$, but rather economises the search for the optimum solution by selecting certain promising vertices.

The simplex method will generate a vertex of $M$ only if the value of $f$ at that vertex is expected to be at least as large as the value already computed.

We assume that the problem of linear programming is $\max (\vec{c}, \vec{x})$ with $\vec{x}$ elem of $X$, as stated in the previous section. Let $\vec{x}^0$ denote the initial basic feasible solution, and let $\hat{x}$ be the new basic feasible solution obtained from $\vec{x}^0$ by using (4.21). Then evaluating the objective function at $\hat{x}$, we have
\[ (c, x) = \sum_{i=1}^{m} (x - T \cdot L) \cdot c_{i,j} + T \cdot c \]
\[ - \sum_{i=1}^{n} c \cdot i,j \cdot i - 1 \]
\[ = (c, x) - T \cdot \{ z - c \} \]

where
\[ m \]
\[ z = \sum_{j=1}^{n} L_{i,j} \cdot c_{i,j} \]

The maximisation of the objective function requires an evaluation of the following three cases:

case 1: There exists a \( j_0 \) between \( m+1 \) and \( n \) such that \( z_{j_0} - c_{j_0} < 0 \) and \( L_{i,j_0} > 0 \) for some \( i \). Letting \( T = \min(x_{i_0}^0/L_{i,j_0}) \), we observe that \( T > 0 \) and \( x_{i_0}^0 - T \cdot L_{i,j_0} \geq 0 \) for \( i = 1, \ldots, m \), with equality holding for at least one value of \( i \). The vector \( \hat{x}^1 \) determined by this choice of \( T \) satisfies (4.23) and has at most \( m \) non-negative components, and hence this is a basic feasible solution of (4.17). From eq. (4.23), \((\hat{c}, \hat{x}^1) > (\hat{c}, \hat{x}^0)\), and thus we obtain a new basic feasible solution which provides a larger value of the objective function.

case 2: The inequality \( z_j - c_j \geq 0 \) holds for all \( j \) between \( m+1 \) and \( n \). In this case \( \hat{x}^0 \) is an optimal basic feasible solution. To prove this statement, let \( \hat{x} \) denote an arbitrary basic feasible solution of (4.17), which yields

\[ b = \sum_{j=1}^{n} x \cdot j - 1 = \sum_{j=1}^{n} x \cdot \sum_{i=1}^{m} L_{i,j} \cdot i = 1 \]
\[ a = \sum_{j=1}^{n} x \cdot j - 1 = \sum_{i=1}^{m} L_{i,j} \cdot \]
\[ \sum_{i=1}^{m} x \cdot i = \sum_{i=1}^{m} x \cdot i \]

since \( \hat{x}^0 \) is also a basic feasible solution of (4.17). Now, since \( \hat{a} \), \( i = 1, \ldots, m \), are linearly independent, it follows that

\[ \sum_{j=1}^{n} x \cdot i = \sum_{j=1}^{n} x \cdot i,j \cdot j \]

(4.25)
Furthermore, the given condition is equivalent to $z_j \geq c_j$, $j = m+1, \ldots, n$, and $z_j = c_j$, $j = 1, \ldots, m$. Hence

$$
\sum_{j=1}^n c_j \leq \sum_{j=1}^m z_j = \sum_{j=1}^m \sum_{i=1}^n L_{i,j} \cdot c_i
$$

Hence

$$
\sum_{i=1}^m \sum_{j=1}^n x_{i,j} = \sum_{i=1}^m \sum_{j=0}^n c_i
$$

i.e., for any arbitrary basic feasible solution $\bar{x}$ of eq. (4.17), $(\bar{c},\bar{x}) \geq (c,\bar{x})$ and hence $\bar{x}$ is an optimal solution.

Case 3: There exists a $j_0$ between $m+1$ and $n$ such that $z_{j_0} - c_{j_0} < 0$ and $L_{1,j_0} \leq 0, (i = 1, \ldots, m)$. In this case $x^0_{i,j} - T \cdot L_{i,j} \geq 0$, and hence $x^0$ is a basic feasible solution of eq. (4.17) for all $T > 0$. As $T$ grows to infinity, it is clear from eq. (4.23) that $(\bar{c},\bar{x})$ increases without any upper bound and hence the problem has no finite optimum solution.

It is now clear that the above given scheme generates a new basic feasible solution of eq. (4.17) such that the value of $(\bar{c},\bar{x})$ is successively increased. If the problem has finite optimal solutions, at least one of these will be found after a finite number of iterations. Otherwise case 3 will be encountered, and the iterations can be stopped.

**Geometric interpretation:**

In the simplex method we always set up a local coordinate system with the current solution as origin and move towards another neighbouring vertex along an edge which would bring an improvement to the objective function. When the new vertex is reached, we set up another coordinate system with the new vertex as origin. The equalities are stored in a tableau, at each iteration we perform Gauss elimination in the tableau.

**3.10 Primal integer programming algorithm.**

Consider a linear program

7.91
\[
\begin{align*}
\text{max } & \quad \sum_{a} x - a \\
\text{subject to } & \quad \sum_{l} x - a \\
& \quad \sum_{m} x - a
\end{align*}
\]

where
\[
x \geq 0 \quad (j = 1, \ldots, n)
\]

If we require in addition to the constraints that all variables \( x_j \) \((j = 1, \ldots, n)\) must be integers then the program is called an integer program. The term primal denotes that the equalities are consistent and we can always start with a legal solution.

The constraints in eq. (4.21) define a convex region of n dimensions. This is shown in fig. (18) OABCD is the convex region. The X's represent lattice points with integer components. Those lattice points within OABCD are then feasible solutions to the integer program. Since optimum solutions to a linear program always occur on the boundary solution space and non of the boundary points on ABCD are integer components, these are not even feasible solutions of the integer program. Suppose that the feasible region of the linear program
can be shrunk to the convex hull of its lattice points inside the feasible region. This convex hull is shown as the shaded area OEFGH. This shaded area can be thought of as a feasible region of some other linear program. In fact, if the linear program that defines the feasible region OABCD is restricted by more constraints such as RR' as shown in fig (18), then the modified linear program can have as a feasible region the shaded area. This new feasible region in the shaded area has two important features:

- It contains every feasible point with integer components of the original linear program (since it is the convex hull of those points).

- All extreme points of this new feasible region have integer components. Therefore, any basic optimum solution of the modified linear program has integer components and it is also the optimum solution to the original integer program.

The all integer algorithm denotes a method that proceeds to an optimal solution through a sequence of successively better solutions that are all feasible in the sense of satisfying both the integer and linear restrictions on the variables. The all integer programming algorithm is just a method of systematically generating additional constraints to cut the feasible region down to the convex hull of its integer feasible points which is an extension of the simplex algorithm and its basic optimum solution will automatically be an integer solution.

The all integer algorithm has the following properties:

- Generated constraints never exclude from the new feasible region an integer feasible point of the original integer program.

- In a finite number of steps enough constraints will be generated to cut the feasible region of the original program so that the optimum solution of the modified linear program has integer components. Cycling is prevented by the selection of the lexicographically smallest pivot column and a pivot row selection rule so that never the same basis can occur in the tableau.

- The generated constraints (or hyperplanes) pass through at least one integer point although not necessarily an integer point inside the convex hull and will reduce the feasible region of the original linear program. It should be emphasised at this point that before the feasible region of the original linear program is cut to its convex hull, we may obtain already the optimum solution of the original integer program. Also since
the optimum integer solution can be defined as the intersection of \( n \) hyperplanes, these \( n \) hyperplanes are actually the only ones that are really needed; some of these may be constraints of the original problem. Basic idea is the generation of a normalised Gomory cut generated from the chosen pivot row \( v \):

\[
s_k = \left[ \frac{a_v}{L} \right] + \sum_{j=m+1}^{n} \left[ \frac{a_{v,j}}{L} \right] \cdot (-x_j)
\]

(4.29)

where

- \( s_k \) is a new basic variable
- \( L \) is a temporarily undetermined positive constant
- \([\text{number}]\) denotes the largest integral value less than or equal to number

Now if we set \( L = a_{v,s} \) where \( a_{v,s} \) is the natural pivot, then eq. (4.29) will have the following two important properties:

\[
\frac{a_v}{v,0} \leq \frac{a_v}{v,s}
\]

(4.30)

which means that the feasibility of the solution is preserved if this row is used as pivot row. If we choose \( L = a_{v,s} \) then the pivot is 1 (if this row is used as pivot row). It is easily verified by inspection of the simplex method that a unit pivot will convert an all integer matrix into another all integer matrix (Gauss elimination).

- The equations are extended by \( n \) trivial equations:

\[
\begin{align*}
    x_{m+1} & = x_m \\
    \vdots & \\
    x_{m+n} & = x_{m+n}
\end{align*}
\]

(4.31)

representing the Gomory variables. Since a new Gomory cut always serves as pivot row, the variable removed from the basis is always a Gomory slack variable. And by pivoting the Gomory cutting plane is added as a new constraint to the set of equalities. Accordingly, no variable of the original system ever leaves the basis.
The algorithm requires that in the initial stage a basic, feasible, integer solution is given.

In the tableau all variables have to be integers, while these remains integers through the pivot operations by setting $L = a_{v,s}$. By each iteration a new Gomory constraint is added to the tableau.

4. 2*N WIRING PROBLEM, DYNAMIC PROGRAMMING APPROACH.

4.1 Introduction.

This chapter describes the solution to solve a general 2*N wiring problem with the dynamic programming approach. This solution is based on the traditional one-net-at-a-time routing. The algorithm used is an extension of [1] which finds the optimal Steiner tree with arbitrary costs on a 2*N grid in linear time. The algorithm is based on dynamic programming and is described in detail because it is developed during my training period.

4.2 Steiner trees.

A minimal tree for a set A of points in the plane is a tree which connect s all points in A using lines with minimal total weight. Usually the weight of a line is defined to be the length of the line and the tree associated with it is the Steiner tree. The problem of constructing minimal Steiner trees has been studied extensively; mostly with the variation on this problem in which all connecting lines are required to run either horizontally or vertically. A rectilinear Steiner tree for a set A of points is a tree composed of horizontal and vertical lines which interconnect all members of A. There typically will be many optimal rectilinear Steiner trees for A with total weight. Various algorithms have been proposed, each of them requires a number of operations which increases exponentially with the number of points to be connected and is impractical for even moderate-sized problems. In fact the problem of constructing an optimal rectilinear Steiner tree has been shown to be NP-complete.

The algorithm presented, is based on the dynamic programming approach which construct an optimal rectilinear tree with arbitrary weights for the special case in which all points of the set A lay on a 2*N grid. The algorithm requires a number of operations which is proportional to the length of the grid.
4.3 Principles of dynamic programming.

Dynamic programming is an optimisation method which is particularly applicable to multistage decision processes. We introduce the concept of a general multistage decision process. Such a process contains three variables, called the stage variable, the state variable, and the decision variable. The process is described by a mathematical equation involving these three variables. The stage variable is scalar-valued and can be defined either on a interval or on a discrete set of points. For simplicity we assume that the stage variable is a real valued variable defined on a discrete set of points. In most cases the values of the stage variable are considered to be discrete instances of time so we shall refer to the stage variable as the time variable in the rest of this paragraph.

The state variable completely describes the decision process at any instance of time. We assume that at every instance of time the values of the state variable are finite-dimensional vectors called state vectors. A decision variable represents the independent decision inputs inherent in a multistage decision process. Its values are also assumed to be finite-dimensional vectors, called decision vectors. Letting and denote the state and the decision vectors at the i-th instance of time, a multistage decision process is represented by an equation of the form

\[ x_{i+1} = g(x_i, u_i) \]  

where \( i = 0, 1, 2, \ldots \) and the initial state vector is given.

The state vector has the following fundamental characteristics:

- Given and \( x^{i+1} \) is unique.
- Contains all relevant information about the decision process at the i-th instance of time. This condition means that for a given and \( x^{i+k} \), \( k = 0, 1, 2, \ldots \) any future state \( x^j \), \( j > i \), can be computed without any knowledge of the past states \( x^p \), \( p < i \).

Suppose the decision-maker chooses at the i-th instance of time. Then at the \( (i+1) \)st instance, the state of the process changes from \( x^i \) to \( x^{i+1} \), according to eq. (5.1). Due to this change of the state, the performance of this process is either improved or degraded according to the same objective function specified by the decision-maker. The multistage decision process consists of finding a sequence of decision vectors over a given period of time that results in the best performance of the process according to the
object function measure used by the decision maker.

We assume that the decision-maker uses an object function, which is a function of \( i \) and \( i \) and is denoted by \( J(\cdot, i) \), to measure the performance of the system over the time period \([i, i+1]\). The cumulative measure of the performance of the process over \( N \) stages is given by

\[
N-1 \\
\sum_{i=0}^{N-1} J(x^i, u^i, i) 
\]  

(5.2)

According to eq. (1), for each \( i > 0 \), is a function of and \( j = 0, \ldots, i - 1 \), and hence \( J(\cdot, i) \) is also a function of and \( j = 0, \ldots, i - 1 \). The cumulative measure of the performance of the process given by (5.2) then becomes a function of \( N \), and \( j = 0, \ldots, N-1 \). Denoting this function by \( J(\cdot, N, \ldots, -1) \), we have:

\[
J(x^N, u^{N-1}, \ldots, u^0) = \sum_{i=0}^{N-1} J(x^i, u^i, i) 
\]  

(5.3)

Frequently the value of \( J(\cdot, i) \) is referred to as the incremental cost of the process at the i-th stage, and the value of \( J(\cdot, N, \ldots, -1) \) is called the total cost of the process over \( N \) stages. The multistage decision process consists of finding an extreme value of \( J(\cdot, N, \ldots, -1) \) with respect to the decision vectors \( \ldots, -1 \). To be specific, we consider the problem of minimising \( J(\cdot, N, \ldots, -1) \) with respect to \( \ldots, -1 \).

The problem of minimisation can be further complicated by the presence of the constraints on the allowable values of and \( . \) For example and may be required by physical considerations to be in some given subsets:

\[
x \in X(i), \quad u \in U(i)
\]  

(5.4)

Method of direct enumeration.

One obvious method for solving the minimisation problem is to evaluate the cost function \( J(\cdot, N, \ldots, -1) \) for all allowable input sequences \( \ldots, -1 \) and obtain the optimal solution by direct inspection. This method is called direct enumeration and is feasible only if \( U(i) \) contains a limited number of elements for every \( i \). Letting \( x^i \in X(i) \) denote an admissible state at the i-th instance of time, then we obtain a finite set of states \(+1\) from by applying
el of \( U(i) \). We then compute the incremental cost of the \( i \)-th stage by \( J(i,i) \) for all such that the \( i+1 \) el of \( X(i+1) \), and add this incremental cost to the previously computed cost arriving at from to obtain the total cost of arriving at \( i+1 \) from . This procedure is repeated until \( N-1 \). After termination of this procedure we have the available values of \( J(N,N,...,1) \) for all allowable choices of \( ...,1 \) and the optimal sequences of decision vectors are obtained by direct inspection of the values of \( J(N,N,...,1) \).

The principal disadvantage of directly enumerating all allowable sequences of decision vectors lies in the huge storage required for reasonably large values of \( N \) and/or dimensionality of \( u \).

The method of dynamic programming was developed by Bellman for sequentially solving minimisation problems which can be formulated as multistage decision process. This method is based on the principle of optimality.

**Principle of optimality:**

An optimal decision sequence has the property that whatever the past states and decisions are, the remaining decisions form an optimal decision sequence with respect to the state from the past decisions.

**Functional equation.**

Let \( I(N-j) \) denote the minimum cost of the \((N-j)\) stage decision process with el of \( U_j \), starting at time instance \( j \) and ending at \( N \), i.e.,

\[
I(x,N-j) = \min_{u \in \text{el of } U} \left\{ \sum_{i=j}^{N-1} J(x,u,i) \right\}
\]

We can rewrite the above expression for \( I(N-j) \) as

\[
I(x,N-j) = \min_{u \in \text{el of } U} \left\{ J(x,u,j) + \sum_{i=j+1}^{N-1} J(x,u,i) \right\}
\]

and by the principle of optimality
\[ I(x, N-j) = \min_{u \in U} \min_{i=j+1} \sum_{j=i}^{N-1} J(x, u, j) + \min_{u \in U} \sum_{j=i}^{N-1} J(x, u, j) \] (5.7)

We note that the first factor \( J(x, u, j) \) is independent of the choice of \( i = j+1, \ldots, N-1 \). Hence we write this equation as:

\[ I(x, N-j) = \min_{u \in U} \left( \min_{i=j+1} \sum_{j=i}^{N-1} J(x, u, j) + \sum_{j=i}^{N-1} J(x, u, j) \right) \] (5.8)

By definition (5.5):

\[ \min_{i=j+1} \sum_{j=i}^{N-1} J(x, u, j) = I(x, N-j) \] (5.9)

and hence

\[ I(x, N-j) = \min_{i=j+1} \left( \min_{u \in U} \sum_{j=i}^{N-1} J(x, u, j) + I(x, N-j-1) \right) \] (5.10)

where \( +1 \) is given by (5.1).

This equation is called the functional equation of dynamic programming and provides a recursive scheme for obtaining \( I(\cdot, N-j) \) for all \( j \) of \( X \) given a knowledge of \( I(\cdot, N) \) by minimising the right-hand side of eq. (5.10). In order to use eq. (5.10) recursively to generate \( I(x, i) \), it is necessary to start at stage \( N-1 \) and compute

\[ I(u, 1) = \min_{u \in U} \left( \sum_{j=1}^{N-1} J(x, u, j) + I(x', N-j-1) \right) \] (5.11)

We can derive this formula also intuitively in the following manner. Suppose is the state of the process at the \( i \)-th instance of time and we want to find the optimal decision vector. For each choice of we arrive at a specific value for \( +1 \) given by eq. (5.1) and each of these states at the \((i+1)\)st instance of time has associated with it at least one optimal path ending in an
admissible value for. Therefore, the total cost starting from should be no greater than \( J(, i) + I(+1, N - i - 1) \), and the optimal \( u \) is obtained by minimising this expression with respect to as shown on the right-hand side of eq. (5.10). Equations (5.10) and (5.11) provide the optimal value for one decision vector. The optimal values of \( +1, k = 1, \ldots, N - j - 1 \), must be computed sequentially before the optimal value of can be determined. This method is clearly different from direct enumeration which provides the optimal values of all decision vectors simultaneously. The amount of computation in the minimisation problem is also reduced, since 

\[
\min \hat{u} 0, \ldots, \hat{u}_{N-1} J(,N,\ldots, -1)
\]

is now replaced by a sequence of minimisation problems each involving one decision vector. Note, that in order to obtain the optimal value of from eq. (5.10) it is necessary to know \( I(+1, N - j - 1) \). Therefore the values of \( I(+1, N - j - 1) \) must be stored for every feasible value of \(+1\). In general, the number of feasible values of \(+1\) increases with increasing number of components of \(+1\), and hence the storage problem becomes critical with large state vectors.

4.4 Graph terminology.

A graph \( G \) is an ordered triple \((V(G), E(G), f)\) consisting of:

- a non-empty set \( V(G) \) of vertices.
- a set \( E(G) \) disjoint from \( V(G) \) of edges.
- an incidence function \( f \) that associates with each edge of \( G \) an unordered pair of necessarily distinct vertices of \( G \).

If \( e \) is an edge and \( u \) and \( v \) are vertices such that \( f(e) = uv \), then \( e \) is said to join \( u \) and \( v \); the vertices \( u \) and \( v \) are called the ends of \( e \). By definition every graph is finite and has no loops or multiple edges. Whenever \( e = [u,v] \) is in \( E \), we say that the vertices \( u \) and \( v \) are adjacent and that each is incident with edge \( e \). A walk of a graph is a sequence, beginning with a vertex and ending with a vertex, in which vertices and edges alternate and each edge is incident with the vertices preceding and following it. The length of a walk is the number of occurrences of edges in it. A path is a walk in which all vertices are distinct. A cycle is a closed walk with more than two vertices all of which are distinct; except the first and the last which coincide. A graph is connected if each pair of its vertices are joined by a path. A tree is a connected graph with no cycles. A subgraph of a graph \( G \) with vertex set \( V \) and edge set \( E \) consists of subsets of \( V \) and \( E \) which together form a graph. In a spanning subgraph of \( G \) all the vertices of \( V \) occur. With
each edge $e$ of $G$ let there be associated a real number $ew(e)$, called the edge weight. With each vertex $v$ of $G$ let there be associated a real number $vw(v)$, called the vertex weight. Then $G$ together with the weights on its edges and vertices is called a weighted graph. A spanning subgraph of $G$ is a subgraph $H$ with $V(H) = V(G)$. A minimum weight spanning subgraph is a spanning tree $T$ of $G$. A minimum weight spanning tree of a weighted graph will be called an optimal spanning tree.

4.5 Problem definition.

![Figure 19. The lattice graph $L$ representing a $2^N$ grid.](image)

We are dealing with the case in which the set $A$ of vertices to be connected is a subset of the grid points of a $2^N$ grid as indicated by the lattice graph $L$ in figure 19. We label the grid vertices on the top-line by $a_1 \ldots a_n$ and on the bottom-line by $b_1 \ldots b_n$ as illustrated. We assume that the set $A$ (vertices "X" in picture) contains at least two grid-points otherwise the problem isn’t a connection problem. The following vertex and edge-weights are introduced:

- Let $ew(e)$ denote the edge weight associated with each edge $e \in E$ with the restriction that this weight is always semi-positive.

$$ew(t) \geq 0 \quad e \in E$$  \hspace{1cm} (5.12)

- Let $vw(v)$ denote the vertex weight associated with each vertex $v \in V$. The vertex costs $vw(v)$ are functions of all incident edges of $V$ and depend therefore on $E$. Each vertex weight $vw(v)$ satisfies the following conditions:

7.101
\[ \forall v \in V \quad \text{vw}(t) \geq 0 \tag{5.13} \]

and

if \( G(V(G), E(G)) \) and \( G'(V(G'), E'(G')) \) with \( E \subseteq E' \)

then

\[ \forall v \in V \quad \text{vw}(v) \leq \text{vw}'(v) \tag{5.14} \]

This means that each vertex weight is always semi-positive and that an additional incident edge of a vertex \( v \) always leads to a non-decreasing value of the vertex weight associated with \( v \).

We now have to construct an optimal spanning tree \( \text{OST}(G,V) \) for the set \( A \) which has the following properties:

- the optimal spanning tree is a subgraph of the lattice graph \( L \).
- the set of vertices of \( A \) is a subgraph of the optimal spanning tree.
- the total weight of the optimal spanning tree formed by the summation of all vertex weights and all edge weights is minimal.

4.6 Transformation of the initial problem in the dynamic programming approach.

The "2*N wiring algorithm" is a linear algorithm based on dynamic programming that finds the minimal weight tree as mentioned in paragraph 5.2 where the weight is simply the sum of edge weights and vertex weights.

The input of the algorithm consists of:

- the set of vertices of \( A \) which have to be interconnected.
- edge weights

- \( \text{VEW} \) is an \( N \) element vector which stores the weights of the vertical edges in the lattice graph; \( \text{VEW}(j) \) indicates the weight that must be added if the tree uses the edge \([a_j, b_j] \).

- \( \text{HAEW} \) and \( \text{HBEW} \) are two \( N \) element vectors which store the weights of the horizontal edges in the lattice graph; \( \text{HAEW}(j) \), \( \text{HBEW}(j) \) indicates the weight that must be added if the tree uses the edge \([a_j, a]\) respectively \([b_j, b]\).
- **vertex weights**

  - AVW is an N element vector which stores the vertex weight functions for the vertices \( a_i \) with \( i = 1, \ldots, N \).
  
  - BW is an N element vector which stores the vertex weight functions for the vertices \( b_i \) with \( i = 1, \ldots, N \).

Let **first** denote the leftmost abscissa of all vertices in set \( A \) :

\[
\text{first} := \{ \min i | a_i \text{ el of } A \text{ or } b_i \text{ el of } A \} \quad (5.15)
\]

Let **last** denote the rightmost abscissa of all vertices in set \( A \) :

\[
\text{last} := \{ \max i | a_i \text{ el of } A \text{ or } b_i \text{ el of } A \} \quad (5.16)
\]

Assume that an optimal Steiner tree, \( \text{OS}T(G, V) \) has been constructed for the set \( A \). Consider now abscissa \( k \) with \( \text{first} < k < \text{last} - 1 \).

Let \( G_A(V(G_A), E(G_A)) \) denote that part of the optimal Steiner tree \( \text{OS}T \) situated left of abscissa \( k \). Graph \( G_A \) consist of the following vertex and edge set :

\[
V(G_A) = \{ a_i \text{ el of } V, b_i \text{ el of } V | i \leq k \}
\]

\[
E(G_A) = \{ [a_i, b_i] \text{ el of } T, [a_i, a_i] \text{ el of } T, [b_i, b_i] \text{ el of } T | i \leq k \}
\]

Let \( G(V(G), E(G)) \) denote that part of the optimal Steiner tree \( \text{OS}T \) situated right of abscissa \( k + 1 \). Graph \( G \) consist of the following vertex and edge set :

\[
V(G) = \{ a_i \text{ el of } V, b_i \text{ el of } V | i > k \}
\]

\[
E(G) = \{ [a_i, b_i] \text{ el of } T, [a_i, a_i] \text{ el of } T, [b_i, b_i] \text{ el of } T | i > k \}
\]

The situation in the \( 2^N \) grid is as follows :

\[
\begin{array}{cccccccccccccccc}
& & & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & & \\
\end{array}
\]
Figure 20. Trees through the vertical cut $k$, $k+1$.

We make the following important observation: at abscissa $k$ one of the following assertions holds:

1. Edge $[a_k, a]$ is an element of $T$ and edge $[b_k, b]$ is not an element of $T$ which implies that $G_A$ and $G$ are trees, otherwise $OST$ is not a tree.

2. Edge $[a_k, a]$ is not an element of $T$ and edge $[b_k, b]$ is an element of $T$ which implies that $G_A$ and $G$ are trees, otherwise $OST$ is not a tree.

3. Edge $[a_k, a]$ is an element of $T$ and edge $[b_k, b]$ is an element of $T$, $G_A$ is a tree and $G$ is a forest consisting of two different trees $T^*$ and $T^{**}$ where $a$ is an element of $V(T^*)$ and $b$ is an element of $V(T^{**})$.

4. Edge $[a_k, a]$ is an element of $T$ and edge $[b_k, b]$ is an element of $T$, $G_A$ is a forest consisting of two different trees $T^*$ and $T^{**}$ where $a_k$ is an element of $V(T^*)$ and $b_k$ is an element of $V(T^{**})$ and $G$ is a tree.
We define the *stage variable* to be the x-abscissa in the lattice graph L. The *state variables* must satisfy with the above mentioned assertions. We propose an extension of the stages from left to right (with increasing abscissa number) which means extension of $G_\lambda$. The total weight of $G_\lambda$ at abscissa $k$ is simply the sum of all edge weights and all vertex weights with exception of the possible edge weights for the edges $[a_k,a]$ and $[b_k,b]$ by which $G_\lambda$ is further extended.

By the *principle of optimality* every tree at abscissa $k$ have to be minimal weighted.

Note that at abscissa $k$ we have to know the further extensions $[a_k,a]$ and $[b_k,b]$ to calculate the vertex weights of vertices $a_k$ and $b_k$.

\[
\begin{align*}
\text{set } A & \quad \text{first} \quad \text{last } N \\
\begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & \begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & 1 \\
1 & \text{first} & \text{last } N
\end{align*}
\]

\[
\begin{align*}
\begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & \begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & 1 \\
1 & \text{first} & \text{last } N
\end{align*}
\]

\[
\begin{align*}
\begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & \begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & 2 \\
1 & \text{first} & \text{last } N
\end{align*}
\]

\[
\begin{align*}
\begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & \begin{array}{ccc}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
\end{array} & 3 \\
1 & \text{first} & \text{last } N
\end{align*}
\]

7.105
Definition of the state variables:

For inspection of the above mentioned assertions we need the following definitions:

1. Let $T_1(k)$ denote the minimal weight tree which interconnects the following set of vertices:
   \[
   \{ a_i, b_i \mid a_i \text{ elem of } A, b_i \text{ elem of } A \text{ with } i \leq k \}
   \cup a_k
   \]
   with future edge extension $[a_k, a]$

2. Let $T_2(k)$ denote the minimal weight tree which interconnects the following set of vertices:
   \[
   \{ a_i, b_i \mid a_i \text{ elem of } A, b_i \text{ elem of } A \text{ with } i \leq k \}
   \cup b_k
   \]
   with future edge extension $[b_k, b]$

3. Let $T_3(k)$ denote the minimal weight tree which interconnects the following set of vertices:
   \[
   \{ a_i, b_i \mid a_i \text{ elem of } A, b_i \text{ elem of } A \text{ with } i \leq k \}
   \cup \{ a_k, b_k \}
   \]
   with future edge extensions $[a_k, a]$ and $[b_k, b]$

4. Let $T(k)$ denote the minimal weight forest consisting of two different trees $T^*$ and $T^{**}$ where $a_k$ elem of $T^*$ and $b_k$ elem of $T^{**}$ which interconnects the following set of vertices:

Figure 21. Dynamic wiring within a 2*N grid.
( \( a_i, b_i \mid a_i \text{ elem } A, b_i \text{ elem } A \text{ with } i \leq k \) )

\( U \{a_k, b_k\} \)

with future edge extensions \([a_k, a]\) and
\([b_k, b]\) (This means that the trees have to be joined later).

According to eq. (4.10) we need for the construction of the trees \( T^i(k+1) \),
\( i = 1,2,3,4 \), all old trees \( T^i(k) \) and evaluate all possible extensions. We
simply enumerate all possibilities:

. : future edge extension.
* : edge already used.

\[
\begin{array}{ccccccc}
  \text{k} & \text{k+1} & \text{k+2} & \text{k} & \text{k+1} & \text{k+2} & \text{k} \\
  a & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
  b & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
 1 : T(k) \rightarrow T(k+1) & 1 : T(k) \rightarrow T(k+1) & 2 : T(k) \rightarrow T(k+1) & 3 : T(k) \rightarrow T(k+1) & 4 : T(k) \rightarrow T(k+1) & 5 : T(k) \rightarrow T(k+1) & 6 : T(k) \rightarrow T(k+1) \\
 7 : T(k) \rightarrow T(k+1) & 8 : T(k) \rightarrow T(k+1) & 9 : T(k) \rightarrow T(k+1) & 10 : T(k) \rightarrow T(k+1) \\
\end{array}
\]
11: T(k) → T(k+1)

12: T(k) → T(k+1)

13: T(k) → T(k+1)

14: T(k) → T(k+1)

15: T(k) → T(k+1)

16: T(k) → T(k+1)

17: T(k) → T(k+1)

18: none → T(first)

19: none → T(first)

20: T(left) → T(first)

21: none → T(first)

22: none → T(first)

23: T(left) → T(first)

24: none → T(first)

7.108
Construction possibilities $T^3(k + 1)$:

<table>
<thead>
<tr>
<th>pos</th>
<th>previous tree</th>
<th>edge extensions</th>
<th>vertex extensions</th>
<th>increase weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$T^1(k)$</td>
<td>$[a_k, a] + \ a$</td>
<td></td>
<td>$ew([a_k, a]) + vw(a)$</td>
</tr>
<tr>
<td>2</td>
<td>$T^1(k)$</td>
<td>$[a_k, a] + [a, b] + \ a$</td>
<td>$b$</td>
<td>$ew([a_k, a]) + ew([a, b]) + vw(a) + vw(b)$</td>
</tr>
<tr>
<td>3</td>
<td>$T^2(k)$</td>
<td>$[b_k, b] + [a, b] + \ a$</td>
<td>$b$</td>
<td>$ew([b_k, b]) + ew([a, b]) + vw(a) + vw(b)$</td>
</tr>
</tbody>
</table>
To construct $T^1(k + 1)$ we enumerate all admissible extensions from $T^i(k)$, $i = 1, \ldots, 4$, compute the total weights and select the admissible possibility with minimal total weight for $T^1(k + 1)$.

**Remark:**
Dependent on the set $A$ some extension possibilities are not allowed. If $b$ is member of set $A$ then possibility 1 is out of consideration because this case may form a cheapest tree which does not satisfy the definition of the OST (not all members of set $A$ are in the OST).

**Construction possibilities $T^2(k + 1)$:**

<table>
<thead>
<tr>
<th>pos</th>
<th>previous tree</th>
<th>edge extensions</th>
<th>vertex extensions</th>
<th>increase weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$T^1(k)$</td>
<td>$[a_k , a]$</td>
<td>$[a,b]$</td>
<td>$\text{ew}([a_k , a])$ + $\text{ew}([a,b])$ + $\text{vw}(a)$ + $\text{vw}(b)$</td>
</tr>
<tr>
<td>7</td>
<td>$T^2(k)$</td>
<td>$[b_k , b]$</td>
<td></td>
<td>$\text{ew}([b_k , b])$ + $\text{vw}(b)$</td>
</tr>
<tr>
<td>8</td>
<td>$T^2(k)$</td>
<td>$[b_k , b]$</td>
<td>$[a,b]$</td>
<td>$\text{ew}([b_k , b])$ + $\text{ew}([a,b])$ + $\text{vw}(a)$ + $\text{vw}(b)$</td>
</tr>
</tbody>
</table>
Remark:
Dependent on the set A some extension possibilities are not allowed. If a is
member of set A then possibility 7 is out of consideration because this case
may form a cheapest tree which does not satisfy the definition of the OST (not
all members of set A are in the OST).

For $T^{2}(k + 1)$ we select the admissible possibility with minimal total weight.

Construction possibilities $T^{3}(k + 1)$:

<table>
<thead>
<tr>
<th>pos</th>
<th>previous tree</th>
<th>previous edge extensions</th>
<th>previous vertex extensions</th>
<th>increase weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>$T^{1}(k)$</td>
<td>$[a_{k}, a]$</td>
<td>$[a, b]$</td>
<td>$ew([a_{k}, a])$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$ew([a, b])$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$vw(a)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$vw(b)$</td>
</tr>
<tr>
<td>12</td>
<td>$T^{2}(k)$</td>
<td>$[b_{k}, b]$</td>
<td>$[a, b]$</td>
<td>$ew([b_{k}, b])$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$ew([a, b])$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$vw(a)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$vw(b)$</td>
</tr>
<tr>
<td>13</td>
<td>$T^{3}(k)$</td>
<td>$[a_{k}, a]$</td>
<td>$[b_{k}, b]$</td>
<td>$ew([a_{k}, a])$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$ew([b_{k}, b])$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$vw(a)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$vw(b)$</td>
</tr>
</tbody>
</table>
We select the possibility with minimal total weight for $T^3(k + 1)$.

Construction possibilities $T(k + 1)$:

<table>
<thead>
<tr>
<th>pos</th>
<th>tree possibilities</th>
<th>increase weights</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>$T^3(1)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[a_k, a] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[b_k, b] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[a, b] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$vw(a)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$vw(b)$</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>$T^3(2)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[b_k, b] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[a_k, a] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[a, b] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$vw(a)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$vw(b)$</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>$T^3(k)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[a_k, a] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[b_k, b] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$[a, b] +$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$vw(a)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$vw(b)$</td>
<td></td>
</tr>
</tbody>
</table>

We select the possibility with minimal total weight for $T(k + 1)$.

Construction of the initial trees $T_i^4(\text{first}), i=1,2,3,4$:

We are now dealing with abscissa first and must construct the initial trees to start the dynamic program. We must now emphasise the fact that a left detour (a tree which uses edges left of abscissa first) might result in a cheaper tree than a direct edge $[a_\text{first}, b_\text{first}]$. Considering this fact we construct an auxiliary detour tree of minimal total weight situated in the left part of the grid left to abscissa first and future edge extensions $[a_\text{first-1,} a_\text{first}]$ and $[b_\text{first-1,} b_\text{first}]$. $T$ left detour is the tree with:

7.112
first - 2
\[ \text{ew}([a,b]) + \sum_{i=s}^{\text{first}-2} \{ \text{ew}([a_{i+1},a]) + \text{ew}([b_{i},b]) \} + \]
\[ \text{first} - 1 \]
\[ + \sum_{i=s}^{\text{first}-1} \{ \text{vw}(a_i) + \text{vw}(b_i) \} \]
is minimal
\[ i = s \]
with
\[ 1 \leq s \leq \text{first} - 1 \]

---

**Figure 23.** An example of T left detour.

Denote k is abscissa first.

Construction possibilities of initial tree \( T_1(\text{first}) \):

<table>
<thead>
<tr>
<th>pos</th>
<th>previous tree</th>
<th>edge extensions</th>
<th>vertex extensions</th>
<th>increase weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>none</td>
<td>a</td>
<td></td>
<td>( \text{vw}(a) )</td>
</tr>
<tr>
<td>19</td>
<td>none</td>
<td>([a,b] +)</td>
<td>a +</td>
<td>( \text{vw}(a) ) +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b</td>
<td></td>
<td>( \text{vw}(b) )</td>
</tr>
<tr>
<td>20</td>
<td>( T \text{left} )</td>
<td>([a_{k+1},a] +)</td>
<td>( a ) +</td>
<td>( \text{vw}(a) ) +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([b_{k+1},b] +)</td>
<td>( b ) +</td>
<td>( \text{vw}(b) )</td>
</tr>
</tbody>
</table>

Remarks:

Dependent on the set A and the existence of \( T \) left detour some extension possibilities are not allowed:

- \( b_{\text{first}} \) not elem of set A then possibility 18 is minimal weighted.
- \( b_{\text{first}} \) elem of set A and \( T \) left detour does not exist then possibility 19 is the only allowable.
- If $a_{\text{first}}$ is not an element of set $A$ then possibility 21 is minimal weighted.

- If $a_{\text{first}}$ is an element of set $A$ and $T$ left detour does not exist then possibility 22 is the only allowable.

- If $a_{\text{first}}$ is an element of set $A$ and $T$ left detour exist then select out of the possibilities 22 and 23 the one with the minimal total weight.

Construction possibilities of initial tree $T^2(\text{first})$ :

<table>
<thead>
<tr>
<th>pos</th>
<th>previous edge extensions</th>
<th>increase weights</th>
<th>tree vertex extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>none</td>
<td>$b$</td>
<td>$vw(b)$</td>
</tr>
<tr>
<td>22</td>
<td>none</td>
<td>$[a,b] + a + b$</td>
<td>$vw(a) + vw(b)$</td>
</tr>
<tr>
<td>23</td>
<td>$T$ left</td>
<td>$[a_k,b] + a$</td>
<td>$vw(a) + vw(b)$</td>
</tr>
</tbody>
</table>

Remarks:
Dependent on the set $A$ and the existence of $T$ left detour some extension possibilities are not allowed.

- If $a_{\text{first}}$ is not an element of set $A$ then possibility 21 is minimal weighted.

- If $a_{\text{first}}$ is an element of set $A$ and $T$ left detour does not exist then possibility 22 is the only allowable.

- If $a_{\text{first}}$ is an element of set $A$ and $T$ left detour exist then select out of the possibilities 22 and 23 the one with the minimal total weight.

Construction possibilities of initial tree $T^3(\text{first})$ :

<table>
<thead>
<tr>
<th>pos</th>
<th>previous edge extensions</th>
<th>increase weights</th>
<th>tree vertex extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>none</td>
<td>$[a,b] + a + b$</td>
<td>$vw(a) + vw(b)$</td>
</tr>
</tbody>
</table>
Remarks:
Dependent T left detour some extension possibility 25 is not allowed.

Construction possibilities of initial tree T(first):

<table>
<thead>
<tr>
<th>pos</th>
<th>previous</th>
<th>edge extensions</th>
<th>vertex extensions</th>
<th>increase weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>none</td>
<td>a +</td>
<td>b +</td>
<td>vw(a) +</td>
</tr>
</tbody>
</table>

We select for all initial trees $T^i$(first), $i=1,...,4$ the one with minimal weight.

Remark:
This is always the possibility with minimal weight.

Construction of the trees $T^i$(last), $i=1,2,3,4$:
We are now dealing with abscissa last and have to construct the tree $T$ last without future extensions to end the dynamic program.

Figure 24. An example of T right detour.

We must now emphasise the fact that a right detour (a tree which uses edges right of abscissa last) might result in a cheaper tree than a direct edge $[a_{\text{last}},b_{\text{last}}]$. This means that if a right detour is permitted, we must extend T further to the right.

First we consider the possibilities to generate a tree $T_{\text{last}}$ with no further extensions than abscissa $i_{\text{last}}$ out of the trees $T^i$(last - 1).

Denote k is abscissa last - 1.
Construction possibilities of tree $T_{\text{last}}$:

<table>
<thead>
<tr>
<th>pos</th>
<th>previous tree</th>
<th>edge extensions</th>
<th>vertex extensions</th>
<th>increase weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>$T^1(k)$</td>
<td>$[a_k, a]$</td>
<td></td>
<td>$ew([a_k, a]) + \text{vw}(a)$</td>
</tr>
<tr>
<td>28</td>
<td>$T^1(k)$</td>
<td>$[a_k, a] + [a,b]$</td>
<td></td>
<td>$ew([a_k, a]) + \text{vw}(a)$</td>
</tr>
<tr>
<td>29</td>
<td>$T^2(k)$</td>
<td>$[b_k, b]$</td>
<td></td>
<td>$ew([b_k, b]) + \text{vw}(b)$</td>
</tr>
<tr>
<td>30</td>
<td>$T^2(k)$</td>
<td>$[b_k, b] + [a,b]$</td>
<td></td>
<td>$ew([b_k, b]) + \text{vw}(a)$</td>
</tr>
<tr>
<td>31</td>
<td>$T^3(k)$</td>
<td>$[a_k, a]$</td>
<td></td>
<td>$ew([a_k, a]) + \text{vw}(a)$</td>
</tr>
<tr>
<td>32</td>
<td>$T(k)$</td>
<td>$[a_k, a] + [b_k, b] + [a,b]$</td>
<td></td>
<td>$ew([a_k, a]) + \text{vw}(a)$</td>
</tr>
</tbody>
</table>

The possibility with minimal weight and which is admissible is selected.

Remarks:

- If $a_{\text{last}}$ elem of set $A$ then possibility 29 is not admissible.
- If $a_{\text{last}}$ not elem of set $A$ then possibility 29 has less total weight than possibility 30.
- If $b_{\text{last}}$ elem of set $A$ then possibility 27 is not admissible.
- If $b_{\text{last}}$ not elem of set $A$ then possibility 27 has less total weight than possibility 28.
The minimal weight tree of all this extension possibilities is stored in tree T_{last}.

If a right detour is possible then we construct the auxiliary right detour tree which covers the vertices a_i and b_i with:

\[
\begin{align*}
    s & \quad \text{ew}([a, b]) + \sum_{i = 1}^{s} \left( \text{ew}([a, a_i]) + \text{ew}([b, b_i]) \right) + \text{ew}([a, a_{i+1}]) + \text{ew}([b, b_{i+1}]) \\
    & \quad + \sum_{i = s}^{s+1} \left( \text{vw}(a) + \text{vw}(b) \right) \text{ is minimal}
\end{align*}
\]
with

\[s + 1 \leq s \leq N\] (where N is the length of the grid)

We construct now the tree T_{last} and join this with the auxiliary tree resulting in tree T_{right detour}. The optimal Steiner tree is now the cheapest tree of T_{last} and T_{right detour}.

This results in the following algorithm to derive a 2*N wire solution:

algorithm:

proc OST (Optimal Steiner Tree)
    determine set A, first, last
    abscissa := first
    construct initial trees T_i(first), i=1,2,3,4
    while abscissa < last
    do
    for all i
    do
    construct T_i(abscissa + 1) by extension of the previous trees T_i(abscissa)
    select the cheapest possibility
    od
    abscissa := abscissa + 1
    od
    construct T_{last}
    construct cheapest T_{right detour}
    OST := cheapest (T_{last}, T_{right detour})
corp

7.117
Remarks:

- In case one net results in several 2*N wiring subproblems it is necessary to limit the detour areas of those subproblems in such a way that the wire solutions never can have overlapping areas because this would result in superfluous interconnections; the subproblems are already interconnected somewhere at the outside of the 2*N grid.

Lower row is split at the current hierarchical level.

(a) route at the previous hierarchical level.

```
............X******.............
............X******.............
```

(b) wire solution of subproblem 1, has right detour.

```
............X**********.............
............X**********.............
```

c) wire solution of subproblem 2.

```
............X*****X**********.............
............X*****X**********.............
```

c) total result with superfluous interconnection.

Figure 25. Example of no area limitation with a superfluous interconnection.

Suppose a net on the previous level of hierarchy has taken the route depicted in fig. 25(a) and generates two new 2*N subproblems on the current level of hierarchy. No limitations of the detour area might result in the solutions 25(b), 25(c) which if combined result in a superfluous interconnection at abscissa k, see 25(d). This solution is avoided by limiting the detour area of those subproblems. In the current program the detour area between two subproblems is split in half.
Note that at each stage of recursion we make a constant number of comparisons and the total computation time is \(O(N)\) where \(N\) is the length of the grid. In the initial stage computing \(T^i(\text{first}), \ i=1,\ldots,4\) takes \(O(\text{first})\) time, construction of \(T^{\text{last}}\) is performed in time \(O(\text{last} - \text{first})\) and a final right detour is computed in at most \(O(N - \text{last})\).

Usually the interval \([\text{first}, \text{last}]\) is small in comparison with \([1, N]\), so we start the construction of the left detour and the right detour at respectively abscissa \(\text{first} - 1\) extending to the left and at abscissa \(\text{last}\) extending to the right to eliminate unnecessary computations. For example the extension to the left of the left detour is stopped if the tree consisting of

\[
\begin{align*}
\text{first-2} & \quad \sum_{i=j}^{\text{first}-1} (\text{ew}([a, a]_{j, j+1}) + \text{ew}([b, b]_{j, j+1})) + \\
\text{first-1} & \quad + \sum_{i=j}^{\text{first}-1} (\text{vw}(a_i) + \text{vw}(b_i)) \\
\text{with} & \quad 1 \leq j \leq \text{first} - 3
\end{align*}
\]

has higher weight than a previous determined minimal weight left detour because addition of further edges and vertices can only increase the total weight. In most cases the time spent for the \(2\times N\) dynamic routing will be \(O(\text{last} - \text{first})\).

4.7 Net specific interconnections.

A net specific interconnection is introduced when the instance of a macro uses a piece of interconnect (usual in metal) which appears in several global wiring grid cells. In the global wiring model accessible elements of a macro for a net now are subdivided into two groups:

- **Single points** in a global grid cell. The pin or piece of interconnect of the macro resides completely within the global grid cell area.

- **Polygons** consisting of several global grid cells. The pieces of fixed interconnect form a route crossing several global grid cell boundaries. These fixed interconnections may not be altered by the router.

Remarks:
- The polygons are electrical conducting material so these polygons are only reserved for the specific net.

- Our net description consists of a set of elements form one or both groups which should be interconnected.

- The routers tasks case of a polygons element is to make at least one interconnection with the polygon. We do not specify a particular grid cell of the polygon to make connection.

4.7.1 Modification of the dynamic wiring procedure.

After placement the macros on the chip, channel and via capacities are determined. Of each net the elements to be connected are collected. The router must determine global wire routes for each net in such a way that:

- channel and via capacity restrictions are obeyed.

- all elements of a net are connected with each other. The router may introduce several branches on a polygon to make connections with the other elements.

If the router uses a net specific interconnection to cross from one cell to the other the channel capacity of the corresponding boundary is not updated because the local router can use the fixed interconnection and have not to use a free track for properly embedding the wire at final track resolution.

When a supercell has at lower hierarchical level a fixed interconnection crossing the boundary at the current hierarchical level a fixed interconnection between this cell and the adjacent supercell is assumed (assuming that at the lowest level of hierarchy the fixed interconnection is used to cross the boundary).

We assume that all cells of a polygon may be used by the router to make a interconnection.

Refinement of the dynamic wiring algorithm:

In the 2*N lattice graph the set A no longer consists of vertices but also of edges representing the fixed interconnections.
X : vertices of set A
* : edges corresponding with fixed interconnections.

Figure 26. Example of new set A.

Modifications :

- All vertices in the lattice graph which are member of a polygon serve as terminal cells.
- The weight of edges which correspond with fixed interconnections are made zero to force the edges in the optimal solution (all weights of the other edges are greater than zero!).
- When backtracking of the optimal solution is performed only the channel capacities of edges of the optimal tree which do not correspond with already fixed interconnections are updated.

Additional constraints :

The presence of fixed interconnections introduce additional constraints in the dynamic wiring algorithm :

- Absence of a tree type at some stage in the dynamic programming.

Tree type 1 :

$T^k$ is defined with future edge extension $[a_k, a]$. If edge $[b_k, b]$ corresponds with a fixed interconnection this tree does not exist at column $k$ because edge $[b_k, b]$ have to be element of the minimal weight tree by given fixed connect constraint.

Tree type 2 :

This tree does not exist case edge $[a_k, a]$ corresponds with a fixed interconnection.

Tree type 4 :

This tree does not exist case edge $[a_k, b_k]$ corresponds with a fixed interconnection (this tree is no longer a forest as required by definition)
Forced extension which can create loops in the minimal weight tree. The minimal weight tree is then no longer a tree.

Case the edges \([a_k, a], [b_k, b], [a_k, b_k]\) all corresponds with fixed interconnections all these edges should appear in our final solution. The tree element extension at that stage will utilise all these edges (extension of the previous trees \(T_1^3(k-1)\) and possibly \(T_1^4(k-1)\)) and thus possibly creating a loop case \(T_1^4(k-1)\) does not exist.

Example:

```
+-----X*****XxxxxxXxxxxxxX-----X-----+
+-----X*****XxxxxxXxxxxxxX-----X-----+
```

\(k\) \(k+1\)

*: interconnect made by the router

\(x\): fixed interconnect

Figure 27. Solution with forced loop.

This leads towards the modified dynamic algorithm:

**Algorithm:**

```
proc OST (Optimal Steiner Tree)
    determine set A, fixed interconnection, first, last
    abscissa := first
    construct all feasible initial trees
    while abscissa < last
        do
            determine feasible trees at current abscissa
            for all feasible trees
                do
                    construct feasible \(T_1^i(abscissa + 1)\) by extension of
                    the previous feasible trees \(T_1^i(abscissa)\)
                    select the cheapest possibility
                od
            abscissa := abscissa + 1
        od
    construct T_last
    construct cheapest T_right detour
    OST := cheapest (T_last, T_right detour)
corp
```

7.122
4.8 Cost functions.

The weight of the edges which correspond with the horizontal and vertical boundaries are chosen to be exponentially decreasing function of the remaining boundary capacities. If B is the boundary, C(B) the initial boundary capacity and N(B) the number of already routed nets crossing the boundary B, then:

$$H_{AEW,HB EW}(B) = C_1^{(N(B)-C(B))} + C_2$$

Where C2 is an additional constant to incorporate the wire length of a net.

The via costs are chosen similarly. If M(G) is the maximal number of vias that can be realised in cell G and T(G) is the number of vias already in this cell then:

$$AV_{W,BVW}(G) = C_3^{(T(G)-M(G))} + C_4$$

The constants C1, C2, C3, C4 are determined during a fine tuning process.

Remarks:

- if the aspect ratio of the grid cells is large then the coefficients for the horizontal weights are chosen different from those for the vertical weights.
- C(B), M(G) are the 2*N boundary and via capacities of chapter 1.

4.9 Rerouting.

Once all nets are embedded in the 2*N grid, the total solution may result in some overcongested regions in the sense that the number of boundary crossings exceeds its initial capacity. In order to reduce the net order dependency and the improvement of the distribution of the remaining boundary capacities rerouting is performed. The rerouting procedure selects a net, removes it, updates the boundary and via capacities, computes new boundary and via costs (against the background of all other wires) and executes the dynamic wiring procedure again.

The net selection can be performed

- randomly in order to reduce the net order dependence.
- based on wire length, long nets are selected first to speed up the convergence.
based on congestion. Each net crossing a boundary is assigned a weight which is a function of the congestion in the whole path of the net. The net with the highest weight is first selected for rerouting to move the net segments away from the congested peaks.

Experiments so far indicated that:

- Rerouting based on congestion weights delivers the best result. Most nets choose exactly the route they had before, but several of them choose other routes improving the distribution of the remaining boundary capacities.

- Repeated rerouting is not practical because in the third and subsequent stages the nets choose the same routes as before. The small thickness of the grid (2 strips) is probably the reason why the rerouting process converges so fast.

5. Grid definition.

This chapter evaluates several global grid cell definitions of the gate array. By the architecture of the image gate arrays are divided into two major groups: island cell style and poly cell style. For global grid cell definitions the following facts should be considered:

- Cells should be big enough to ignore local design rule constraints. In our model we have only capacity and via restrictions.

- Channel capacities and via capacities should be determined in such a way that when the capacity restrictions are obeyed the local router can complete his job (embedding all wires at the final tracks).

- Cell definition should be uniform as possible. The heuristics used by the global router gives better result with uniform examples.

5.1 Island cell style.

Usually appearing in bipolar technology where the pins are defined by the basis and emitter contacts of the single transistors. The functional decomposition delivers a net list consisting of unique defined pins, no fixed underpaths or feed throughs occur.

A island consists of several functional cells which are placed "back to back" to each other. A suitable model for the final grid is to define each island to be two final grid cells, with a common boundary coincident with the line define by the back of the all functional cells on an island. The extension to a grid
**Figure 28. Island cell with definition of the grid.**

is performed by division of the routing area on the chip in cells defined by extrapolating the previously defined final grid cell boundaries of the islands. The bonding pads can be modelled by definition of extra cells around the formed grid and introducing extra pins for each used bonding pad within this cell.

5.2 Poly cell style.

The poly cell style is characterised by functional cell rows located between routing channels.

**Figure 29. Poly cell style with definition of the grid.**

The most obvious feature are couples of two opposite electrically common pins facing both succeeding routing channels. These pins are already interconnected by fixed underpaths forming a gate in the gate array. One should distinguish the following two cases:
- a net using a gate for electrical performance must have an interconnection with at least one of the pins of the actual gate.

- feed through (gates having no functional performance) may be used by all nets to make an interconnection between two succeeding wiring channels.

This leads to the following definitions:

**Interconnection terminal couple** are two opposite pins in a functional cell perform an electrical function and have to be interconnected with other functional cells (belong to the net list).

**Feed through** two opposite pins in a functional cell performing no electrical activity. These pins can be used to make a jump between two succeeding wiring channels.

Grid lines parallel to the wiring channels etc. are defined in the middle of the functional cells so each functional cell is divided into two parts with the pins of feed throughs and interconnection terminal couples at both sides of the grid line.

An interconnection terminal couple is modelled by a fixed interconnection and a feed throughs by the capacity definition of the cell boundary. By this definition we get separate treatments of feed throughs and interconnection terminals couples (which are net specific). We define the final grid cells as follows: A suitable parameter for the boundary capacity on the grid line is the number of feed throughs crossing this line. One must emphasise the fact that a crossing of the grid line by an interconnection terminal couple does not affect the capacity (number of feed throughs remains the same). This leads to a separate treatment of crossings, one should know if a interconnection terminal couple or a real feed through is used.

Assume that we are dealing at this hierarchical level where the pins are located in two adjacent supercells. In our model this means that the terminals are represented by two adjacent vertices in the lattice graph. The edge between these vertices denotes a possible interconnection but in fact this interconnection is already made by the fixed underpath. So in our model the edge weight of this edge is set to zero (lowest weight possible, so forcing the edge to be always member of the optimal Steiner tree) and no update of the corresponding boundary capacity.

**Interconnection terminals at high level of hierarchy.** At a high level of hierarchy we need to know the exact place of a interconnection terminal in the supercell. We must know the possible underpaths to neighbouring supercells by
crossing the supercell boundaries. In this case we can treat the crossing different. This approach is also suitable for making interconnection with predefined wire segments crossing various final grid cells. We model this wire by a sequence of interconnection terminals having fixed underpaths which each other. A net to be interconnected may access this functional cell by making contact at at least one of those pins. Definition of a functional cell to be a final grid cell becomes now very complicated.


The current program can be regarded as a contribution towards a gate array design package. However it is quite difficult to evaluate the current version of the program by presence of one realistic and two artificial gate array examples. Also the placement algorithm and the local routing algorithm are not completely finished yet.

As mentioned in chapter 5 the program is adapted to handle net-specific or fixed interconnections usually apparent in gate arrays with poly cell style. The only realistic example available was a poly style gate array. The results of the global router using this example with the given placement where encouraging: no overflows at all and most nets near minimal total length (probably because only about 70% of the gate array was occupied).

The routing results achieved with the dynamic programming approach are much better than those achieved with the "divide and conquer" approach using linear programming method. The reasons for this are:

- By the divide and conquer approach there are more heuristics involved than in the dynamic programming approach.

- The divide and conquer approach is more sensitive for non uniform distribution of the initial channel and via capacities in the 2*N grid.

- No overflow handling in the linear programming. When overflows occur the wires of some nets are removed out of the database. In the dynamic programming approach overflowed nets remain resident, at lower hierarchical levels these overflows may disappear.

It is extremely difficult to adapt the linear approach for the net specific interconnections because this result in an enormous set of equations resulting in a big tableau.
7. REFERENCES.

RECTILINEAR STEINER TREES: EFFICIENT SPECIAL CASE ALGORITHMS.

MIN-CUT PLACEMENT.

HIERARCHICAL CHANNEL ROUTER.

HIERARCHICAL WIRING OF GATE-ARRAY VLSI CHIPS.

HIERARCHICAL WIRE ROUTING.

HIERARCHICAL VLSI LAYOUT:
SIMULTANEOUS PLACEMENT AND WIRING OF GATE ARRAYS.
Int. Conf. on VLSI, 1983, p. 45-60.

[7] Deutsch, D. N.
A ‘DOGLEG’ CHANNEL ROUTER.

[8] Dunlop, A. E.
AUTOMATIC LAYOUT OF GATE ARRAYS.

LAMBDA, AN INTEGRATED MASTER-SLICE LSI CAD SYSTEM.
[10] Harary, F.
GRAPH THEORY.
Reading, Massachusetts : Addison-Wesley, 1969.

A MATURE I2L/STL GATE ARRAY LAYOUT SYSTEM.

A GENERALIZED CHANNEL ROUTER.

INTEGER PROGRAMMING AND NETWORK FLOWS.
Reading, Massachusetts : Addison-Wesley, 1969.

[14] Hwang, F. K.
THE RECTILINEAR STEINER PROBLEM.

A GATE ARRAY DESIGN SYSTEM ADAPTIVE TO MANY TECHNOLOGIES.

[16] Lee, D. T., S. J. Hong and C. K. Wong
NUMBER OF VIAS : A CONTROL PARAMETER FOR GLOBAL WIRING
OF HIGH-DENSITY CHIPS.

[17] Lee, C. Y.
AN ALGORITHM FOR PATH CONNECTIONS AND ITS
APPLICATIONS.

[18] Nan, N. and M. Feuer
A METHOD FOR THE AUTOMATIC WIRING OF LSI CHIPS.

7.129
A GREEDY CHANNEL ROUTER.

YACR : YET ANOTHER CHANNEL ROUTER.

A CONSIDERATION OF THE HORIZONTAL GRIDS USED IN THE
ROUTING OF A MASTER SLICE LAYOUT.

[22] Wismer, D. A. and R. Chattergy
INTRODUCTION TO NONLINEAR OPTIMIZATION.

[23] Yoshimura, T. and E. S. Kuh
EFFICIENT ALGORITHMS FOR CHANNEL ROUTING.
Local routing.

A.G.J. Slenten.

Eindhoven University of Technology.
1. Introduction.

Some aspects of local routing are discussed in this paragraph. In the first section, 2, problems that are due to routing in small sub-areas are mentioned. A definition of the problem to be solved by the routing algorithm is given in section 3. The routing landscape, the space in which nets are connected is described in section 4. In the last section, 5, the choice of the algorithm used to make the interconnections is discussed.

2. Routing in sub-areas.

As the name says, a local router tries to connect the pins of a net in a sub-area of the chip. These sub-areas are defined at the global router stage and are called cells or boxes. The definition of these boxes is somewhat arbitrary.

The problem is to define a box that is useful for both global and local routing. For global routing, a criterion for the choice of a box is that meaningful capacities can be defined at the edges of the boxes. Capacity in this context is a measure of how many wires can cross a specific boundary or how many wires can be routed within the box region. An overflow exists if more nets are crossing a boundary then the capacity for that boundary indicates.

The result of the global routing phase is a collection of boxes for the local router. A set of nets is specified in each box, and information is given for each net indicating if the net has to be routed within the box or if it has to cross one or more boundaries. Essential for the local router is that the results of the global router are reliable. This means that if, in the ideal situation, the global router indicates that a set of nets can be routed in a specific box, without any capacity overflow, that this solution does exist.

Unfortunately, practical results show that it is doubtful if this statement can be proved. Of course, a very pessimistic capacity choice can be made, but it implies that the global router will create many capacity overflows in practice and the result of the global router becomes useless.

So, with a realistic choice of capacities, it is doubtful if a given set of nets can be routed in the box, even if no capacity overflows exists. Until now, research done to come to a technology independent choice of boxes has been unsuccessfully.

* This paper has been produced in the context of TASK IX of the project MR-09 supported by the Commission of the EEC, under the 3744/81 Microelectronics program. Partial support by the Ministry of Economic Affairs of the Netherlands under the NELSIS program is hereby also acknowledged.
Given the situation that, even with no capacity overflow, some nets may not be routed, the question how to route these nets becomes more important. It has to be pointed out that if some nets are not routed at this (local routing) stage, the result can be regarded as useless. This is because the connection problems left will be so complex, that even human interaction is no solution. Further, the system aims to be automatic, so human interaction is not preferred. This implies that some strategy has to be developed to solve the interconnection problems with nets that can not be routed within the initial box. Two strategies are presented here.

Assume the situation that some nets have to be routed from one box to another, which is quite usual. Then, if a net can not be routed in one box, this net can not be completed in the other box. So, one strategy could be to route all nets that have to cross a boundary first and the nets that stay in the box afterwards. If the last few nets can not be routed in this case, the influence on the surrounding boxes is minimal. On the other hand, to route the incomplete nets in the box becomes very difficult. A solution might be to try to route these nets at a later stadium in a greater area then this initial box. But in the situation that even the not completed nets have to cross the box boundary, some rule has to be formulated to determine which nets are routed and which are not. So this strategy is not so simple as it seems to be.

The other approach is just the other way around, try to route the local nets first and afterwards the nets that cross the boundary. The idea is that for large nets a path can be found outside the box to complete this net. Here the same problems arise as above if all nets have to cross the boundary of a box. If the not completed nets are routed at a later stadium in a bigger area, say a rectangle of 3 by 3 boxes, the algorithm becomes more complex because at each side of the new box the nets have to cross the initial boundary given by the global router, otherwise the correct completion of the nets can not be guaranteed any more. Further, given the set of nets in this new box, no guaranty can be given that these nets can be routed in this bigger sub-area. A way to eliminate these problems is to choose an area in which at least two not connected parts of the net are found and to try to connect these parts in this area.

A last remark to this problem is that, in general, if the space in which a net has to be routed, increases, the routing algorithm will become slower if the space is rather empty. For a gate array environment this space will not be very empty so this approach can be used here.
3. Problem definition.

The task of the local router can be divided in building the routing landscape and the actual routing. The major part of the landscape is built by the image compiler. This initial landscape describes the gate array chip before customisation and is the same for each design made with that chip. Before routing can start, this landscape has to be completed by mapping the modules used for the design in it. This mapping is done with the information given by the placement. The essential information of the macros is stored in a macro library. The placement determines where each macro is situated on the chip. Combined with the information of the library, the exact pin positions of all modules can be located. Next, given the netlist for this design, nets can be assigned to the pins of the modules. After the landscape is completed, the actual routing starts. The results of the global router are used to guide this phase. The global router presents a collection of boxes with nets to be routed in these boxes. With each box a small area on the chip is associated. The nets in a box can be divided into two classes: one class for nets that have positions in the box and the other for nets that don’t. The nets in the last class are feed-through nets: nets who enter the box from one side and leave the box on another side and have no terminals in the box. With terminals, fixed positions are meant, assigned to a net before routing due to a pin of a placed macro. To route these feed-through nets in an optimal way, routing is deferred to a later stadium, where, due to routing in the abutting areas, positions on the box boundaries are assigned to such a net. The nets that have fixed positions in the box can be routed at once. So the routing has to proceed in a recursive way. Each box has to be examined until all nets in the box have been successfully routed or no net can be routed any more. The problem description can now be formulated as follows:

Given a set of boxes, representing sub-areas on the chip. For each box, a set of nets is supplied which have to be routed in that box. Each net is supplied with routing information. The terminals of a net can be located in the chip. For each sub-area, try to find paths in the chip for each net that has a terminal in that box. If a net is complete, delete it. If no more nets have to be routed in the box, the box is ready. Proceed until all boxes are completed or no nets can be connected. Try to connect these nets in an area of the grid in which at least two not connected part of that net are found.
4. The routing landscape.

The routing landscape is a detailed description of the gate array. The landscape can be divided into two parts: one part is fixed and determined by the image description and consists of macro placement and net assignment. The choice which routing algorithm can be used for net routing highly depends on this landscape. The characteristics which determine the choice of routing algorithm are discussed below. Most important feature of gate arrays is that the available routing space is fixed. This means that no extra space can be created to complete the routing. Second, the routing space is very dense. This means that in general, no large free spaces are available for routing. Even in the routing channels, obstacles may exist. This situation occurs if one layer is available for routing and all vias to diffusion or poly underpaths are prediffused. A last important feature is the possible existence of more than one electrically common points in the landscape due to the poly underpaths. As illustration, see Fig. 1.

Figure 1. Electrically common points in the routing landscape due to poly underpasses.

5. The routing primitives.

The routing primitives are already introduced, as the elementary routing operations, in the introduction of this gate array design system. Whatever the choice of the routing algorithm may be, these primitives can be used to control the routing operations.

In fact an abstraction of the routing landscape is made for the router. The router will operate in terms of these primitives, rather than dealing with the detailed landscape, which makes the implementation of the routing algorithm...
easier. With the use of the primitives, all essential detailed information is hidden for the router. This information may be the number of layers available for routing, or if the gate array used has fixed or programmable vias.


The choice which routing algorithm has to be used for routing the nets has to be made carefully. To keep the design system flexible, the router has to use the primitives described above. But also situations as obstacles in the channel or poly underpasses have to be dealt with. This flexibility has to be implicit in the router, otherwise a not acceptable situation arises if the routing algorithm has to be adapted for each technology. In general, routers can be divided into three classes:

1) Maze runners.
2) Line routers.
3) Channel routers.

The most well known mazerunner is the Lee-router [7]. It uses a bounded grid of cells as representation of the routing area. Obstacles are easily incorporated in this algorithm. The algorithm is very flexible with respect to the number of layers available for routing. A very important feature is that if a path between two points exist, it will always be found.

An example of a line router is the procedure described by Hightower [5]. Finding a path between two points is done by generating a collection of lines from the start and target point. If two lines, one belonging to the start point and the other to the target point, intersect, a path is found. The line router can not easily be adapted to handle more than one layer. This means that the primitives can not easily be incorporated in the algorithm. Further, the procedure may fail to find a path even if one exists. This situation is unacceptable in a gate array environment.

A channel router is not at all suitable in a gate array environment. First, to be able to complete wiring, it is implicitly assumed that the channel can be widened. But this increase of routing space is not allowed for gate arrays. Next, handling obstacles in the channel is not accommodated in these algorithms. A last reason for which the channel routers are not usable is that they assume that two layers are available for routing, which is not always the case with gate arrays.

Therefore, the best choice for the routing algorithm is a Lee-router algorithm. A detailed description of the basic algorithm and the adjustments made is discussed in the next paragraph.
The LEE-ROUTER

7. Introduction.

In this chapter the basic net connection procedure is discussed in detail. In section 2 the basic Lee-router is presented. Some aspects of the Lee-router are mentioned in the rest of this chapter. Section 3 deals with backtracing techniques, used to update the minimal path found. Because the Lee-router only connects one net at a time, some aspects of net ordering are discussed in section 4. Further, the more general case that nets have more then just two points to be connected is discussed in section 5. The last section of this paragraph, 6, deals with reducing the size of the search for the minimal path. This is not very critical in a gate array environment, which is made clear in this section.

8. The original Lee-router.

The original procedure as described by Lee [7] deals with finding a path between two points in an arbitrary area. The area is normally seen as a rectangular grid of cells. Each cell represents a position that can be used to extend the path. Obstacles are represented as not-admissable cells. The path consists of a collection of abutting cells. Or stated in an other way : each cell of a path has at least one neighbour who also belongs to the path. The path has to meet some requirements before it is accepted as a solution of the problem. These requirements are formulated as a cost function \( F \) which has to be minimised with respect to the path. If a path from \( A \) to \( B \) is denoted as \( p(A,B) \) and the cost of such a path as \( F(p) - F(p(A,B)) \), then the path connection problem can be formulated as follows.

Given a startcell \( A \) and a targetcell \( B \).

Find a path \( p(A,B) \), if one exists, with
\[
F(p) \text{ minimal.}
\]

The algorithm given by Lee gives a solution of this problem. The main procedure is described below.
Algorithm A : Search and Trace algorithm.

1) Initialise $F(p(A)) = 0$;
   Put cell A in list $L$.
   Clear list $L_1$.

2) For all cells in list $L$,
   calculate for each neighbour cell
   the cost to enter this neighbour.
   If the neighbour is not in $L_1$ yet,
   update this neighbour in $L_1$.
   Else if the cost to enter the neighbour
   from this direction are less than
   the cost to enter the neighbour from
   the direction as recorded in $L_1$,
   replace the older update by this one.

3) Choose from list $L_1$ the cell
   with minimal cost.
   If $L_1$ empty no path exists.
   Record the direction from which
   this cell was last entered.
   Delete the cell from $L_1$ and put the
   cell in $L$.
   If the cell is the target cell,
   a path is found.
   Otherwise proceed with step 2.

4) Backtrace path from B to A.
   ready.

The prove of this algorithm depends on one property of the cost function $F$.
This function has to be monotone not-decreasing. It means that given a path

$$p(A, \ldots, c[i], c[i+1], \ldots B)$$

$$F(p(A, c[i])) \preceq F(p(A, c[i+1]))$$

holds for any $i$.

A sufficient condition to construct such a function is:

$$F(p(c[i], c[i+1])) \geq 0$$

and
The proof given by Lee [7] that this algorithm will find a minimal cost path if one exists, is not repeated here. A note has to be made on the so called expansion of a cell in list \( L \), step 2 of the algorithm. Due to poly underpasses, more points have to be considered for expansion. This situation occurs if the poly layer can not be used to make wires and the vias from metal to poly are fixed. For an example, see Fig. 2 below.

Figure 2. Example of expansion of more then one point in case of a fixed poly layer and fixed vias.

If a net is connected to the left via, the possible expansions from this point are the neighbours of this point but also the neighbours of the right via. This is because the net can now be reached at this right via. The algorithm has been adapted to be able to deal with expansions of more then just one point.


In order to be able to backtrace the path found, in each cell of the path the direction from which it has been entered is recorded. This is done by the algorithm Lee proposed, when a cell is chosen to extend the path under construction. If a 3-dimensional grid is used, this labeling requires three bits to represent the six different directions from which a cell can be entered. In order to save some memory by using less bits, if the grid becomes very large, several other labeling techniques have been proposed. Two of these techniques are discussed by Rubin [10]: a proposal of Akers to label the cells as 0-1-1-0, and a labeling proposed by Moore with labeling sequence 0-1-2. In his article, Rubin proves both methods to be inadequate for backtracing. Unfortunately, this prove is based on an erroneous labeling sequence with the result that the labeling sequence proposed by Moore does indeed produce a correct labeling. The prove of Rubin is given below with the examples he used.
and the essential difference are discussed. Consider first the Moore labeling sequence. Given the grid in the Fig. 3 below.

The cost to enter a cell is denoted in the left top corner of that cell.

![Figure 3. Grid used by Rubin to demonstrate the inadequacy of Moore's labeling technique.](image)

According to Rubin, the labeling sequence will proceed as follows: A-0, B-1, C-2, D-1, E-2, F-0, G-1, H-2, I-0, J-1. Which indeed gives an ambiguous situation were the correct predecessor of cell C can not be known. But with algorithm A the labeling sequence will proceed as follows: A-0, D-1, E-2, F-0, G-1, H-2, I-0, J-1, C-2. So in this situation, no ambiguity occurs.
With the Akers labeling method this essential difference in the labeling order of cells, the situation that the correct predecessor is not known still occurs. Consider the situation in Fig. 4.

Figure 4. Grid used to demonstrate the inadequacy of Aker’s labeling technique.

The labeling according to algorithm A will proceed as follows: A-O, D-I, B-I, E-I, C-I, F-O. Now it is impossible to tell whether C or E is the correct predecessor of cell F.

So this labeling method gives wrong results and has to be discarded. However, if the method of Moore is used to label the cells, some time will be lost in determining which neighbour of a cell is the correct predecessor of that cell along the minimal cost path. So, if memory is not that critical it seems better to use the original labeling method of Lee to achieve a fast and correct backtrace algorithm.


The Lee algorithm gives a nice solution for the situation of finding a connection between two points. But the essential characteristic of routing problems being NP-complete has been deferred to an other stadium: the ordering of nets. The algorithm of Lee is not capable to prevent blocking of nets to be routed in future. To determine in which order the nets have to be routed is NP-complete. An example of a critical net ordering is given in Fig. 5.
Suppose Manhattan distance is the criterion to be minimised. Now, if net A is routed before net B, net B will be blocked and cannot be routed successfully. On the other hand, if net B is routed before net A, no blockage of net A exists and both nets are routed successfully.

Several methods have been presented to deal with this ordering problem. Unfortunately, they deal with a rather empty routing space, which does not hold in a gate array environment. Some ordering methods have been investigated by Able [1]. His conclusion is that most known ordering methods have no essential effect on the number of nets completed. Further, a proposed ordering scheme has to be very flexible in order to fit in the design system. Any use of a priori knowledge of the routing landscape is not allowed in this situation. The available a priori knowledge in our design system is so little that none of such methods are capable here, or they become so complex and time consuming that they are not practical any more. A last remark is the performance. It is doubtful if the results of a heuristic net ordering hold in this environment. Because the routing space in gate arrays is very dense, many calculations to determine an optimal net ordering are not feasible. What is needed is a very simple and fast ordering which gives reasonable results in most of the time. That the time spend to determine an optimal net ordering has to be limited, comes from the fact that the routing is performed in a sub-area of the chip. Therefore, for each sub-area, an optimal net ordering has to be determined, which is much more times than the number of nets to be routed. The solution chosen in this situation is to try a number of orderings until a solution is found. To limit the time spend in finding a solution of the interconnection problem, not all orderings are tried. If a net can not be routed initially, it can be discarded because this situation is not due to net ordering. With initial routing is meant that a net is routed as if it would have been chosen
to be the first net to be routed. A method has to be developed to route these nets in a sub-area greater than the initial area as mentioned before in section 2.

11. Multi-terminal nets.

In literature, only two terminal problems are considered in relation with the Lee algorithm. However, in general nets have more than just two terminals. To divide this problem into sequence of two terminal situations gives no solution. This is because the start and target points have to be chosen. How to make an optimal choice is not trivial. Therefore, the Lee algorithm is extended to deal with this more general case. The basic idea is to start from all given net positions at once. Every set of electrically common positions is marked as one subtree of the net. When two subtrees meet, an optimal path is found and afterwards, the algorithm starts again until one subtree is left. Now all subtrees are electrically common so all pins are connected in this sub-area. By starting to find a path from all sub-nets at once, a more natural inter-connection pattern will be obtained. Further, for each sub-problem, the optimal path will be found, which can not be guaranteed if an a priori choice of start and target points is made. A further advantage is that the search space is reduced. This topic will get more attention in the next section. A very nice feature of starting from several points at once, is that, in case of two subtrees to be connected, if no expansion from one subtree is possible, it is known that no path can be found to connect these subtrees. In case of more then two subtrees, this conclusion can not be made this soon, because it is possible that the remaining two subtrees can be connected.

12. Reducing the size of the search.

The complexity of the Lee algorithm is \(O(n^2)\). This can be seen as follows: Suppose the distance of two points to be connected is \(n\). If all cells of distance \(< n\) are expanded, the number of expanded cells will be \(2n(n + 1)\), which is \(2n\) for large \(n\). If the search will start from two points at once, about \(n/2\) levels will be expanded which gives \(2 \times 2(n/2)^2\) cells, which is about half the cells as before.

The drawback is that the cells have to be labeled to be able to detect if a path is doubling upon itself or not. But in the present situation, this is no problem, because in case of multi-terminal nets, this distinction has to be made anyway.

An other important change is that of the cost function. This function has to

7.143
be path-consistent, which means that if \( F(p(A,B)) \) is a minimal path and \( F(p(B,C)) \) is minimal, then \( F(p(A,C)) \) is minimal. This limits the flexibility of the cost function.

Another search space reducing principle has been introduced by Rubin. The idea is to use a predictor to guide the cost function. The predictor uses the distance to the target cell as a lowerbound of the cost of the minimal path. This procedure can not be used in case of a multi-terminal net because the exact target is not known. It is also impossible to choose a target point to predict the cost, because it is not sure that a path can be found to this point.

However, the search space of the gate array environment is already limited because of the density of the routing space, due to macro wire patterns and limited route space.

13. Conclusions.

The Lee router has been successfully extended and implemented according to the remarks made in this report. The router has been extended to handle multi terminal nets and situations due to poly underpasses. The local routing program has been implemented in the language C on a HP9000 host computer. As test case, a hand made design with the AMI gate array has been used. The results of the global router indicates no bottlenecks for this design. However, the local router has difficulties to complete the routing of this design. To eliminate possible conflict situations due to net ordering, the algorithm has been extended with the possibility to try a limited number of orderings. The result has been improved after this extension, but is not good enough to compete with other design systems. A start has been made to try to connect the remaining nets in a greater sub-area of the chip. At this stage the result of the global router will not be used any more. It is expected that this will improve the result to an acceptable level. At this moment, the time needed to reach the situation in which no nets can be routed any more is about one and a half hour. If the router has been extended to solve the remaining net connection problems, this time will increase. Together with the time needed to get an acceptable placement and a global rout, the overall time to complete a design will be about 1 day. Most of the time will be used for placement, but this time can be controlled by the user of the system. The global router is the fastest program, for the example mentioned above, only a quarter of an hour was needed to obtain the result.

Overall conclusion is that a gate array system adaptive to many technologies
can be developed and, with the extension of the local routing, will be able to compete with other gate array design systems which are designed for one technology only. Two examples are given in colour plates (). In the first colour picture, an example is shown of an one metal gate array with poly-underpasses and fixed vias. The second picture shows an example of a two metal layer gate array where the vias are programmable.
14. References

[1] Able L.C.
On the ordering of connections for automatic wire routing.

A modification of Lee's path connection algorithm.

[3] Deutsch D.N.
A dogleg channel router.

Steiner minimal trees.

A solution to line-routing problems on the continuous plane.

A gate array design system adaptive to many technologies.

[7] Lee C.Y.
An algorithm for path connections and its applications.

[8] Nuijten P.A.C.M.
Hierarchical wire routing of gate arrays
Eindhoven University of Technology, Department Electronic Engineering, master thesis.
A ‘greedy’ channel router.  
Proc IEEE 19th Design Automation Conf., 1982,  
pp. 418-424.

[10] Rubin F.  
The Lee path connection algorithm.  
IEEE Trans. on Computers, Vol. C-23, no 9, 1974,  
pp. 907-914.

The complexity of design automation problems.  
Proc. 17th Design Automation Conf. IEEE, 1980,  
pp. 402-411.

Routing techniques for gate array.  
IEEE Trans. on CAD vol CAD-2, no 4, 1983,  
pp. 301-312.
Chapter 8

Workstations
The CADMUS 9230
ICD Graphic Workstation 1

By D. Larsson, Dr. P. Schinner et al.

Topics:
Communication requirements
Description of Hardware Components
Description of Software Components
Communication Procedure
Interactions between the Software components
Portability and Compatibility
Error detection and recovery
Conclusions

The work described here was carried out with the partial support of the EEC under Microelectronics project MR-09-DFT (ICD).
1. Abstract

This report describes the configuration of the first-generation Cadmus 9230 color graphics workstation built for the ICD project by the graphics group of PCS in Munich. We will discuss the multi-processor hardware design and the communication and data methods and formats we used in some detail as an introduction to this system. In lieu of a detailed evaluation of this system, we invite the interested reader to study the further development generations of our Cadmus-based workstations for this project's applications.

2. Communication Requirements

To connect the Host Computer System and the Graphic Processor subsystem a fast bidirectional communication link must be implemented. It is preferred that different Host Application programs should be able to communicate with different programs in the Graphic System in a "parallel" manner. These requirements calls for functions like lock, unlock, etc, to avoid deadlock situations, and a I/O scheduler function to differentiate between different priority I/O requests, and a mechanism to make it possible to identify messages between individual programs in the Host System and the Graphic System. This implies that a powerful software package is developed to manage the whole complex of communication between the two systems.

3. Description of Hardware Components

3.1. Overview

As stated in the Technical Annex, the Graphic Workstation System should preferably consist of a dual processor system, both systems equipped with a MC68000 processor, or equivalently sophisticated processor chip, connected by a fast parallel bus. One of them will serve as basic design system, henceforth called the Host System, while the other will be dedicated to graphic processing, controlling the Graphic Display and operator interaction, and henceforth called the Graphic System.

The Host System will run under the UNIX Operating System, and the Graphic System will need a specially tailored real-time-multitasking Operating System. The Host System will support the Data Base Management System and execute most of the functions defined in the other Tasks of this Project. The Graphic System will perform all graphic oriented operations, especially graphic editing functions. The CADMUS 9230 Computer System has been found an appropriate Workstation System, as it can be equipped with a graphic coprocessor directly connected to its Q-Bus.
3.2. CADMUS 9230 Computer System

This MC68000/Q-bus-based computer forms a very powerful general purpose supermicrocomputer, its is very fast due to a dual-port memory-technique, this means that the processor board and the memory boards are connected via a special bus, the S-bus, thus allowing the 10 MHz processor to access memory with only one waitstate. On its Q-bus you can use a very wide variety of peripherals, as the Q-bus is one very commonly used mini-computer-bus. The processor also has a memory management unit that transforms its 16 Mbyte address space to the 4 Mbyte available on the extended Q-bus. A new Model using a new Version of the MC68000 CPU-chip that will support true virtual memory management is under development. With 1.0 MIPS and up to 440 MBytes of Winchester Disc Storage the CADMUS 9230 is able to handle large System consisting of 16 or more connected alphanumeric Terminals, printers, etc.

3.3. CADMUS Graphic Processor Board

This quad-slot Q-bus-board forms its own complete computer processing unit. It has 2 serial interfaces (RS232/RS422), one parallel interface and a mouse interface to communicate with it's peripherals. The local MC68000 CPU-chip can access 16 KByte of onboard EPROM and 256 KByte of onboard dynamic RAM's. Interrupts can be handled from serial and parallel interfaces as well as from/to the Q-bus interface. Over the Q-bus the Host System can access all of the onboard memory and peripheral interfaces. The Graphic System is capable of controlling a Display System, consisting of a Pixel Processor with pixel memory and a Colour Display Monitor. The Display System could be connected to the Graphic System by a serial or a parallel interface, where the parallel interfacing is to be preferred.

4. Description of Software Components

4.1. Overview

The different software tasks such as:

- Data-base management
- Design system/foundry interface
- Multi level verification
Section 4.1

Overview

- Mixed level simulation
- Data format conversions
- Graphic processor operating system
- Interactive graphic editor
- Algorithmic description and compilation
- Placement and routing

can be split between the two processors as follows:

Graphic System
- Graphic processor operating system
- Data format conversions
- Interactive graphic editor
- Low level Design Rule Checking

Host System
- all other tasks

This means that all graphic display and interactive graphic operations needed in the software running on the Host System must perform these functions via the interactive Graphic Editor running on the Graphic System. This demands for a powerful and efficient communication between the Graphic System and the Host System.

4.2. Host Driver

The Host Driver Program must handle the difficult task of arbitrating between communication requests only concerning output to the Graphic System or input from the Graphic System and communication requests that implicate a dialogue, that is to request some information from the other system and await the answer. This latter type of communication will occur in several cases, for instance:

- Host System application program requesting status of Graphic System
- Graphic System application program requiring graphical data, for instance a macro-cell-definition.
To fulfill these requirements the Host Driver must include differentiated open/close calls, as well as a set of i/o-control calls.

The Host Driver also must implement a handshake protocol to synchronize its actions with the Graphic System Driver.

4.3. Host Communication Tasks

The Graphic System must be able to initiate communication, one simple case is mentioned above, and therefore there must be some software in the Host System to serve these requests from the Graphic System. In the first version these functions will mainly be of the type 'get database information' or 'put database information'. This Task must probably be running all of the time the Graphic System is used as a Graphic Workstation.

It may also be convenient if you can use the Graphic System as a standard Terminal when not using it as a Graphic Workstation.

4.4. Graphic System Driver

This Program will mainly have the same tasks as the driver in the Host System. It will also have to manage some buffers in the onboard memory for communication uses. These buffers are implemented to speed communication with the Host System by significantly reducing the number of handshake procedures needed. The Graphic System Driver will also interact closely with the Graphic System Operating System, as requests from the Host System can activate different programs in the Graphic System.

4.5. Graphic System Communication Task
The Graphic System Communication Task will have some very important functions in the Graphic System, as

allocating and managing buffer memory for communication and

control the allocation and management of the display file memory and

control the conversion / reconversion between data formats and

activating the appropriate function in the Graphic System software according the request from the Host System.

As the manager of the display file in the Graphic System this task will play a key role in the Graphic System software system. It must communicate with practically all other tasks in the Graphic System, and thereby guarantee data integrity in the display file. Accesses to the display file must in any case be fast and efficient.

5. Communication Procedure

5.1. Overview

There are two major cases of communication between the Host System and the Graphic System, namely

using the Graphic System as a normal alphanumeric Terminal

using the Graphic System together with the Integrated Circuit Design System.

In the first case the communication will be very simple, just like a normal terminal with 8-bit data-bytes. The Graphic System acts as a more or less "dumb Terminal" and the normal UNIX terminal facilities are available for any Host System Program.
The second case however is much more complex. It will need the transmission of different format data and the identification and connection of different programs as communicating partners. It also implies the need for faster data transfer, as large amounts of data representing graphic information will be sent from time to time.

5.2. Communication Method

The basic communication will rely on a handshake protocol, used to synchronize Driver accesses to the communication buffers allocated in the Graphic System memory. Extensive use of interrupts will be implemented to reduce processor I/O overhead. Although the primary goal is to define a communication scheme for a parallel communication connection, we will also take into account implications of a possible solution with a serial connection. This will lead to two slightly different definitions, where differences will be held at a minimum to improve portability. The handshake protocol will implement the following states:

- Idle
- Host requesting to send
- Graphic System requesting to send
- Host sending data
- Graphic System sending data

In this bidirectional communication scheme you can use multiple communication buffers to speed handling of transmitted data, but this communication scheme would complicate the Driver design. For error detection reasons it would be convenient if a time-out detection function could be implemented.

5.3. Communication Formats

The communication will be based on an 8-bit Byte-oriented transmission, with no interpretation of any control codes. Data of longer format, (e.g. 16 or 32 bits), are transmitted most significant byte first. Synchronization between the communicating Systems will be achieved by a handshake protocol.
Section 5.3 Communication Formats

In each handshake sequence a block of data will be transmitted, where each block is divided into two parts, namely a header part and a data part. The header part must contain information about:

- Header block length
- Communication / Program function
- Sending program identification
- Program identification of receiving program
- Data block length

The header part can optionally contain information about:

- Data format information
- Data load addresses
- Header checksum

The data block can contain the following information:

- Binary Byte memory image format
- Program code in program load format
- Graphic data in display file format
- Graphic data in data-base format
- Data in ASCII format

The data block can be empty.

The data block should if not empty using the serial transmission concept contain a checksum.
5.3.1. Considerations for a serial connection

A Detailed description of the communication formats

Both block types have the same general structure,

\[ \text{length, first data, ... last data, two byte checksum,} \]

where the \(<\text{length}\)> is a 8-bit unsigned binary number (0 - 255). The \(<\text{length}\)> value is calculated as the number of data bytes, the \(<\text{length}\> \text{ and } <\text{checksum}> \text{ bytes are not included.}\) The \(<\text{length}>\) value 0 corresponds to a block length of one byte, while the \(<\text{length}>\) value 255 means a block length of 256 bytes. A larger block length was rejected to establish a good error detection probability with the aid of checksum calculations.

Data is defined as 8-bit bytes.
The checksum generation is described in the "Error recovery" part of this document.

In the following description we will omit to mention the \(<\text{checksum}>\) part for greater clarity, it must however always be contained in the transmission of a communication block.

Header block formats

The header block must contain the following information, each one byte long, except for the checksum which is two bytes long:

- Header block length
- Communication / Program function
- Sending program identification
- Program identification of receiving program
- Data block length
- Header checksum
The header block can contain the following optional information, (this is a function of the <Communication / Program function> field):

Data format information
Data load addresses

where the load address is of 4 byte size.

Data block formats

The data block can have one of the following formats:

- Binary Byte memory image format
- Program code in program load format
- Graphic data in display file format
- Graphic data in data-base format
- Data in ASCII format
- The data block can be empty

The data block should if not empty contain a two byte checksum.

5.3.2. Considerations for a parallel connection

Detailed description of the communication formats

Both block types have the same general structure,

length, first data, ... , last data,

where the <length> is a 16-bit unsigned binary number (0 - 64K). As this is a direct memory access technique, the need for an error checking algorithm is very low. The <length> value is calculated as the number of data bytes, the <length> byte is not included. The <length> value 0 corresponds to a
block length of one byte, while the <length> value 64K means a block length of 64K+1 bytes.

Data is defined as 8-bit bytes.

**Header block formats**

The header block must contain the following information, each one byte long, except for the length which is two bytes long:

- Header block length
- Communication / Program function
- Sending program identification
- Program identification of receiving program
- Data block length

The header block can contain the following optional information, (this is a function of the <Communication / Program function> field):

- Data format information
- Data load addresses

where the load address is of 4 byte size.

**Data block formats**

The data block can have one of the following formats:

- Binary Byte memory image format
- Program code in program load format
- Graphic data in display file format
- Graphic data in data-base format
Data in ASCII format

The data block can be empty.

The length part must always be contained in the transmission of a communication block.

In this scheme functions to request a memory buffer in the Graphic System could be implemented, to enable the Host System to load data directly into the work storage of Graphic System programs, thus avoiding extra overhead in communication buffer handling. This would require a possibility to shut the application program in the Graphic System out of accessing these data while transmitting to avoid errors due to inconsistent data.

6. Interactions between Software Components

There will be a very close interaction between all software components involved in the communication. Therefore special care must be taken in designing this system, to avoid deadlocks and other equally unpleasant situations.

One good way is to define each software component as a finite-state-machine, with each state change depending on one or more of the other software modules, also described as finite-state-machines. With this method it is possible to describe the complex handshaking procedure and integrate it in the different software components.

7. Portability and Compatibility

As it is preferred that the software developed in this project should be portable between a wide variety of different computer systems, certain measures must be taken to provide this feature. However due to the specialized Hardware in the Graphic System there may be severe constraints placed on the porting of this part of the software. With careful software design it will be possible to keep these hardware dependencies in a relatively small number of routines.

Our first goal will therefore be to define generalized software interfaces, to ease the usage of different Display Systems. We will also try to design the communication software to be as independent as possible of the actual hardware communication channel between Graphic System and Host System, to enable the usage of the Graphic System with other
Hosts than the CADMUS machine.

8. Error detection and recovery

In a system with such complicated communication requirements some effort must be invested into making the communication as fail proof as possible or in case of an failure enable it for the application programs to recover gracefully. There are several classes of errors to be considered:

Transmission errors
Syntactical errors in a transmission block
Errors detected while executing a communication function in the Graphic System or in the Host System, (ex. a requested macro-cell does not exist in the data-base)

In the communication software only errors of the first kind (transmission errors) and some instances of syntactical errors can be dealt with. Reporting error conditions occurring while executing or interpreting some transmitted data must be done on the application program level using normal communication facilities.

It can be assumed that transmission errors have an extremely low probability when transmitting via the parallel Q-bus, and thus we will not implement any checking functions in this case. The serial transmission case however needs an error detection mechanism.

8.1. Transmission errors on a serial connection

Errors in data transmitted can be:

detected
corrected.

In this implementation a simple error detection scheme will do, because it is relatively easy to retransmit a block after an error occurred. Therefore each byte should be transmitted with one parity-bit. Another simple method to detect errors is
Section 8.1 Transmission errors on a serial connection

by calculating the checksum of each transmitted block and append this checksum to the block, this procedure is repeated at the receiver end of the line and the checksums compared, if they differ an error occurred and the corresponding error report should be produced. The checksum will be calculated as an unsigned 16-bit integer by adding all transmitted unsigned 8-bit bytes, that is adding up the <length> byte and all <data> bytes without carry-rotation in the 16-bit integer.

8.2. Syntactical errors in a transmission block

No syntactical errors can be detected by the communication software in data parts of blocks.

As syntactical errors we classify syntactical errors in a header block, like specifying unknown functions or program identification numbers. Also specifying a false block length is considered a syntax error. Syntax errors of the first kind are relatively easy to detect, while the second type can cause more trouble. The sending program will have to specify the length of the header part while the data part length can be either implicitly specified (e.g., through a function) or explicitly specified in the header. Specifying a too small block length can be detected already on the transmitting side, but if the first part of the block is already transmitted we could use a break function, or we let this be detected by the receiver, while a too large block length only can be detected by a timeout, if at all (by the transmission of large blocks we must consider the operating system latency in the Host System).

In the case that a syntactical error is detected in the receiving driver program, an error condition report will be produced and returned to the sending program. The driver program on the sending side will then be able to distinguish between error reports that it can handle itself, and error reports that have to be handled by the application program.

9. Conclusion, Evaluation, Notes

We have described the hard- and software architecture of the first generation of our color graphics support systems. This work was done in early 1984 and it can now (late 1985) be said that it was a good and well-built system that was implemented. Due mainly to the elements of high cost and large size however, we have since changed the hardware (simpler subsystem
without a full coprocessor) as well as software (Macintosh-like world with integrated support on several levels) design of the later generations of this system.

The coprocessor system was built, programmed and demonstrated by our group and has since been succeeded by the PCSMac software and the CGC4 color graphics controller hardware that are documented elsewhere. The newer data and communications formats are more closely tied to widely-used standards like the Macintosh Toolbox and VDI as described in an accompanying paper by Richard Cole. We do feel however, that this first generation design was an invaluable experience for us and lead to the implementation of a useful and powerful tool for color graphics software development and evaluation. We therefore offer these comments for others planning to build such a system.
Notes on the PCS Color Graphic Controller – CGC–4

Dr. Klaus Lueddecke, Stephen Pope
PCS – Periphere ComputerSysteme GmbH
RTK / BST Departments
Munich, September, 1985

Abstract

This short technical note is an introduction to the motivation and hardware architecture of PCS’s new CGC–4 color graphic controller for use with our Cadmus 9000 Series 68020–based general–purpose computers. We have developed this hardware after long experiments with various other manufacturers’ systems and considerations of hardware and software design for an integrated color graphics workstation that should offer a very high price/performance ratio.

Introduction

PCS has developed several versions of black–and–white and color graphics hardware for our Cadmus 9000 series of computers. Currently, we deliver the Cadmus 2200 bit–mapped B&W terminal (also referred to as the BMT) with a coprocessor for graphics operations and we have specified and designed a new color graphics subsystem for the new Cadmus 68/32, 68020–based computers.

This note describes the steps that lead to the specification of this subsystem and its characteristics. The prototype is currently (late September, 1985) being produced and we can expect to comment on the performance of the integrated HW / SW system before year’s end 1985.

We will describe some of the background of the decisions that lead to the determination of the basic characteristics of this system and the new Cadmus 68/32 computers before presenting the CGC–4 in more detail. This note has been prepared to generate a discussion of the design of future additions to this basic color graphics line and to help shape the software structure for this new configuration.

History of the Development Generations of PCS CGC

PCS has used several different color graphics subsystems from various manufacturers in the past, including products from Raster Technologies, Rampage Inc., Tektronix and others. We decided to design our own quad card Q–Bus format color graphics subsystem after analysing the available systems and the requirements of our customers and users. In discussions at PCS, we set out the basic characteristics of a new, workstation–based, high–performance, affordable color graphics system for integration into the Cadmus 9000 68020–based computers.

We have gone through several different design proposal stages and compared varying

* – This work was carried out with the partial support of the EEC Microelectronics Project MR–09–DFT (ICD)

September, 1985

8.16
architectures and workstation configurations with respect to our concerns of overall system performance and price per user. We have also placed the design strategies we proposed into the current and expected color graphics marketplace, showing (on paper) our Cadmus 68/32 + CGC-4 system aside future competitor architectures.

There are several important questions that dictate the specification of any such system:

- desired cost range of the subsystem
- graphics resolution
- user input possibilities
- interconnection with host computer
- system speed
- software integration & library support for applications

The previous phases and versions of this design varied substantially in several of these most basic respects. We started with a 2-card subsystem offering coprocessing capacity and a deep frame buffer. This version was discussed at the ICD project general meeting in Enschede in May, 1985. In further discussions in-house at PCS, the large performance gain provided by such a system increasingly seemed to be outweighed by the cost and power supply load of this alternative architecture. The next version designed that was approached differed in that it was decided to remove this 2-card subsystem from the host's Q-Bus and install it in its own chassis (with its own power supply), to be mounted under the color monitor (as seen in several other manufacturers' comparable systems).

The full specification of a color system in this form factor (as a separate box) was delivered to our marketing and management during the early summer of 1985 and it was then decided, for technical as well as commercial reasons, to simplify the design again and make a single-board frame buffer (without a coprocessor), to be used with the new 68/32 host. This became the first of our designs to be carried through into production and is currently called CGC-4.

The Cadmus 68/32 Machine Architecture

Before discussing the CGC-4 in detail, one should examine the basic architecture of the Cadmus 68/32 68020-based workstation systems. These computers offer approximately 4 times the performance of older, 68000-based systems like the Cadmus 9200 series, placing them at least in the range of DEC's larger VAX-11/780 systems. The system architecture of the Cadmus 68/32 can be diagrammed as follows:
Cadmus 68/32 Architecture Block Diagram

Basic Cadmus 68/32 Characteristics:

- 68/32 Processor – MC68020 CPU – Dual-bus system on 2 quad cards
- Proprietary fast paging MMU
- 16 KB on-board Cache memory
- 68881 Floating-point support
- High speed (40 MB/Sec) main memory bus (P-Bus)
- Simple and compatible I/O bus (Q-Bus)
- 4 to 16 MBytes P-Bus Main Memory (1 to 4 boards)
- ICC Communications controller – 68000 Coprocessor with interfaces to:
  - SCSI bus (for disk, tape, floppy etc.),
  - Ethernet (LAN) and
  - RS232 serial lines (WAN, terminals etc.)

It is worth mentioning that our bus connection structure for the 68/32 has changed relative to the earlier Cadmus 9000 computers based on the MC68010. We now use one half of a quad card's connectors for the P-Bus connection instead of the S-Bus method of ribbon cables between the backs of the cards.

September, 1985

8.18
For this advanced and high-performance hardware, we have now implemented a color graphics system for use in any of a wide range of applications areas. It is important to repeat that we view color graphics as a continuum in both time and equipment. We cannot try to develop an ATAP (all things to all people) and that we cannot try to develop a system that will remain competitive or useful for a lifespan of more than about 5 years. The CGC-4 design we have reached will certainly exceed the requirements of several major applications areas needing improved color graphics hardware/software systems. There will remain, however, some areas better suited by foreign hardware on Cadmus host computers.

The Fourth Generation CGC Hardware

We can now discuss the basic architecture of the CGC-4 in light of its predecessors and its hardware system surroundings. The decision to make CGC-4 a frame buffer, i.e. to remove the coprocessor and local program memory was met after special deliberations about the price-performance of the resulting integrated workstation. The previous designs entailed adding a coprocessor to a 68010-based Cadmus 9000 (i.e. a 9230 or 9600) as is the case with the BMT. It was a question we raised whether a frame buffer in a more powerful, 68020-based system couldn’t achieve higher performance at lower cost with easier SW design. The current CGC-4 architecture with a single-user Cadmus 68/32 host can be expected to deliver high performance, high resolution color graphics response at the system level at lower total cost than comparable 68010 designs or other coprocessor (RasterTech etc.) systems.

The CGC-4 hardware is packaged on a single Quad Q-Bus card and includes the color monitor and user input devices in the subsystem. The P- and Q-Bus interface circuitry and memory elements are the centers of the CGC-4 and fit together with the I/O and (display and system) control components on one (full) quad card. The edge connectors for video and other I/O allow the system to be mounted in cabinets of the Cadmus 9600 series computers, bringing an important price advantage relative to the older designs.
We are currently awaiting delivery of the prototype controller hardware and expect to assemble the prototype system (with monitor & 68/32) during the first half of October. Basic SW drivers are the next step, before we can report more about the system's operation.

**Color Graphics Controller – 4 Architecture**

The main new features of CGC-4 are the large frame buffer memory, the P-Bus DMA connection to the host's CPU and the interrupt connection to the Q-Bus. The specification of a very-high resolution (1280 * 1024) monitor can be seen as one of the most important design decisions. The flexible configuration of input devices is also an important point of the specification.

**Characteristics of the CGC-4**

- Bit-mapped color graphics subsystem for use with Cadmus 68/32 Computers
- Color Graphics Subsystem – 1 Q-Bus / P-Bus card + Color monitor
- High resolution – 1280 * 1024 visible pixels – 60 Hz refresh rate  
  (pixel frequency = 109 MHz)
- 4-bit planes in hardware – 16 colors out of 16 million in color table
- 1 MB Frame Buffer memory (640 KB visible, 384 KB for fonts, icons etc.)
- Local control of mouse, tablet and/or keyboard input
- Direct frame buffer connection to (synchronous) P-Bus processor & host memory
- Asynchronous Q-Bus control, interrupts & I/O cycles
- 32-bit transfers on P-Bus, 99% of buffer bandwidth free for CPU cycles
- 16 million colors through controller local lookup tables  
  (4 times more available through software dithering)

September, 1985
Pixel I/O via P-Bus (40 MB/Sec) DMA
UNIX SW driver as memory device – simple and portable SW development

CGC-4 Features relative to Previous Designs

Configurable as standalone color graphics system or workstation in a network
SW development on main 68020 CPU
Optimal use of 68/32 FFP and/or future FFP accelerators expansion
Very cost-effective – subsystem for approx. DM 30000 add-on to 9600 box 68/32
CPU sees frame buffer as normal memory with MMU
(permits virtual windows & bitmaps...)
Frame buffer data is not host cached, 2 caches (internal or external)
(fast graphics primitives)

Status of the Project

As mentioned above, we have finished the layout and expect a prototype board soon. The software support for the basic graphics functions and terminal emulation must follow before we can use these prototypes for their tests.

We plan to begin moving SW from the BMT to CGC-4 when several stable systems can be delivered to the graphics SW group in BST. The basic integrated system should be demonstrable before the end of 1985.

The planning of the basic software, that entails the MUNIX kernel driver, the host library interfaces and the basic terminal emulation can be carried out in advance of the delivery of the development systems, provided we have sufficient access to the early 68/32 systems and MUNIX therefore. We will, in any case, be able to report on the tests at one of the upcoming ICD manager meetings and will produce a short report thereof.

September, 1985

8.21
The PCS Graphics Software World 1985
Richard H. Cole
PCS BST1 Graphics Group

Topics of Discussion:
0. What do We need?
1. Our Proposal
2. Basic Graphics Library
3. PCSMac
4. Meta Graphics Library (VDI)
5. BGL, PCSMac and Meta Graphics Implementation for B&W and Colour

Abstract
This short note discusses some of the factors that determined the design of the current PCS color and black-and-white graphics software systems. We present the demands placed on this software by users and applications programs and our proposal for the structure of the software components. The state of this programming is very varied, i.e. some components are almost finished while others remain to be fully specified. This note is in the form of an outline because we have specified several of the software elements fully in other reports and are still involved in the perfection of the system interaction and data transfer methods.

0. What do we need?
PCS users need a graphics world that:
- allows graphics for CAD customers as well as customers who wish an engineering workstation with windowing.
- Software to operate on both the PCS B&W and Colour Hardware
- allows Multi-User use of the machine
- allows Multi-Processes to run simultaneously
- Supports the rapid and easy portation of Graphical Standards.
  - VDI, GKS, 41XX emulators
- Delivers Maximum Performance.

This work was carried out with the partial support of the EEC under Microelectronics Project MR-09-DFT (ICD)
1. Our Proposal
The current generation of PCS graphics software can be described as needing more strict integration as to
the levels and applications areas of our high and low level software support. This has been historically
determined by the multitude of areas we have to support. To alleviate this difficulty, we plan a
well-thought-out world, i.e. a set of libraries, applications and formats, that allow the use of several
widely used standards like GKS, Macintosh and/or VDI. Our planned structure might be visualised so:

```
<table>
<thead>
<tr>
<th>PCSMac Application Programs</th>
<th>Meta Graphics Libraries (VDI?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Terminal Win, Paint Qsh, Finder</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCSMac</th>
<th>CAD Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>M G S</td>
<td>G K S</td>
</tr>
<tr>
<td>M C A</td>
<td></td>
</tr>
<tr>
<td>C N</td>
<td>V D I</td>
</tr>
</tbody>
</table>

PCSMac Windows Graphics

Basic Graphics Library

The elements of this environment can be summarised as follows:

**Basic Graphics Library**
- Graphics Primitives; line, circle, bit-blit, ...........
- Multi-Processing, Multi-Window
- Terminal Emulator Output mixed with Graphics
2. Basic Graphics Library

The routines are divided into 3 classes: Output, Input and mixed input/output.

<table>
<thead>
<tr>
<th>Output</th>
<th>Mixed</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal emulator</td>
<td>Cursor Tracking</td>
<td>Events,</td>
</tr>
<tr>
<td>– vt100, 4014</td>
<td>Menu</td>
<td>– Mouse</td>
</tr>
<tr>
<td>Layers</td>
<td>Pick</td>
<td>– Keyboard</td>
</tr>
<tr>
<td>Drawing Primitives</td>
<td></td>
<td>– Tablet</td>
</tr>
<tr>
<td>with Fill</td>
<td></td>
<td>– Attributes</td>
</tr>
<tr>
<td>Text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit-Blit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Colour</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cursor Tracking, Menu functions, and Pick functions are in a mixed class since they couple one of the input devices to screen output.

The Basic Graphics Library should contain the basic 2D graphic primitives. The coordinate system for these primitives is integer hardware coordinates.

This interface should become the PCS Standard low level graphics library, and should be present on all PCS graphics hardware, both B&W and Colour.

The Mxt driver allows simultaneous multiple processes to access these primitives.

Since this software is implemented locally on the controller or in the driver the user gets real time response for cursor tracking, without significant load to user processes and without polling (as with the competition's software).

The underlying software for the BGL already exists.

Since this is the underlying software for both CAD/CAM applications and PCSMac they can co-exist in the same graphics world.
3. PCSMac Libraries

The Cadmac / PCSMac package offers:

- Macintosh style of Multi-Window user interaction.
- Based on the Macintosh documentation and the PCS Cadmac code.
- Ported to run on the Basic Graphics Library.
  - **multi-user**: multiple bitmap and multiple user machines
  - **multi-process**: output from simultaneous processes
  - can run with any other BGL based application
- Status:
  - we have large amounts of Cadmus code in a semi-finished state
  - lacks application programs - worst missing piece is a reasonable desktop shell

The following figure shows some of the relationships between the different levels and packages we discuss here. It is important to note the applications library, UNIX Kernel and RTK Kernel levels that we differentiate for optimal performance and portability between different hardware systems (ie. B&W and color).

- This library is intended for use with all software that needs higher level graphic primitives, i.e. low end CAD applications. The Meta Graphics Libraries are also the interface for the implementation of various standard graphics libraries: VDI, GKS, PHIGS, Core, etc.

- Could best be implemented as a full VDI interface.

- Implemented on top of the Basic Graphics Library, and therefore programs running with this library can also be called from a PCSMac window.

- Contains primitives for
  - Viewport–Window Transformation
  - Graphics Primitives
5. BGL, PCSMac, and Meta Graphics Implementation for B&W and Colour

For B&W the Basic Graphics Library is implemented in the driver and on the bitmap controller as multiple processes under RTK. For the colour implementation however, there is not enough local intelligence and therefore the full BGL is implemented in the driver, using a kernel process on the host.

The graphics primitives of the Basic Graphics Library are based on Robert Pike's Layer concept. Currently we have the basic layer primitives (a layer is an overlapping virtual bitmap) and primitives for line, circle and bit-blit working under RTK on the bip. The new vt100 emulator is also underway, this emulator,

- uses real vt100 escape codes
- works in layers and therefore in windows with graphics
- includes the vt100 semigraphics required by some users

These primitives will also be the basis of the BGL on the Colour system.

Major Components:

- Mxt Driver: Multiplexes the process I/O
- OS. Layer Mgr: together with the Mxt driver the layer Mgr. determines the active process.
- OS. Event Mgr: holds the active process event queue
- Graphics Mgr.: carries out the graphic primitives on the bitmaps and keeps track of the layers as well as the process output attributes
- Event Mgr: produces events that are sent to the driver and keeps track of the active process input attributes
- Cursor tracker: is the process responsible for tracking and updating the cursor

The following figures depict the interaction levels and communication paths for the color and B&W versions of this software.
6. Implementation steps
The PCSMac software has been written and will be delivered in a test version in late 1985 on the BMT-1 hardware without RTK. We are developing (coding) the RTK Layer software in parallel with this as well as with the basic color support software. By mid-1986 we hope to have the MXT and Layer software in use and begin implementation of the other components as described. The cooperation with several user groups within the ICD and FHG projects is a motivation and help for our work.
Chapter 1.4 - Plate 1
The Schematics Editor: An edit of a filter circuit, with a view into one instance, showing the opamp circuit.

Chapter 1.4 - Plate 2
The Schematics Editor: Zooming in automatically activates more details, such as instance names and terminal names. The context window (left bottom) shows the position of the window, relative to the instances. The grid is optionally shown.
Chapter 1.4 - Plate 3
The Schematics Editor: An edit in the representation area of an opamp circuit. Besides the symbol itself, formal parameters can be edited.

Chapter 1.4 - Plate 4
The Schematics Editor: A mixed analog/digital circuit is edited. Blue denotes digital, green denotes analog. These two types cannot be connected together directly, however, components can define relations between the two types of terminals.
**Chapter 1.4 - Plate 5**
The Schematics Editor: The use of busses, bus splitting, and interconnection by abutment. Furthermore the verify function is shown, picking out the errors in the circuit. Here the errors are: a loop in a wire, an open end of a wire, and an unknown formal parameter in the expression of an actual parameter.

**Chapter 2.2 - Plate 1**
Layout view of the random counter network used as an example for the hierarchical extractor.
Chapter 3.1 - Plate 1
Routing result example: chip 1

Chapter 5 - Plate 1
ASTRA System: Four views of one cell: stick diagram, mask data, log diagram, floor plan placement.
Chapter 5 - Plate 2
ASTRA System: Floor plan and plot to show where stretching occurred during chip assembly.

Chapter 5 - Plate 3
ASTRA System: Stick diagram of cell showing on-line error checks.
Chapter 5 - Plate 4
ASTRA System: Spaced cell showing constraint information.

Chapter 5 - Plate 5(a)
Chips designed using the ASTRA System:
STAR
Chapter 5 - Plate 5(b)
Chips designed using the ASTRA System:
METEORITE

Chapter 7 - Plate 1
Gate Array Placement.
Chapter 7 - Plate 2
Example of routing in CMOS array with one metal layer, poly underpaths and fixed vias.

Chapter 7 - Plate 3
Example of routing in I^2L array with two metal layers and programmable vias.