Graph-based Representations and Coupled Verification of VLSI Schematics and Layouts

by

Cyrus S. Bamji

S.B. (Mathematics), Massachusetts Institute of Technology (1982)
S.B. (CS), Massachusetts Institute of Technology (1983)
S.M. (EE & CS), Massachusetts Institute of Technology (1985)
E.E. (EE & CS), Massachusetts Institute of Technology (1985)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 1989

© Massachusetts Institute of Technology 1989

Signature of Author ____________________________
Department of Electrical Engineering and Computer Science
September 28, 1989

Certified by ____________________________
Jonathan Allen
Thesis Supervisor

Accepted by ____________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 06 1990
LIBRARIES
ARCHIVES

Graph-based Representations and Coupled Verification of VLSI Schematics and Layouts

by

Cyrus S. Bamji

Submitted to the Department of Electrical Engineering and Computer Science on Septermber 28, 1989, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

Structural verification of VLSI schematics and layouts is formalized. Both schematics and layouts are modeled as graphs and structural correctness is tied to a rigorous set of graph composition rules which define how blocks of schematics and layouts may be composed. Novel, non-heuristic verification techniques which allow structural verification to be performed for a continuum of schematic and layout block sizes are introduced. Using one potent structural verification mechanism, these techniques provide a unified approach to schematic design style verification, layout design rule verification and schematic vs. layout comparison. The verification techniques are fast and can be performed incrementally as the schematics and layouts are created.

For schematic design style verification the composition rules are captured by graph transformations akin to *context free* grammatical productions. The productions describe how a small set of module symbols may be composed. Using these productions a hierarchical parse tree that can demonstrate the correctness of the schematic is constructed. For layouts the composition rules are represented by graph templates. Design rule verification is achieved by covering the layout graph with these templates. Schematic vs. layout correspondence verification is achieved by allowing individual templates to span both schematics and layouts and simultaneously covering the schematic and layout with these templates.

Thesis Supervisor: Jonathan Allen

Professor

Acknowledgments

I would like to thank my advisor, Professor Jonathan Allen, for providing focus to this work while at the same time allowing me considerable lattitude to explore my own ideas which have led this dissertation into some of the uncharted areas of VLSI design. The content and style of this thesis has been greatly influenced by this freedom.

I thank Filip Van Aelten, Robert Armstrong and Bernard Szabo for their help and many useful discussions which have substantially helped clarify my own ideas.

Thank you Mom and Dad, your love and support are still the pillars of my educational accomplishments.

Lastly and above all I wish to say "thank you" to my lovely wife Nagja for her patience and support in the face of my seemingly ever receeding graduation date. Through careful proofreading she has considerably sanitized the writeup of this document and the committed reader will be truly grateful to her for this.

This work has been supported by the Air Force Office of Scientific Research Grant AFSOR 86-0164, IBM and Analog Devices.

Contents

1 Introduction **11**
- 1.1 Overview . . . 11
- 1.2 Thesis Organization . . . 13

2 Schematic Design Style Verification **15**
- 2.1 Introduction . . . 15
- 2.2 Existing Work . . . 16
  - 2.2.1 Simulation . . . 17
  - 2.2.2 Electrical Model based Techniques . . . 17
  - 2.2.3 Electrical Models . . . 17
  - 2.2.4 Pattern Matching Techniques . . . 19
- 2.3 Main Contributions . . . 19
- 2.4 Existing Grammars . . . 21
  - 2.4.1 Overview . . . 21
  - 2.4.2 String Grammars . . . 21
  - 2.4.3 Graph Grammars . . . 26

3 Circuit Grammars **29**
- 3.1 Circuit Representation & Circuit Grammars . . . 29
  - 3.1.1 Circuit Representation . . . 29
  - 3.1.2 Circuit Grammars . . . 33
  - 3.1.3 Network Expansion and Reduction . . . 34
  - 3.1.4 Context Free Circuit Grammar Definition . . . 38
  - 3.1.5 Reducibility Condition . . . 39
  - 3.1.6 Parsing . . . 43
  - 3.1.7 Deterministic Reduction . . . 44
  - 3.1.8 Waveform Generators . . . 50
  - 3.1.9 Net Bundles . . . 51
  - 3.1.10 Equality of Boundary Sets . . . 54
  - 3.1.11 Minimum Number of Pins . . . 55
  - 3.1.12 Net Bundle Definition . . . 56
  - 3.1.13 Examples of Net Bundles . . . 58
  - 3.1.14 Creation of Net Bundles . . . 58
  - 3.1.15 Conditions on Net Bundles . . . 59

3.1.16 The Disjoint Network Problem 59
3.1.17 Reducibility Condition Revisited 62
3.2 Behavioral Verification 63

4 Examples **65**
4.1 A Classical CMOS Grammar 65
4.2 Two Phase Clocking Methodology Grammar 72
4.2.1 Domino Grammar Productions 72
4.2.2 Two Phase Clocking Methodology Grammar Productions 76
4.3 Detecting Common Errors 79
4.3.1 Detecting Open Circuits 80
4.3.2 Detecting Short Circuits 80
4.3.3 Detecting Loops 81

5 Schematic Verification Algorithm & Implementation **85**
5.1 Event Driven Parsing Algorithm 85
5.1.1 Overview 85
5.1.2 Servicing an Event 86
5.1.3 Determining Production Applicability 87
5.1.4 Parsing Complexity 93
5.1.5 Rescheduling due to Absence Conditions 98
5.1.6 Incremental Update 98
5.1.7 Error Reporting 102
5.2 Implementation 103
5.3 Experiments 106

6 Layout Verification **109**
6.1 Introduction 109
6.2 Overview 110
6.3 Layout Correctness 111
6.3.1 Criteria for Layout Correctness 111
6.3.2 Layout Verification using Templates 112
6.4 Review of RSG Connectivity Graphs 112
6.4.1 Cells, Interfaces and Connectivity Graphs 116
6.5 Connectivity Graph based Layout Verification 119
6.5.1 Differences between Layouts and Connectivity Graphs 119
6.5.2 Normalizing the Graph Representation 121
6.5.3 Template Occurrences in Connectivity Graphs 122
6.5.4 Criteria for Connectivity Graph Correctness 124
6.5.5 Dealing with Encoded Cells 124

7 Layout Verification Algorithm & Implementation **127**
7.1 Verification Algorithm 127
7.1.1 Overview 127
7.1.2 Preparing the Graph 129

7.1.3 Finding Template Occurrences 129
7.1.4 Algorithm Complexity 133
7.1.5 Incremental Update 134
7.1.6 Error Reporting 135
7.2 Implementation 135
7.3 Experiments 138

8 Schematic vs. Layout Comparison 139
8.1 Introduction 139
8.1.1 Signature Calculation 140
8.1.2 Path Tracing 141
8.1.3 Rule based Pattern Matching 142
8.2 Shortcomings of Existing Techniques 142
8.2.1 Error Reporting 142
8.2.2 Incremental Comparison 143
8.2.3 Equivalence Flexibility 144
8.3 Benefits of Template based Correspondence Verification 144
8.3.1 Benefits of Operating Directly on the Schematic and Layout Domains 145
8.3.2 Benefits of User Defined Equivalences 146

9 Template based Correspondence Verification 147
9.1 Overview 147
9.2 Correspondence Templates 147
9.2.1 Mappings 148
9.2.2 Regions of Equivalence 150
9.2.3 Correspondence Templates Definition 150
9.2.4 Correspondence Template Occurrence 151
9.2.5 Net Connection Graph 153
9.3 Template based Criteria for Netlist Isomorphism 154
9.3.1 Preliminaries 154
9.3.2 Netlist Isomorphism Criteria 157
9.4 Extensions 159
9.4.1 Equivalence Flexibility 159
9.4.2 Dealing with Bus Instances 164
9.4.3 Dealing with Encoded Layout Cells 167
9.4.4 Schematic vs. Schematic Correspondence Verification 169

10 Correspondence Verification Algorithm & Implementation 170
10.1 Verification Algorithm 170
10.1.1 Overview 170
10.1.2 Preliminaries 171
10.1.3 Servicing an Event 172
10.1.4 Complete Algorithm 175
10.1.5 Algorithm Complexity 176

10.1.6 Incremental Update to the Layout or Schematic 179
10.1.7 Error Reporting 180
10.2 Implementation 181
10.3 Experiments 184
10.3.1 Bit Systolic Multiplier 185
10.3.2 PLA 185

11 Conclusions **187**
11.1 Summary 187
11.2 Future Work 188
11.2.1 Extensions 188
11.2.2 New Directions 190

List of Figures

2-1 Examples of Design Errors 16
2-2 Fragment of RNL file 23
2-3 Equivalent Module and Network 24
2-4 Circuit Represented by the String $N_{1,5,2}P_{1,3,2}I_{2,3,4,5}$ 24
2-5 Two String Representations of the three Inverter Cycle 25
2-6 Three Inverter Cycle 25
2-7 Circuit and Graph Equivalents 27
2-8 Graph with no Corresponding Circuit 27

3-1 Graphical Description 30
3-2 Incorrect Circuit Representation 31
3-3 Isomorphic Circuits 33
3-4 Example of a Circuit Production 34
3-5 Circuit C before Expansion 35
3-6 Latch Production R 36
3-7 Network N_M of C 36
3-8 Expanded Network N_E 37
3-9 Resulting Circuit C' after Expanding C 38
3-10 Circuit with *Illegal Connection* 40
3-11 *Illegally* Reduced Network N_{M_1} 40
3-12 Circuit with Sneakpath Connection 41
3-13 Reducibility Condition 42
3-14 NAND Gate and Inverter Circuit 45
3-15 NAND Gate and Inverter Parse Tree 45
3-16 Presence Condition Modules 47
3-17 Absence Condition 48
3-18 Expanded Absence Module 49
3-19 Expanded Non-absence Condition Module 49
3-20 Waveform Generator Production 51
3-21 Example of Net Bundles 58
3-22 Net Bundle Creation 59
3-23 Disjoint Network Problem 61
3-24 Relations between Nets 62
3-25 Complex Circuit Mapping 62

4-1 Classical CMOS Productions 67
4-2 Classical CMOS Grammar 68
4-3 Transistors in Different CMOS Gates 69
4-4 CMOS Presence Condition Modules 70
4-5 Parse Sequence 71
4-6 Production 1 73
4-7 Production 2 73
4-8 Production 3 74
4-9 Production 4 75
4-10 Production 5 75
4-11 Production 6 76
4-12 Non Series Parallel Domino Blocks 77
4-13 $\phi_i$section Block Diagram 77
4-14 LSB Production 78
4-15 Complex $\phi_i$section Production 78
4-16 ϕsection Production 78
4-17 Start Symbol Production 79
4-18 Loop Checking Production P_{loop} 82
4-19 Parallel LSB Composition 82
4-20 Unbalanced Parse Tree 84

5-1 Positions of Modules C_{blk} 88
5-2 Network and Corresponding Instruction 90
5-3 Module and Net Slots 91
5-4 Network and Instructions for Superior and Inferior Nets 92
5-5 Procedure Execute Instruction 93
5-6 Efficient and Inefficient Instructions 96
5-7 Augmented Parse Tree 99
5-8 Six Transistor XOR Gate 101
5-9 Example of an Error Production 103
5-10 Textual Representation of a Production 105
5-11 Circuit Input Netlist 106
5-12 Xwindow Graphic Display 107
5-13 Systolic Multiplier 107

6-1 PLA Layout 113
6-2 PLA Instances 114
6-3 Examples of Templates 115
6-4 Instance of Cell B in Cell A. 116
6-5 Interface between A and B 117
6-6 Graph and Layout Equivalents 118
6-7 Cycles in the Graph 120
6-8 Equivalent Graphs 121
6-9 Graph Representation for Templates 122
6-10 Template Occurrence 123

6-11 Encoded Cell Templates 126

7-1 Components of the Algorithm 128
7-2 Graph and Corresponding Instructions 131
7-3 Procedure **execute_vertex_instruction** 132
7-4 Textual Representation of a Connectivity Graph 136
7-5 Xwindows Graph Display 137

9-1 Mapped Module and Vertex 149
9-2 Mapped Schematic and Layout Regions 149
9-3 Simple Correspondence Template 151
9-4 Correspondence Template 152
9-5 Net Connection Graph 153
9-6 Equivalence Templates 159
9-7 Equivalence Criterion 160
9-8 Full-adder Cell Implementations 162
9-9 Bus Cells 166
9-10 Correspondence Templates for Bus Cells 168

10-1 Mapping Validation 173
10-2 Inverter Template 176
10-3 Verification Example 177
10-4 Textual Representation of a Correspondence Template 182
10-5 Schematic Input Netlist 183
10-6 Layout Input Graph 183
10-7 PLA Path 186

11-1 Program Inputs and Outputs 189

List of Tables

5-1 Parsing Complexity Summary 97
5-2 Parse Times 106

7-1 Verification Times 138

10-1 Multiplier Correspondence Verification Times 185
10-2 PLA Correspondence Verification Times 186

1

Introduction

1.1 Overview

The design and validation of VLSI circuits having millions of components represents a challenge to both the human designer and the computer-aided design tools. To reduce design time and cost, it is crucial that errors be caught early in the design. Due to the size of today's circuits, the process of identifying design errors can be reliably performed only through the use of computer aids. These verification tools sift through the design, locate errors and report them back to the designer.

One of the tests that can be applied to a design to find errors is structural verification. Structural verification is the process of validating the structure or arrangements of objects in the design. The behavior of the design and design objects is not considered. Rules are used to specify how a collection of design objects can connect together regardless of their functionality. Design errors are found by verifying that the way in which the objects are connected in the design satisfies these rules. Alternatively, the structure of the design may be compared with the structure of another design known to be correct. Inconsistencies between the two structures are reported to the designer as errors.

Structural verification is used in three major areas.

1. Electrical Rules Checking (ERC for short) in which a circuit schematic is checked for errors that arise from incorrect electrical connections such as short circuits, illegal signal loops etc.

2. Layout Design Rule Checking (DRC for short) in which a set of mask patterns to be transferred to a silicon wafer are checked to see if they belong to a set of permissible mask geometries.

3. Connectivity Verification (CV for short) in which a netlist extracted from a layout mask description is compared with another netlist, usually extracted from a schematic.

Existing structural verification techniques use radically different computer models and verification strategies in each of these three major areas. In this dissertation, a single potent technique with a unified view of all three areas of structural verification and with substantial advantages over conventional techniques in each of these areas is presented.

Both schematics and layouts are modeled as graphs. The vertices in these graphs represent modules in the schematic and cell instances in the layout. The modules represented by vertices can be single transistor devices or large aggregates of many transistors. Similarly, the vertices for cell instances can represent single polygons or complex cell instances of hundreds of polygons. This allows the techniques proposed in this thesis to be applicable over a continuum of module and cell sizes enabling the designer to choose between *fine grained* or *coarse grained* verification.

The rules which govern structural correctness are defined by graph transformations on the schematic and layout graphs. For schematics, these transformations are captured by user defined graph-grammar productions and verification is performed by *parsing* the schematic using these grammatical productions. For layouts, correctness is captured by graph templates which define small fragments of layout which are known to be structurally correct. Layout verification is performed by *covering* the layout graph with these templates.

A more general form of connectivity verification, referred to in this thesis as Schematic vs. Layout correspondence verification, is made possible by allowing graph templates to simultaneously span both schematics and layouts. These templates, called correspondence templates, consist of two graphs; one of a small region of layout and the other of a small region of schematic whose netlists are equivalent. By simultaneously covering both the layout and schematic graphs with correspondence templates the equivalence between their netlists can be verified.

The major contributions of this dissertation are:

- A new set of representations and formalisms for both schematics and layouts which cleanly and precisely capture the structural design constraints.

- Fast and incremental non-heuristic verification techniques, both novel and radically different from traditional techniques, with one basic verification method for all three structural verification areas (ERC, DRC and CV).

- Evidence that practical designs and methodologies can be effectively verified within this framework.

For each of the major structural verification areas described above, a detailed introduction as well as an overview of existing verification techniques is provided in subsequent chapters. Benefits of the proposed methods over existing techniques are also expanded upon in these chapters.

1.2 Thesis Organization

This thesis is organized into three major parts which correspond to the three major areas of structural verification. In the first part, circuit grammars are used to perform schematic design-style verification (a form of ERC). In the second part, layout graph templates are used to perform layout verification (a form of DRC). Finally in the third part, correspondence templates are used to perform schematic vs. layout correspondence verification (a more general case of CV). The formalisms and techniques in part three build on those developed in parts one and two. Part three also assumes that the graph for the layout has been successfully verified by the verification technique in part two. Finally, the efficiency and flexibility of part three can be enhanced by using the grammatical reduction capabilities of part one.

Each major part of this thesis has a similar structure. First, the associated major area of structural verification is introduced and existing work is summarized. Secondly, the techniques proposed in this thesis are presented and their advantages over existing verification techniques are listed. Next, precise mathematical models for the graph representations are provided and conditions which guarantee structural correctness are proved. Then an algorithm capable of verifying that these structural correctness conditions are met in a given schematic and/or layout is defined. Each part concludes with experiments in which the verification methods are applied to examples of schematics and layouts using a computer program which implements the verification algorithm.

A chapter by chapter breakdown of this document is given below:

Chapter 2 first describes the problem of schematic design style verification, summarizes existing verification techniques and lists the main contributions of the schematic verification method proposed in this thesis. The second part of this chapter introduces string grammars and explains why they are inadequate for schematic design style verification.

Chapter 3 introduces a new kind of grammar called circuit grammars which are specifically tailored for performing schematic design style verification. Precise definitions for

schematics and circuit grammars are given and conditions based on grammatical parsing which ensure schematic structural correctness are derived and proved.

`Chapter 4` gives examples of circuit grammars for various design styles and how grammars catch various design errors.

`Chapter 5` describes the event driven parsing algorithm which implements the verification methods of chapter 3. This chapter concludes with a description of a computer implementation of this algorithm called GRASP (for **Gra**mar-based **S**chematic **P**arser) and experiments using GRASP to verify a large bit systolic multiplier.

`Chapter 6` introduces layout design rule verification and presents a method for verifying the design rule correctness of layouts constructed from instances of library cells. Layouts are modeled as graphs and conditions for layout design rule correctness based on layout graph templates are derived and proved.

`Chapter 7` describes the event driven layout verification algorithm which implements the verification techniques of chapter 6. A computer program of this algorithm called GLOVE (for **G**raph-based **L**ay**o**ut **Ve**rifier) and experiments using glove to verify a large PLA are then presented.

`Chapter 8` describes schematic vs. layout correspondence verification and summarizes existing correspondence verification techniques. A template based correspondence verification technique capable of verifying the correspondence between a schematic and a layout built from instances of library cells is then introduced and it's advantages over existing techniques are highlighted.

`Chapter 9` builds upon the formalisms of chapters 3 and 6 to derive and prove conditions on schematics and layouts based on correspondence template graphs which guarantee that the schematic and layout have equivalent netlists.

`Chapter 10` describes the event driven correspondence algorithm which implements the correspondence verification methods of the previous chapter. This chapter ends with a description of a computer program for this algorithm called SCHEMILAR (for **Schem**atic vs. **La**yout Compa**r**ator) and experiments using SCHEMILAR to verify a large multiplier and PLA.

`Chapter 11` summarizes the work presented in this thesis, shows how the parts of this thesis interact and concludes with future directions.

2

Schematic Design Style Verification

2.1 Introduction

A circuit schematic is a specification of how a set of electronic components, called modules, are electrically connected together. One of the intermediate steps required for carrying a VLSI design from concept to implementation is building a schematic for the design. Schematic verification is the process of verifying that a circuit schematic obeys a certain set of design constraints. In some cases the schematic is generated mechanically in a *correct by construction* manner. This is accomplished by a sequence of transformations applied to a functional specification of the design. Often however, the schematic is not generated in this correct by construction manner and therefore it becomes necessary to perform schematic verification.

Given present levels of integration, it is no longer possible for a human circuit designer to manually perform schematic verification. In this chapter, an effective method for automating this task is proposed. Many faulty schematics can be weeded out quickly because they violate some simple design criteria such as a short circuit, an illegal signal loop path etc. To facilitate the design as well as the validation of the design, schematics are made to conform to design methodologies (e.g., classical CMOS, ratioed NMOS, domino logic) which impose restrictions on circuits deemed acceptable. These design styles specify how the modules in the circuit can be connected together so that the schematic is *well-formed.*

By demanding that a candidate circuit conform to a design methodology, it becomes possible to establish a *first line of defense* guarding against design errors, irrespective of circuit functionality. For example, the schematic of figure 2-1 has two errors: the output of the bottom inverter is connected to the output of a precharged gate, and because of

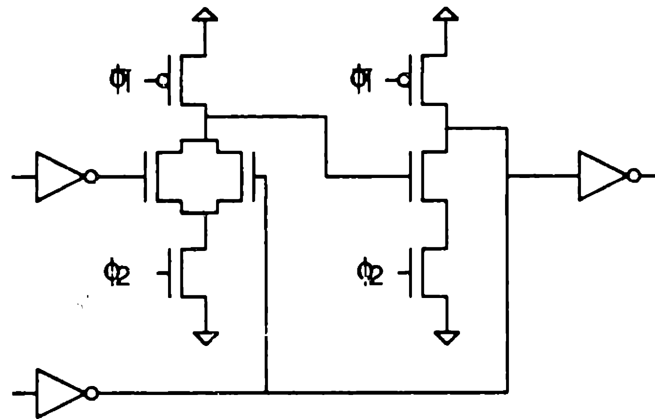


FIGURE 2-1: Examples of Design Errors

rising and falling signal edge considerations, the output of the leftmost precharged gate must go through a static inverter before it can feed the rightmost gate. Both errors can be detected without any knowledge about the circuit's functionality. This form of verification is called design style or design methodology verification and is the object of this chapter.

After the circuit's membership in a design methodology has been established, a functional check of the circuit [1], [52] remains to be performed by some other more complex and slower means. In section 3.2, a method related to the techniques described in this chapter is used to perform functional verification.

In the first part of this thesis a formal technique is proposed that is capable of verifying that an electrical circuit is well-formed by ensuring that the circuit conforms to a circuit design methodology. The technique has been implemented in a computer program called GRASP [9] (Grammar-based Schematic Parser). GRASP incorporates several novel techniques and formalisms which allow a clean capture of a circuit design methodology and very fast verification speeds. The use of context free circuit grammars to describe a circuit design methodology coupled with GRASP's event driven verification algorithm results in a technique that is one to two orders of magnitude faster than previous heuristic approaches. GRASP's algorithm is also incremental and can verify the circuit as it is being edited and modified by the circuit designer.

2.2 Existing Work

Well-formedness verification techniques can be split into two basic categories. In the first technique a set of electrical constraints that all well-formed circuits must satisfy is derived. In digital design these constraints are typically requirements that guarantee

that the voltages on the inputs and outputs of logic gates, latches etc. can be abstracted by 0 and 1 values. Knowledge of these principles is built into the system.

In the second technique the syntax of the schematic is examined. Portions of the schematic are matched against patterns of interconnected modules provided by the user. Electrical knowledge about correct circuit configurations is embedded in the patterns. They are known to have certain specific electrical properties and their presence or absence in the schematic determines whether the schematic is well-formed. The verification system manipulates the patterns but the underlying electrical meaning of the patterns is not known to the verification system.

The schematic verification technique in this thesis falls in the latter of these two categories. It differs significantly from existing techniques. Instead of an ad hoc collection of illegal module configurations that must not occur in the schematic, the patterns are used to define a context free circuit grammar which precisely defines a set of well-formed circuits.

2.2.1 Simulation

Simulation [13], [48], [51] is sometimes used to find schematic design errors. Simulation differs from design style verification in two major respects.

1. Simulation is not independent of circuit functionality. The functionality of the circuit must be known in order to interpret the results of the simulation.

2. Simulation shows errors rather than looking for them. In order to demonstrate the error via simulation, an appropriate set of inputs needs to be applied. The task of determining this set of inputs lies with the user.

2.2.2 Electrical Model based Techniques

2.2.3 Electrical Models

Techniques such as [12], [52], [14], [26], have an electrical model for correct schematics. The models impose conditions on the electrical properties of nets and conditions on the electrical and signal paths between the nets. For example, to avoid problems associated with charge sharing there are constraints on the capacitances of each net. Examples of constraints on electrical and signal paths between nets would be that electrical short circuit paths between **vdd** and **gnd** are to be avoided as are signal paths through an odd number of inverters.

These techniques are based on fundamental electrical principles common to many different design styles and technologies. The correctness of a wide range of designs can be verified by a succinct set of electrical principles and the electrical characteristics of circuits that follow them can be accurately characterized.

It is important to note that these techniques use simplified models of the underlying electrical devices. For example, transistors are modeled as resistors, capacitances are linear and are always to ground etc. Each model exhibits a different set of tradeoffs between completeness and verification efficiency. The models try to be conservative so that incorrect circuits are not reported as correct. In certain cases however some electrical phenomena may not be accurately captured and the model breaks down. The changes to the model required to accommodate these cases may be substantial and hence impractical to implement. Given that verification knowledge is embedded within the system, augmenting the system to handle these special cases is difficult and awkward to incorporate within the same framework.

Verification Methods

Once the electrical constraints that characterize well-formed schematics are defined, it remains to create an algorithm capable of deciding whether a given schematic meets these constraints.

Verifying that the electrical constraints are met at each net may require that a large number of different possible electrical paths from that net need to be examined. Different paths can be formed from a net depending on which transistors are conducting. For example, in order to verify that the output of a logic gate is not simultaneously pulled high and low a variety of different input combinations to the gate may have to be considered. The number of possible configurations that have to be considered may get large and hence these methods are computationally expensive.

To reduce verification time [52][1] exploits hierarchy already present in the schematic. In [12], heuristics which recognize certain patterns that are known to be well-formed are used to increase verification speeds for commonly used subcircuits.

In [26], subcircuits are represented by matrices representing the corresponding switch graph. Conditions on these matrices equivalent to the electrical conditions on the nets are derived. Matrix manipulation techniques are used to verify that the conditions on the matrices are met.

[1] [52] is principally concerned with behavioral verification but well-formedness verification is performed as a prerequisite to behavioral verification.

2.2.4 Pattern Matching Techniques

Techniques such as [31], [27] and [40] rely on user supplied rules to verify the schematic. These rules contain patterns of connected transistors which when encountered in the schematic, trigger some action by the verification system.

The rules capture electrical constraints similar to those described in section 2.2.2. The patterns of transistors in the rules represent configurations which either satisfy or violate the electrical requirements. Since the verification knowledge is contained in the rules and not in the verification program itself, the verification strategy and the electrical constraints underlying the rules can be changed without modifying the program.

An expert system with a rule language suitable for describing circuit design styles is described in [31]. In the database technique of [27], attributes are first computed for each net. These attributes as well as the patterns of transistors surrounding them are matched with configurations and conditions on nets stored in a data base. The program described in [40] is an expert system where the rules describe sets of illegal configurations that must be avoided in well-formed schematics.

2.3 Main Contributions

The main contributions of this work are to:

1. Cast the problem of circuit design methodology verification into that of parsing a circuit network in accordance with a network grammar. The grammar is a specification of the *range space* of the methodology. As a result, a clean and precise description of circuit correctness is captured by the grammar specification. The use of grammars is made possible by introducing the concept of net bundles, described in section 3.1.9, which allows packets of nets in the circuit to be combined and dealt with as one object. GRASP is inspired by graph grammar theory [19] which is modified and augmented to deal with practical circuits.

2. Show that practical circuit methodologies can be described by grammars which can be efficiently parsed. GRASP's efficient, incremental and hierarchical parsing algorithm allows rapid verification of any circuit in the *range space* of the grammar. There are no false positives or false negatives. The algorithm allows addition and deletion of modules even after the circuit has been fully parsed. Errors such as shorts, ill-formed modules, rising and falling edge violations, illegal loops, or any error that would cause a circuit not to be in the *range space* of the grammar, can be caught by the GRASP verifier.

> In the same way that a programming language parser (such as that contained in the front end of a compiler) reads a program source file and checks for syntax errors in the program while building a parse tree, GRASP reads a circuit netlist file and builds a circuit parse tree in accordance with a user specified circuit grammar.

A circuit methodology grammar is described by a set of circuit grammar productions specified in a Lisp-like syntax. The grammatical productions (also called *grammar rules*) used in GRASP are very different from heuristic production rules [31], [40] or database techniques [27]. The grammar is simply a compact hierarchical specification of the set of circuits that conform to the methodology. The use of heuristic production rules by contrast, is more a programming style that describes the action to be taken if a certain set of conditions holds true. The entire operation of the GRASP verifier is restricted to replacing subcircuits by modules. This operation is called parsing. The process is guaranteed to succeed if the circuit is in the *range space* of the grammar and to fail otherwise. Since the entire checking process is performed using this one potent operation, algorithmic speed, tractability and simplicity is achieved.

Unlike the techniques described in [26], [27], [31] and [40] which search for illegal circuit configurations, GRASP uses a specification of the methodology itself in a fast, non-heuristic and incremental verification technique which identifies syntactically correct structures.

The class of circuit grammars that can be handled by GRASP's parsing algorithm belongs to a subset of deterministic context free grammars. These grammars are sufficiently restricted in structure to be parsed efficiently. The next major class of grammars beyond context free grammars is the class of context sensitive grammars [25]. This class is much too unstructured and unwieldy for efficient parsing techniques to be applicable.

The remainder of this chapter introduces the reader to grammars. Existing grammars, namely string grammars, are described and the reason for their unsuitability for verifying circuits is explained.

Section 3.1 introduces a special kind of grammar called circuit grammars which is used to precisely define the set of schematics that obey a design methodology. Using a technique called net bundling the problem of design style verification is then cast into grammatical parsing using a circuit grammar. The formalisms of this chapter are extended to incorporate behavioral verification as described in [1].

In chapter 4 a deterministic context free grammar for the common CMOS two phase clocking methodology [53] is described. With this grammar, GRASP can verify whether a transistor level description of a circuit obeys the CMOS two phase clocking methodology. This methodology consists of classical CMOS gates, dynamic gates (e.g. domino) and

latches combined in accordance with the two phase clocking requirements. This section concludes with an example of a parse on a CMOS static gate.

Chapter 5.1 describes the event-driven parsing algorithm used in GRASP and its capability of allowing modules to be incrementally added or deleted from the circuit even after the circuit has been parsed. The chapter also provides experimental timing results obtained by applying GRASP to a large circuit, namely a bit systolic retimed multiplier.

2.4 Existing Grammars

2.4.1 Overview

Grammars specify how a set of objects called the alphabet of the grammar may be connected together. The purpose of this section is to familiarize the reader with some of the terminology and formalisms of grammars and to show that the types of grammars used in programming languages, namely string grammars, are not suitable for design methodology verification. The discussion of this section is informal. In chapter 3 a precise definition is provided for some of the terms informally discussed in this section.

In this section string and graph grammars are described. String grammars are shown to be inadequate for describing circuit methodologies. The shortcomings of string grammars lie in the fact that strings are inadequate representations for circuits. The class of string grammars necessary to encode useful circuit methodologies is too general to be effectively handled by a grammar based methodology verification algorithm. Graphs can be used to effectively represent circuits, however as shown in section 2.4.3, for some graphs there is no corresponding circuit equivalent. A representation specifically tailored for circuits and a new kind of grammar called circuit grammars that operates directly on circuit representations is introduced in chapter 3.1.

Section 2.4.2 introduces string grammars and the basic formalisms common to all grammars. The shortcomings inherent in string grammars that render them useless for representing circuits are then described. Section 2.4.3 briefly introduces graph grammars, gives references for their definition and uses and also sets the stage for the *graph-like* circuit representation of chapter 3.1.

2.4.2 String Grammars

String grammars are widely used in Computer Science and are at the heart of the Theory of Computation [29], [24]. Only certain classes of string grammars can be readily

used for syntax verification. The most widely used are *deterministic context free* grammars. Syntax verification using these grammars is called parsing. The computer program that accomplishes this is called a parser. These grammars can capture most hierarchical programming language constructs and efficient parsers for them can be built.

During the design of a compiler for a programming language such as the C programming language, a grammar specification of the C language is first generated. A parser for the C language is then derived from the grammar. Programs such as YACC [4] can automatically generate a parser from a grammar. Given a specification of the grammar for a language L, YACC generates C language source code which when compiled acts as a parser for the language L.

String Representation of Circuits

A *string* is a sequence of *string symbols* juxtaposed. For example, if x, y and z are symbols xyz is a string. Circuits can be described by strings. In fact the input to most circuit simulators [51], [48], [13] is a file containing a string (text) description of the circuit. One of the simplest formats for such a description is the format used in the RNL [48] simulator. Each line of this type of file begins with the name of a module type followed by net numbers. A line such as:

```
ntrans 3 2 5
```

signifies that there is a module of type `ntrans` connected to nets 3 2 and 5. The ith net in the list connects to the ith pin of the module. By convention, pins 0, 1 and 2 of an ntrans type module refer to the *gate, source* and *drain* of the n-channel transistor. Similar conventions are used for every other module type.

Since circuits can be encoded by strings, it is theoretically possible to encode the circuit to be verified by a string and use a string grammar to verify the circuit. This strategy is, however, not practical as is explained in the following subsections. First, string grammars which are compact formal descriptions of a finite or infinite set of strings are introduced. This introduction to grammars will also familiarize the reader with grammars before circuit grammars are introduced in section 3.1.2.

String Grammar Definition

A context free string grammar G (CFSG for short) is denoted by $G = (N, T, P, S)$ where N and T are finite sets of string symbols [24] (called *nonterminals* and *terminals* respectively) with $N \cap T = \emptyset$. P is a finite set of productions of the form $A \rightarrow x$ where $A \in N$ and x is a string of symbols in $N \cup T$. S is a distinguished symbol in N called

ntrans	1 5 2
ptrans	1 3 2
inverter	2 3 4 5

FIGURE 2-2: Fragment of RNL file

the *start symbol*. The relation $\Rightarrow$ between strings is defined as follows: If $A \rightarrow x$ is a production in P and α and β are two strings then $\alpha A\beta \Rightarrow \alpha x\beta$. The relation $\stackrel{*}{\Rightarrow}$ is the transitive closure of $\Rightarrow$ defined by: for any strings α, β, γ $\alpha \stackrel{*}{\Rightarrow} \alpha$ and if $\alpha \stackrel{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$ then $\alpha \stackrel{*}{\Rightarrow} \gamma$. The set of strings that obey the grammar G is called the *language of* G denoted by L_G and is defined by $\alpha \in \mathrm{L}_G$ if and only if $S \stackrel{*}{\Rightarrow} \alpha$ and α contains only symbols in T. Any set of strings that is the language of some CFSG is called a context free language.

Context free attribute string grammars [23] (CFASG for short) are a variant of CFSG, more convenient than CFSG for describing circuits. A variant of CFASG grammars are used in this section. It will be shown in section 2.4.2 that even this more powerful form of grammars is inadequate for circuit verification due to the inherent weaknesses of string grammars for adequately representing circuits.

A CFASG $G = (N, T, P, S)$ is similar to a CFSG except that the symbols in $N \cup T$ have attributes. For example, symbol A might have attributes x, y, z and this is denoted by $A_{x,y,z}$. The productions in P are of the form $A_{x,y,z} \rightarrow \gamma$ where γ is a string of attributed symbols in $N \cup T$ and x, y, z are functions of the attributes of the symbols in the string γ. For the purposes of this section it will be assumed that the attributes are integers which represent net numbers.

Circuits are readily expressed by a string of attributed symbols. Each symbol represents a module of a certain type and it's attributes represent the nets it is connected to. For example, if N, P and I are symbols for n-channel transistor, p-channel transistor and inverter respectively, then the RNL circuit file in figure 2-2 can be expressed by $N_{1,5,2}P_{1,3,2}I_{2,3,4,5}$. The underlying meaning of a production of the form $I_{x,y,z,t} \rightarrow N_{x,t,z}P_{x,y,z}$ is that an n-channel and a p-channel transistor connected as in $N_{x,t,z}P_{x,y,z}$ (pictorially represented by figure 2-3 (a)) is *equivalent to* and can be replaced by an inverter connected as in $I_{x,y,z,t}$ (pictorially represented by figure 2-3 (b)). The language of the grammar represents the set of all circuits expressed by strings that can be derived by expanding the start symbol S.

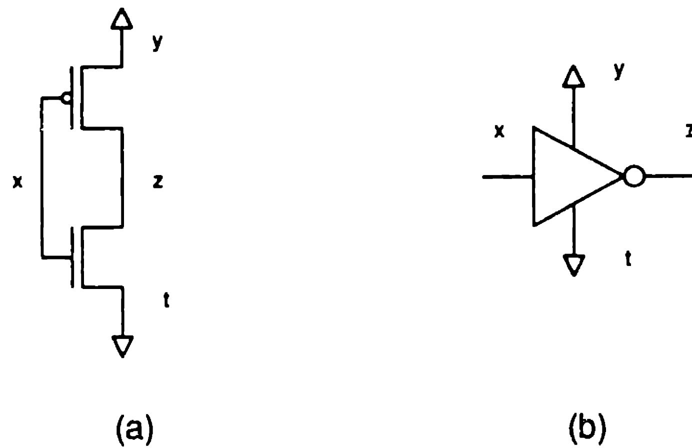


FIGURE 2-3: Equivalent Module and Network

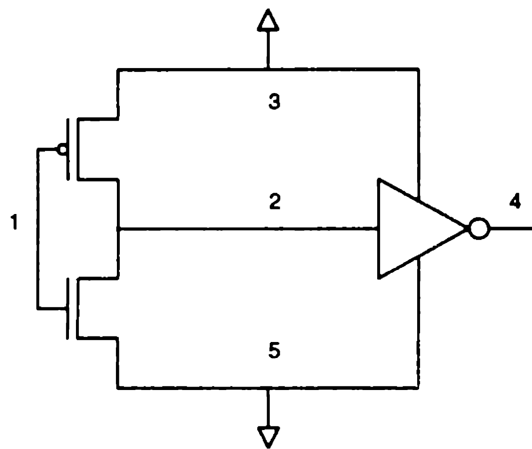


FIGURE 2-4: Circuit Represented by the String $N_{1,5,2}P_{1,3,2}I_{2,3,4,5}$

Problems with String Grammars

The problem with string representations of a circuit is that the order of the symbols as they appear in the string is not relevant. Changing the order of the symbols in the string does not change the underlying circuit represented by the string. For example, the circuit represented by the string $N_{1,5,2}P_{1,3,2}I_{2,3,4,5}$ is the same circuit as those represented by $I_{2,3,4,5}P_{1,3,2}N_{1,5,2}$, $P_{1,3,2}I_{2,3,4,5}N_{1,5,2}$ or $I_{2,3,4,5}N_{1,5,2}P_{1,3,2}$. All four strings represent the circuit of figure 2-4. If grammar G is to be useful for verifying circuits then if s is a string in L_G any string s' obtained by permuting the symbols in s must also be in L_G. CFASGs and CFSGs are unfortunately sensitive to the order and location of symbols in the string. This incompatibility makes string grammars unsuitable for verifying circuits as is shown by an example in the next paragraph.

Let G be a grammar whose language represents CMOS transistor circuits which form cycles of inverters. Without loss of generality it is assumed that the only production in G which manipulates the N and P symbols is the $I_{x,y,z,t} \rightarrow N_{x,t,z}P_{x,y,z}$ production. In any

$$P_{1,4,2}N_{1,5,2}P_{2,4,3}N_{2,5,3}P_{3,4,1}N_{3,5,1}$$
(a)

$$P_{1,4,2}P_{3,4,1}P_{2,4,3}N_{2,5,3}N_{1,5,2}N_{3,5,1}$$
(b)

FIGURE 2-5: Two String Representations of the three Inverter Cycle

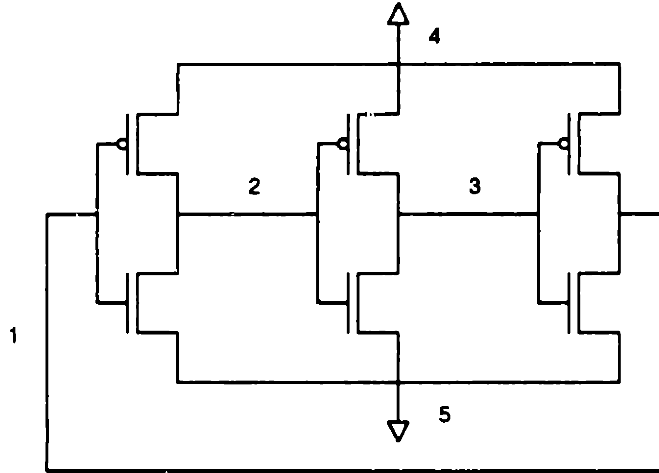


FIGURE 2-6: Three Inverter Cycle

string s in L_G, symbols derived from the application of a given production will appear close to each other in the string s. Hence in any string s in L_G, transistor symbols N and P derived from the same inverter symbol I will appear at consecutive locations.

Figure 2-5 (a) shows one possible string s in the grammar. The string s represents the three inverter cycle circuit shown in figure 2-6. Figure 2-5 (b) shows another string representation s' of the same circuit. The 2nd and 5th transistor symbols in s' belong to the same inverter but appear at non consecutive locations in the string and therefore s' cannot be in L_G.

in order to verify the circuit represented by the string s' using grammar G, the order of the symbols in s' must be rearranged so that the N and P symbols derived from the same inverter appear at consecutive locations. It is unfortunately not always possible to rearrange the string so that neighboring modules in the circuit appear at consecutive locations in the string encoding of the circuit. For example, the power supply module connects to a large number of modules all of which cannot be adjacent to the supply module in a string representation of the circuit. In fact the *appropriate* ordering is not intrinsic to the circuit but depends also on the grammar. In general, finding the *appropriate* ordering of symbols in a string s in order for a grammar G to be able to

parse it is a difficult problem for which no practical solution exists.

Constructing a grammar whose language contains all symbol permutations of strings in G is not a practical solution. Given a CFASG G, let $G' = Per(G)$ be a grammar such that for any string s in L_G any string s' obtained by permutation of the symbols in s is in $L_{G'}$. For a string grammar G, such as the ring inverter grammar described above, the language of $G' = Per(G)$ consists of all strings representing transistor networks of inverter rings regardless of the order of the symbols in the string. Not only is $N_{1,2,4}P_{1,3,4}I_{4,3,2,5}$ in $L_{G'}$ but so are: $I_{4,3,2,5}P_{1,3,4}N_{1,2,4}$, $P_{1,3,4}I_{4,3,2,5}N_{1,2,4}$ and $I_{4,3,2,5}N_{1,2,4}P_{1,3,4}$.

In general G' is not a CFASG and belongs to a more general class of string grammars for which the verification techniques required are much more complex making circuit verification using G' impractical. For example, it can be shown that the string language $L_G = \{a^n b^n c^p d^p\}$ is context free but using the pumping lemma for context free string grammars[22] it can be shown that for $G' = Per(G)$, $L_{G'}$ is not context free.

Because attribute string representations of circuits are not sensitive to the order of the symbols in the string and since CFSGs (and CFASGs) lack the ability to deal effectively with symbol permutation within the string, string grammars are unsuitable for circuit design style verification.

2.4.3 Graph Grammars

The objects manipulated in graph grammars are the graphs themselves and as such graph grammars (GGs for short) do not suffer from the *string permutation* problems inherent in string grammar representations of graphs. Work has been done on graph grammars particularly as they relate to biology and computer science [19]. Various forms of graph grammars and a description of their applications can be found in [20] and [19]. [34] is an extensive list of references for various sorts of graphs and their applications. Many of the graphs and graph grammars in [20], [19] and [34] are tailored for a specific application.

Circuits can be represented by graphs with two kinds of vertices: module vertices and net vertices. The module vertices connect to the net vertices via the graph edges. The labels on the edges represent the pin types of the various modules. Figure 2-7(b) shows the equivalent graph representation of the circuit of figure 2-7(a). In this representation not all graphs can represent circuits. For example, the graph of figure 2-8 cannot represent a circuit because transistor modules have only one gate connection.

In the next chapter a representation for circuits based on graphs similar to the representation of the previous paragraph is described. An associated grammar type called

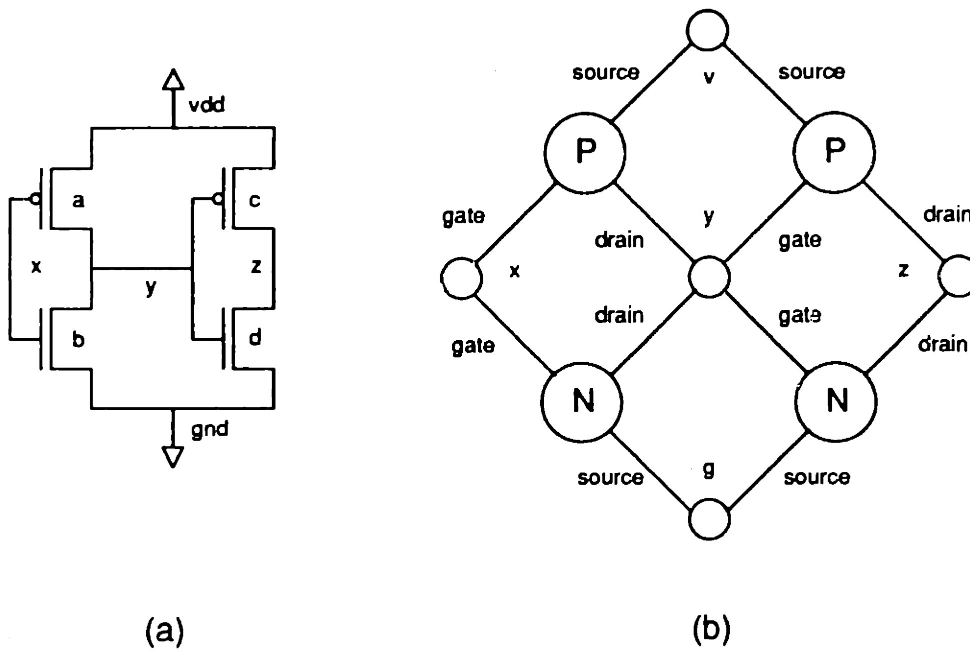


FIGURE 2-7: Circuit and Graph Equivalents

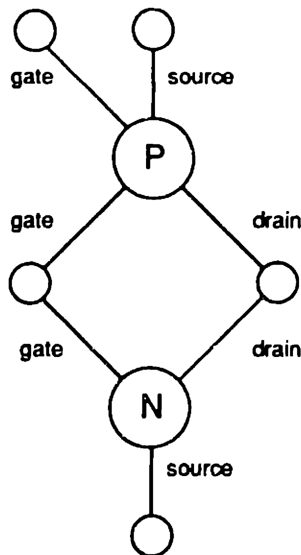


FIGURE 2-8: Graph with no Corresponding Circuit

circuit grammar specifically tailored for the new representation is introduced. These grammars are inspired by the graph grammars of [20] and [19]. The representations for circuits of chapter 3.1 are closer to the usual representations for circuits. The vocabulary used to describe this new representation is derived from circuits rather than from graphs thus making their discussion easier. Also some of the problems related to the fact that there may be instances of the representation for which there is no corresponding circuit (such as the graph of figure 2-8) are not present in this new representation.

3

Circuit Grammars

3.1 Circuit Representation & Circuit Grammars

The problem with string and graph grammars is that strings and graphs are inadequate representations for circuits. Because of the shortcomings of string and graph grammar for effectively handling circuit methodologies a new kind of grammar called circuit grammar with operates on circuits is introduced.

In this chapter first a suitable representation for circuits is described. Then a new kind of grammar which preserves the spirit of string and circuit grammars and directly manipulates the circuit representation is introduced. Just as string and graph grammars describe sets of strings and graphs, circuit grammars describe sets of circuits. In section 3.1.9, *net-bundles*, a necessary ingredient for casting the problem of design style verification into grammatical parsing is described. In that section the benefits of having the grammar directly manipulate the representation will become clear. Each section first gives an intuitive feel for the issues involved and then introduces the necessary formalisms for precise definitions.

3.1.1 Circuit Representation

In this section a representation for circuits is described. The representation closely parallels our intuitive understanding of what a circuit netlist is. It consists of a list of modules and a description of how the modules are electrically related. Each module is an instance of a module type. Modules have various points called pins at which electrical connections can be established. Describing how the modules electrically relate is accomplished by defining which pins are electrically connected together. The underlying

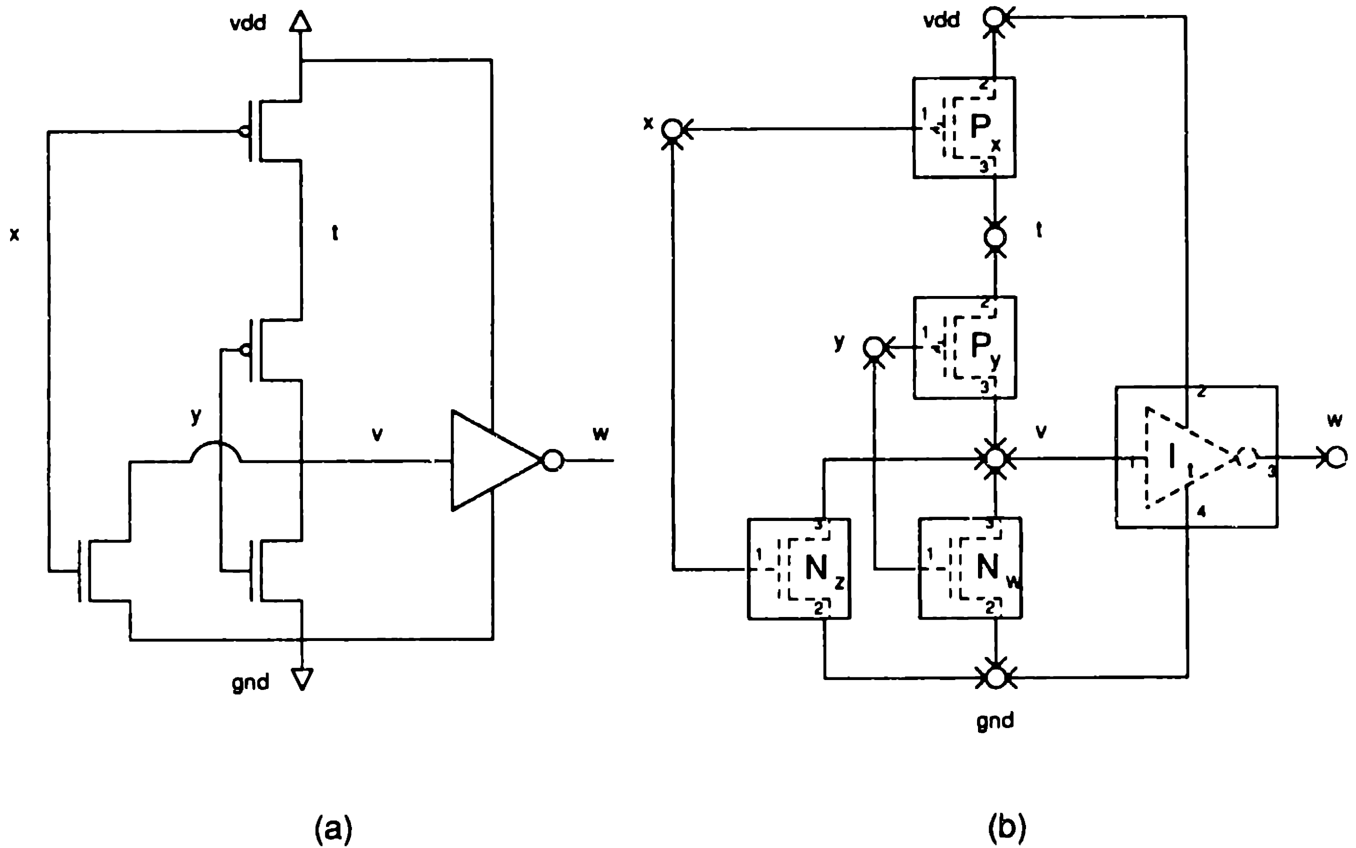


FIGURE 3-1: Graphical Description

electrical meaning of the connections is that all the pins connected together are at the same voltage and they satisfy KCL in that the sum of the currents flowing into them is zero.

Figure 3-1 (a) shows the traditional pictorial representation for a circuit consisting of a CMOS NOR gate and an inverter. The modules are drawn using different symbols for each module type and the pins are identified by distinguished locations on each symbol. Nets are drawn by wires *connecting* the pins. A more suitable pictorial representation of the circuit for our purposes is shown in figure 3-1 (b). Both nets and pins appear explicitly in figure 3-1 (b) as do the connections between them. The nets appear as small circles. The pins are denoted by a symbol, usually a number, inside the parent module's boundary. An arc between a pin and a net signifies that the pin is connected to the net.

Each net must be connected to at least one pin and each pin must be connected to exactly one net. Hence figure 3-2 does not depict a valid circuit because net i does not connect to any pins and pin 3 of the inverter module does not connect to a net.

Formal definitions of pins, nets, connections, module types and modules specifically suitable and relevant to grammar based verification are described in the remainder of this section.

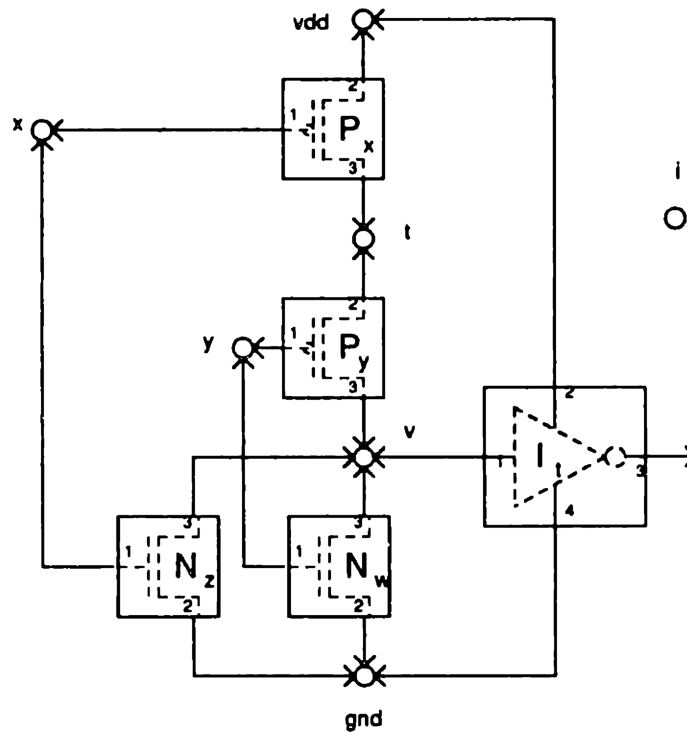


FIGURE 3-2: Incorrect Circuit Representation

Nets, Pins and Module-types

Nets, pins and module types are the atomic objects from which modules, circuits and circuit grammars are built. Each module type T has an attribute which is called *numberofpins*. Nets are drawn by small circles as shown in figure 3-1. The identifiers next to them are unique names given to the nets for explanatory purposes.

Definition 1 *Let $\mathcal{P}$, $\mathcal{N}$, $\mathcal{T}$ be three* **distinct** *sets whose elements are called* **pins, nets** *and* **module types** *respectively. Let numberofpins denote a function from $\mathcal{T}$ to $\aleph$. For $T \in \mathcal{T}$ numberofpins(T) is denoted by σ_T.*

Connections between pins and nets drawn by arcs in figure 3-1 (b) are associations between pins and nets.

Definition 2 *A* **connection** *is a pair (p, η) where $\eta \in \mathcal{N}$ and $p \in \mathcal{P}$. Net η is said to be connected to pin p and pin p is said to be connected to net η. The set of all connections is $\mathcal{P} \times \mathcal{N}$.*

Modules

Modules drawn as rectangles in figure 3-1 (b) consist of a module type and an indexed set of pins. The pins are all unique to each other and to the module in that they are

not shared by other modules. The number of pins is determined by the *numberofpins* attribute of the module type. The pins are indicated by numbers inside the boundary of the module. A large symbol inside the module denotes the module type. The subscript or symbol next to the module type is a name given to the module for explanatory purposes.

Definition 3 *A* **module** *M is an n-tuplet $(T, P_0, P_1 \cdots P_{\sigma_T - 1})$ where $T \in \mathcal{T}$ and $P_0 \cdots P_{\sigma_T-1}$ are pins distinct from each other each of which is unique to M. The set $P_0 \cdots P_{\sigma_T-1}$ is denoted by $\mathcal{P}_M$. T is called the type of module M. For a P_i pinnumber(P_i) $= i$ and module(P_i) $= M$ is defined. The set of all modules is denoted by $\mathcal{M}$.*

Circuits

A circuit consists of a set of modules $\mathcal{M}_C$, a set of nets $\mathcal{N}_C$ and a set of connections $\mathcal{C}_C$ between the set of nets and the set of all the pins $\mathcal{P}_{\mathcal{M}_C}$ of the modules. In this model for circuits, the electrical connections between the modules have zero resistance. Resistances must appear explicitly by using resistor modules. Figure 3-1 is an example of a valid circuit.

Definition 4 *A* **circuit** *C is a triplet $(\mathcal{M}_C, \mathcal{N}_C, \mathcal{C}_C)$ where $\mathcal{M}_C \subseteq \mathcal{M}$, $\mathcal{N}_C \subseteq \mathcal{N}$ and $\mathcal{C}_C \subseteq \mathcal{P}_{\mathcal{M}_C} \times \mathcal{N}_C$, $\mathcal{P}_{\mathcal{M}_C} = \bigcup_{M_i \in \mathcal{M}_C} \mathcal{P}_{M_i}$ and finally $\forall (p, \eta) \in \mathcal{C}_C$, $(\{p\} \times \mathcal{N}_C) \cap \mathcal{C}_C$ is a singleton and $\mathcal{P}_{\mathcal{M}_C} \times \{\eta\} \neq \emptyset$.*

$\mathcal{M}_C$ is called the set of modules in C, $\mathcal{N}_C$ is called the set of nets in C and $\mathcal{C}_C$ is called the set of connections of C. The set of all circuits is denoted by $\mathbf{C}$.

It is often necessary to deal with portions of a circuit C consisting of some but not necessarily all modules and nets in C. These portions of C must themselves be legal circuits and are called networks of C.

Definition 5 *$N = (\mathcal{M}_N, \mathcal{N}_N, \mathcal{C}_N)$ is a* **network** *of circuit C if and only if $N \subseteq C$ and each set $\mathcal{M}_N$, $\mathcal{N}_N$, and $\mathcal{C}_N$ obeys the constraints of definition 4. The set of networks in circuit C is denoted by $\mathbf{N}_C$.*

Circuit Isomorphism

The circuits in figures 3-3 (a) and (b) are not *one and the same* since they have modules and nets that are distinct from one another. They are however identical in every other respect. A relation "$\simeq$" between circuits, such that circuits which are not necessarily *one and the same* but are structurally identical are in relation with one another, is defined. $C_1 \simeq C_2$ is read "C_1 is isomorphic to C_2". The $\simeq$ relation is similar in spirit to

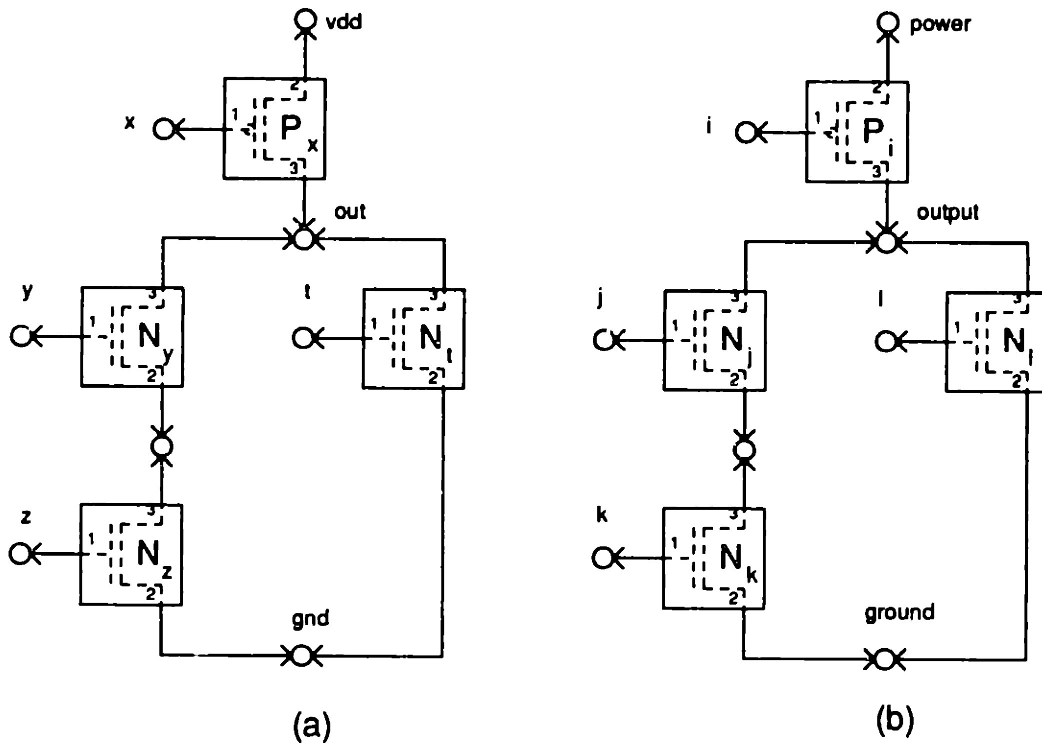


FIGURE 3-3: Isomorphic Circuits

the Lisp list comparison function "similar" [42] which returns true if and only if its two arguments represent the *same* list even if they are built with different *concells*.

Informally, two networks are isomorphic if there is a one to one mapping between their modules, nets and pins such that:

1. The image of the ith pin P of module M is the ith pin of the image of M.

2. If pin P is connected to net η then the image of P is connected to the image of η.

Formally:

Definition 6 *Two networks* $N_1 = (\mathcal{M}_1, \mathcal{N}_1, \mathcal{C}_1)$ *and* $N_2 = (\mathcal{M}_2, \mathcal{N}_2, \mathcal{C}_2)$ *are* **isomorphic** $N_1 \simeq N_2$ *if and only if there exist three one to one mappings* $f_M : \mathcal{M}_1 \rightarrow \mathcal{M}_2$, $f_N : \mathcal{N}_1 \rightarrow \mathcal{N}_2$ *and* $f_C : \mathcal{C}_1 \rightarrow \mathcal{C}_2$. *such that:* $\forall M \in \mathcal{M}_1$, $typeof(M) = typeof(f_M(M))$, *and* $\forall (p_1, \eta_1) \in \mathcal{C}_1$ *if* $(p_2, \eta_2) = f_C(p_1, \eta_1)$ *then* $\eta_2 = f_N(\eta_1)$, $module(p_2) = f_M(module(p_1))$ *and* $pinnumber(p_1) = pinnumber(p_2)$.

3.1.2 Circuit Grammars

In this section a precise definition for context free circuit grammars (CFCG for short) is given. Because the productions in these grammars involve circuits and since circuits

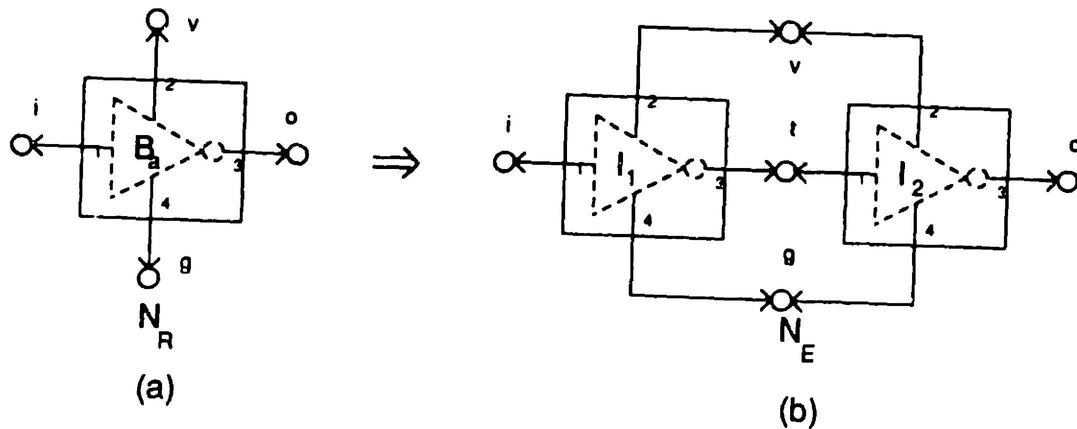


FIGURE 3-4: Example of a Circuit Production

are much more complex constructs than strings, a precise definition for circuit production must first be given.

Circuit Production

Just as string productions describe how a string symbol can be *expanded* in a string, circuit productions describe how a module M or rather how a network consisting of one module M can be expanded in a circuit. A circuit production is an association between a singleton network[1] N_R and a network N_E in which all the nets in N_R also appear in N_E.

Definition 7 *A* **production** *R is a pair of circuits $(N_R, N_E) \in \mathcal{C}\times\mathcal{C}$, $N_R = (\mathcal{M}_R, \mathcal{N}_R, \mathcal{C}_R)$ and $N_E = (\mathcal{M}_E, \mathcal{N}_E, \mathcal{C}_E)$ where $\mathcal{M}_R$ is a singleton and $\mathcal{N}_R \subseteq \mathcal{N}_E$. N_R is called the* LHS *of the production denoted R_{LHS} and N_E is called the* RHS *of the production denoted R_{RHS}. The set of all productions is denoted by $\mathcal{R}$.*

Figure 3-4 shows a pictorial representation of a circuit production. The LHS of the production is the buffer shown in figure 3-4 (a) and the RHS of the production is the two inverter circuit of figure 3-4 (b). Notice that all the nets in N_R appear in N_E.

3.1.3 Network Expansion and Reduction

In this section a new relationship between circuits denoted by "$\overset{R}{\Rightarrow}$" which closely parallels the $\Rightarrow$ relationship for strings is defined.

[1] A circuit which has one module in it.

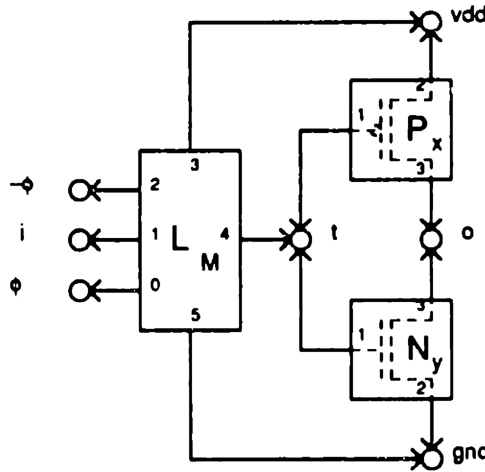


FIGURE 3-5: Circuit C before Expansion

A local operation on a module M in a circuit C called module expansion is defined. The result of applying this operation on M in C is a new circuit C' such that $C \overset{R}{\Rightarrow} C'$.

Let $C = (M_N, N_N, C_N)$ be a circuit, M a module in C and $R = (R_{\text{LHS}}, R_{\text{RHS}})$ a production such that the type of the module in R_{LHS} is the same as *typeof*(M). A pictorial example for C is shown in figure 3-5, an example for R is shown in figure 3-6.

Let N_M be the singleton network in N consisting of the module M, $(N_M = (\{M\}, \mathcal{N}_M, C_M)$ where $C_M = (\mathcal{P} \times \mathcal{N}_M) \cap C_N)$. N_M is shown in figure 3-7.

Let $N_E = (\mathcal{M}_E, \mathcal{N}_E, C_E)$ be a network isomorphic to R_{RHS} with f_N being the corresponding net isomorphism, with N_E such that:

- N_E has no modules in common with $\mathcal{M}_E$, $\mathcal{M}_E \cap M_N = \emptyset$
- The set of nets of network N_E includes the nets in the singleton network $\mathcal{N}_M$, $\mathcal{N}_E \subseteq \mathcal{N}_M$.
- If η is the net connected to pin i of M_{LHS} then $n' = f_N(\eta)$ is[2] the net connected to pin i of M. Formally: $\forall (p, \eta) \in C_{R_{\text{LHS}}}, (p', f_N(\eta)) \in C_M$ and $indexof(p) = indexof(p')$.

Figure 3-8 shows an example of the network N_E where C, R and N_M are the networks of figures 3-5, 3-6 and 3-7 respectively. Notice that the ith pin of M connects to the image net of the net connected to the ith pin of M_{LHS}.

Definition 8 Expanding *module M in N via production R is the process of replacing network N_M in N by network N_E. The new network N' obtained by this process is* $N' = ((M_N - \{M\}) \cup \mathcal{M}_E, N_N \cup \mathcal{N}_E, (C_N - C_M) \cup C_E)$.

The relationship between N and N' is denoted by $N \overset{R}{\Rightarrow} N'$.

[2]Since η is a net in R_{LHS} η is also a net in R_{RHS} and therefore $f_N(\eta)$ is defined.

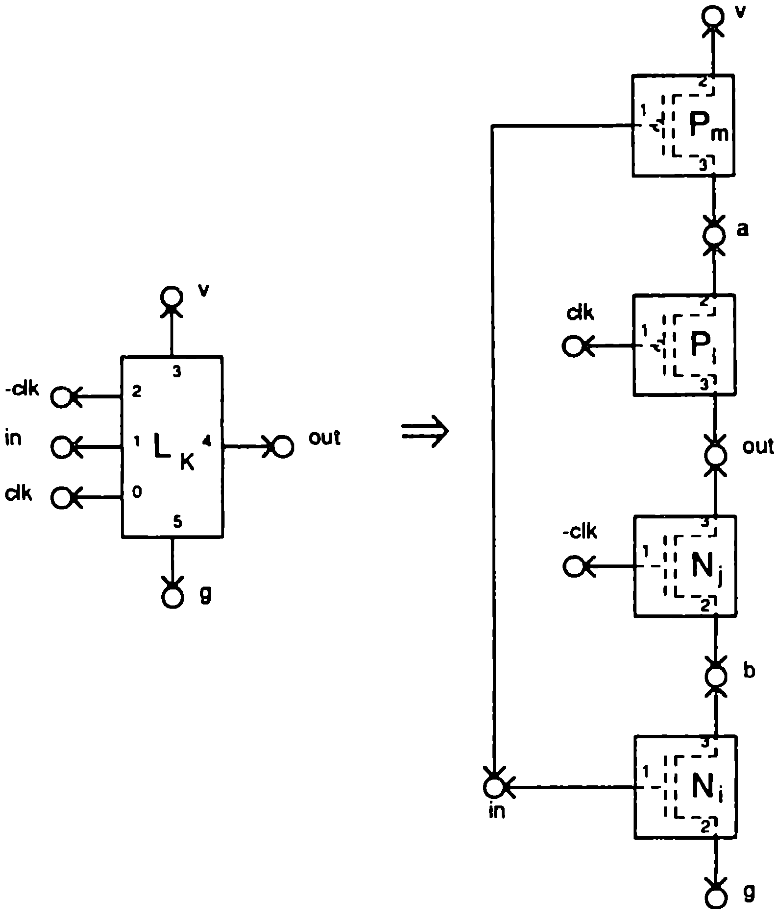


FIGURE 3-6: Latch Production R

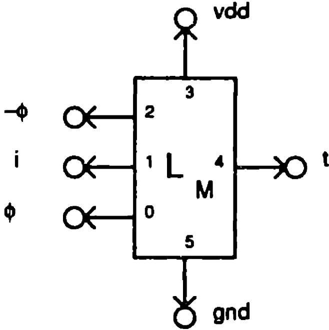


FIGURE 3-7: Network N_M of C

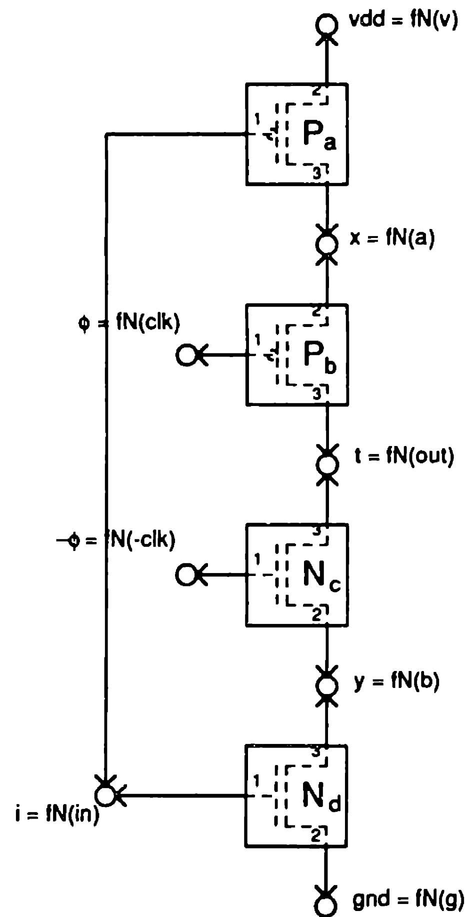


FIGURE 3-8: Expanded Network N_E

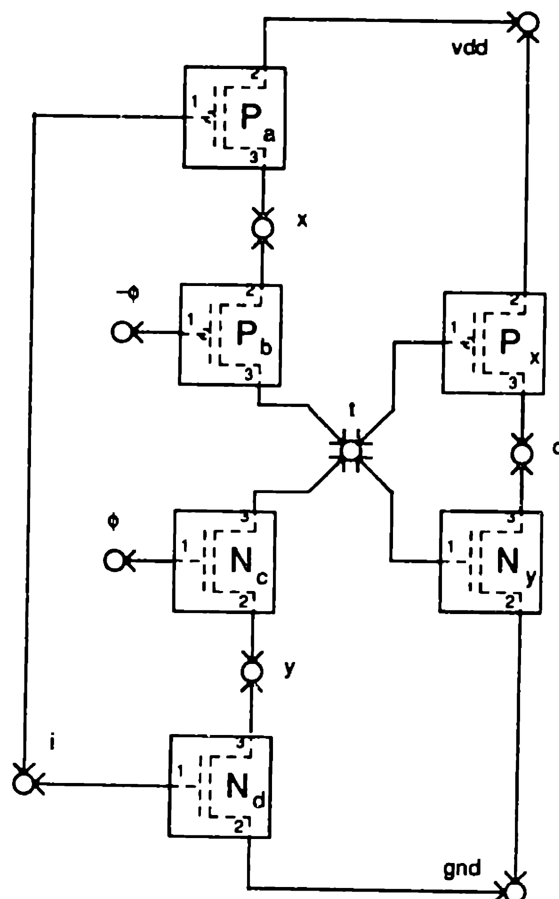


FIGURE 3-9: Resulting Circuit C' after Expanding C

The resulting circuit C' for the case where C, R, N_R and N_E are the networks of figures 3-5, 3-6 3-7 and 3-8 respectively is shown in figure 3-9.

With N, M , R, N_E and N' being defined as above the reverse operation from network expansion is now defined.

Definition 9 **Reducing** *network N_E in N' via production R is the process of replacing network N_E in N by network N_M. The resulting network N is such that $N \overset{R}{\Rightarrow} N'$.*

3.1.4 Context Free Circuit Grammar Definition

The definition of context free circuit grammars parallels that of CFSGs. The differences are that module types are used instead of string symbols and the circuit productions defined in section 3.1.2 are used instead of string productions.

Definition 10 *A context free* **circuit grammar** *G is a quadruple (A_G, T_G, S_G, R_G) where $T_G \subset A_G \subset \mathcal{T}$ and A_G is finite, $S_G \in \mathcal{M}$, typeof$(S_G) \in A_G$, $R_G \subset \mathcal{R}$ where $\forall R \in R_G$ the type of the module in the R_{LHS} is in $(A_G - T_G)$ and the types of the modules in R_{RHS} are elements of A_G. A_G, T_G, S_G, R_G are called the alphabet, terminals, start*

symbol and productions of G respectively. The set of all context free circuit grammars is denoted by $\mathcal{G}$.

The "$\stackrel{G}{\Rightarrow}$" relationship between circuits is now defined by "$C \stackrel{G}{\Rightarrow} C'$" if and only if "$C \stackrel{R}{\Rightarrow} C'$" where R is a production of G.

The relation $\stackrel{G*}{\Rightarrow}$ is the transitive closure of $\stackrel{G}{\Rightarrow}$ defined by: for any circuits C_1, C_2 and C_3, $C_1 \stackrel{G*}{\Rightarrow} C_1$ and if $C_1 \stackrel{G*}{\Rightarrow} C_2$ and $C_2 \stackrel{G}{\Rightarrow} C_3$ then $C_1 \stackrel{G*}{\Rightarrow} C_3$.

Let N_{S_G} be a network consisting of a module of type S_G.

Definition 11 *The* **range space** *or language L_G of a context free circuit grammar $G = (A_G, T_G, S_G, P_G)$ is defined as the set of all networks $\mathcal{N}_i$ which contain modules only in T_G such that $N_{S_G} \stackrel{G*}{\Rightarrow} \mathcal{N}_i$.*

3.1.5 Reducibility Condition

In this section a condition on any candidate network to be reduced, called the *reducibility condition* is introduced. A network can be reduced only if it obeys the reducibility condition. First by using the examples of figures 3-10, 3-11 and 3-12 the consequences of reducing a network that does not obey the reducibility condition is examined. Then by using the formal definitions for network expansion and reduction described in section 3.1.3, a condition on the network to be reduced that guarantees a *correct* reduction is derived.

Module Internal Sneakpath

In figure 3-9, the nets x and y are connected only to modules in N_E. Figure 3-10 depicts a situation where one of these nets, x, is also connected to the inverter module E. The result of replacing N_E by the network of N_{M_1} of figure 3-11 is shown in figure 3-12. The net x is still present in the network in order to accommodate the connection to pin 0 of module E. However the underlying electrical meaning of the module M_1 is not equivalent to and cannot be abstracted from the network N_E because M_1 violates our intuitive understanding of what a module is.

A module can electrically interface with the circuit it is used in only via its pins. There is a *sneakpath* connection from the *interior* of module M to net x. The circuitry *internal* to an abstracted module cannot connect to circuitry outside the module. Therefore if any of the nets x or y are connected to modules not in N_E, the reduction of N_E is illegal and has no electrical significance.

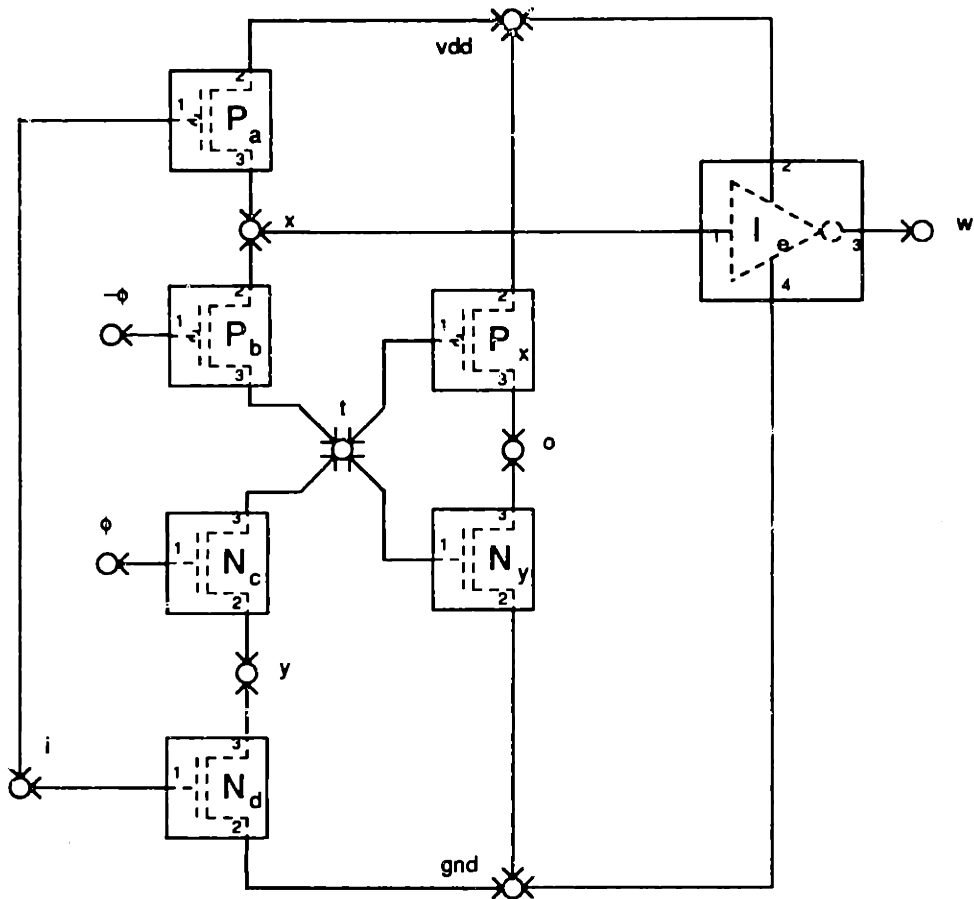


FIGURE 3-10: Circuit with *Illegal Connection*

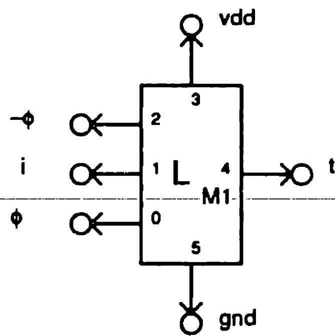


FIGURE 3-11: *Illegally* Reduced Network N_{M_1}

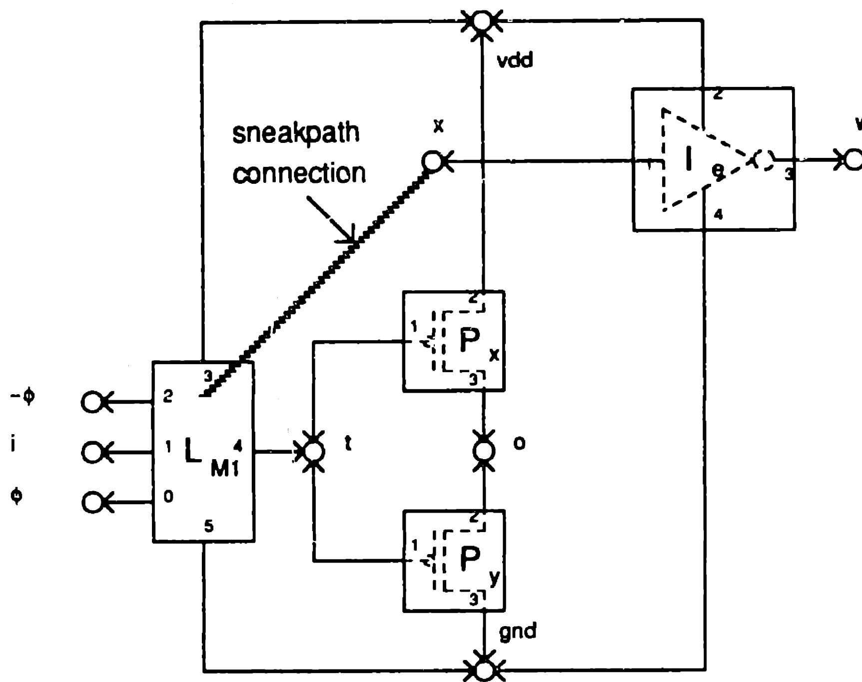


FIGURE 3-12: Circuit with Sneakpath Connection

Reducibility Condition Definition

When module M is expanded in circuit C to yield network C', new nets (x and y) in figure 3-9 are introduced. These new nets are the nets in $\mathcal{N}_E - N_M$ and by construction of network N' they connect only to the pins of modules in N_E. They are the images by f_N of the nets in $R_{\mathrm{RHS}} - R_{\mathrm{LHS}}$. They are referred to as the *internal* nets of R and their images in N_E by f_N are referred to as *internal* nets for the expansion (respectively reduction) of N_M (respectively N_E) via production R.

This result holds for any circuit C' and network N_E of C'. Since reduction is the exact reverse operation from expansion, for there to be a module M, a circuit C and a production R such that the expansion of N_M into N_E via production R yields C' it must be true that the images of the internal nets of R via f_N connect only to pins of modules in N_E. This requirement on the nets of N_E is referred to as the reducibility condition. Before reducing any network N_E in any circuit C by any production R, the internal nets of N_E must satisfy the reducibility condition.

Using the examples for N, N', N_E and R defined in section 3.1.2, figure 3-13 shows the production R and the circuits N and N'. The image nets of the internal nets of R (nets a and b) in N_E are the internal nets of the reduction and are allowed to connect only to modules in N_E in order for the reduction to be well formed.

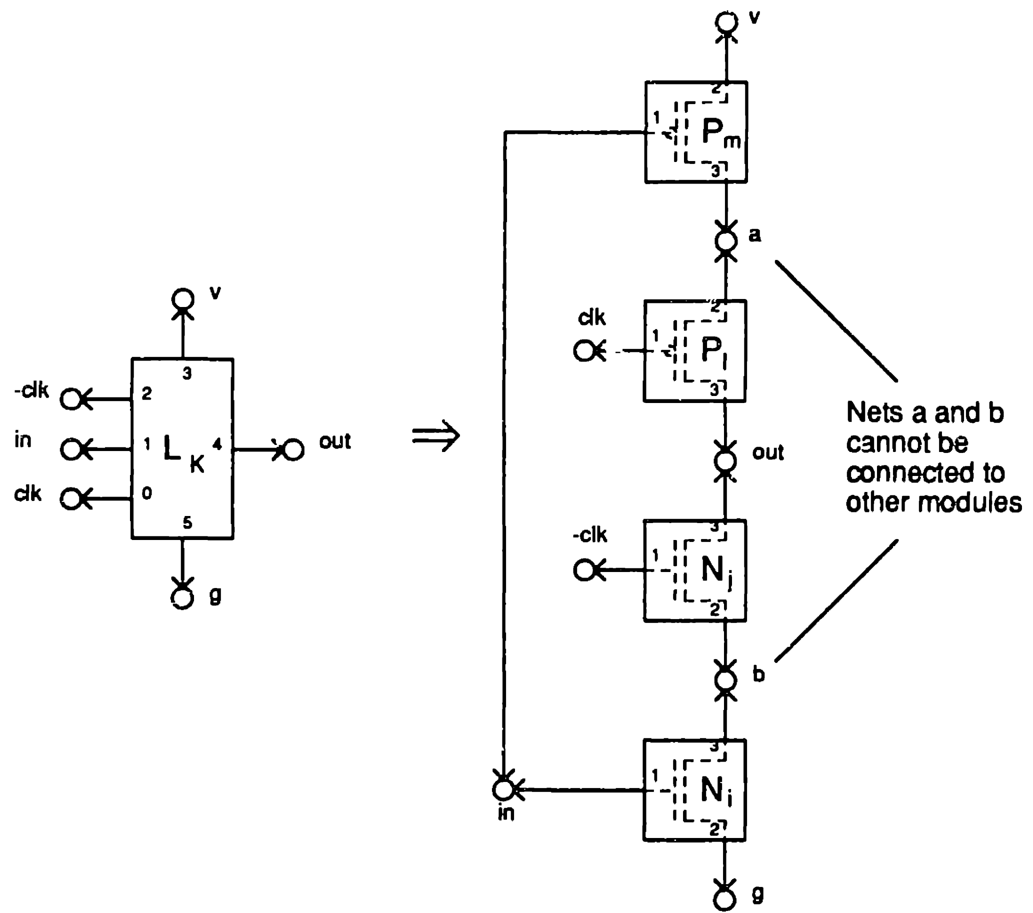


FIGURE 3-13: Reducibility Condition

3.1.6 Parsing

Given a CFCG G and a circuit C, methods for verifying if C is in the range space of G ($C \in L_G$) are examined in this section. Having captured a circuit style by its grammar G, verifying that C obeys the circuit style is accomplished by verifying that C is in the range space of G. An efficient technique for solving the range space membership problem is therefore central to our verification process.

Verifying that C is in the range space of G is equivalent to verifying that $C_{S_G} \overset{G*}{\Rightarrow} C$, where C_{S_G} is a singleton circuit consisting of a start symbol module of G. The two major techniques for accomplishing this are ***top down parsing*** and ***bottom up parsing*** both described in this section.

Top Down Parsing

Top down parsing is the process of finding a sequence of networks $C_0 \cdots C_i \cdots C_n$ such that $C_0 \simeq C_{S_G}$, $C_i \overset{G}{\Rightarrow} C_{i+1}$ and $C_n \simeq C$. For each circuit C_i in the sequence, a production R_i of G and a singleton network N_i^M on which an expansion via R_i will be performed is selected. Each of these expansions yields a new circuit C_{i+1}. The difficulty in top town parsing is choosing the right production R and network N_i^M for each C_i.

Top down parsing is unsuitable for incremental circuit verification. The choice of the very first production R_0 and network N_0^M has an impact on the entirety of each of the circuits C_i and hence affects the entirety of C_n. Therefore top down parsing in general requires knowledge about the entirety (or at least certain *strategic* key features) of the circuit C_n before the very first derivation can be performed. Therefore verification of C can begin only after C has been completed.

Unlike strings, circuits have no beginning from which to start parsing. Since there is no natural ordering of the components in the circuit, in order to search for a distinctive feature in the circuit for selecting the next expansion, the whole circuit may have to be examined. Because the very first expansion affects the entirety of C_n, an examination of a large portion of C_n must be performed before it can be executed. Subsequent expansions may require examination of smaller portions of C_n but there will be several expansions requiring examination of large portions of C_n. This suggests that the selection process for each N_i^M and R_i will require a non local and possibly complex decision algorithm.

Bottom Up Parsing

Bottom up parsing is the process of finding a sequence of networks $C_0 \cdots C_i \cdots C_n$ such that $C_0 = C$, $C_{i+1} \overset{G}{\Rightarrow} C_i$ and $C_n \simeq C_{S_G}$. For each circuit C_i in the sequence, a

production R_i of G and a network N_i^E of C_i on which the reduction via R_i is performed is selected. Each of these reductions yields a new circuit C_{i+1}. The difficulty in bottom up parsing is choosing the right production R_i and network N_i^E for each C_i.

Bottom up parsing is especially suited for incremental circuit verification. Each of the first few reductions affects only a small network of C. The reductions can be performed almost as soon as these networks are completed. In section 3.1.7 it will be shown that the selection process for each N_i^M and R_i can be accomplished by a simple and local decision algorithm. This algorithm examines the *neighboring* modules of a network N and how they are connected to the modules in N and decides which production R in G if any should be applied to reduce N. Thus reduction for production R_i may be performed almost as soon as the corresponding N_i^E is completed. These reductions create new composite modules which ultimately form new networks N_j^E. Reduction of these new networks may also be performed almost as soon as they are completed.

Given that bottom up parsing encourages incremental verification and that a simple bottom up network and production selection algorithm exists, bottom up parsing is used to solve the grammar range space membership problem for circuits. Henceforth in any reference to parsing it will be assumed that bottom up parsing is being performed.

Parse Tree

A useful aid for visualizing the sequence of reductions during parsing is the *parse tree*. The parse tree contains a log of which networks were reduced, which new modules got created in the process and the production involved. Given a circuit C and a sequence of reductions on C which yields the start symbol module for the grammar, the parse tree can be thought of as the proof that the circuit is in the range space of the grammar.

If P_1 (respectively P_2) is a production which expands a 2 input NAND gate (respectively inverter) into its constituent transistors and P_3 is a production which expands an AND gate into a NAND gate followed by an inverter, then the parse tree for the circuit in figure 3-14 might look like figure 3-15.

The parse tree is used in section 5.1.6 for efficiently checking the consistency of incremental additions and deletions to the circuit. It is also useful when reporting errors back to the user.

3.1.7 Deterministic Reduction

It is not necessarily true that a network which satisfies the reducibility condition, and for which there is an applicable rule in the grammar, should be reduced. Sometimes such

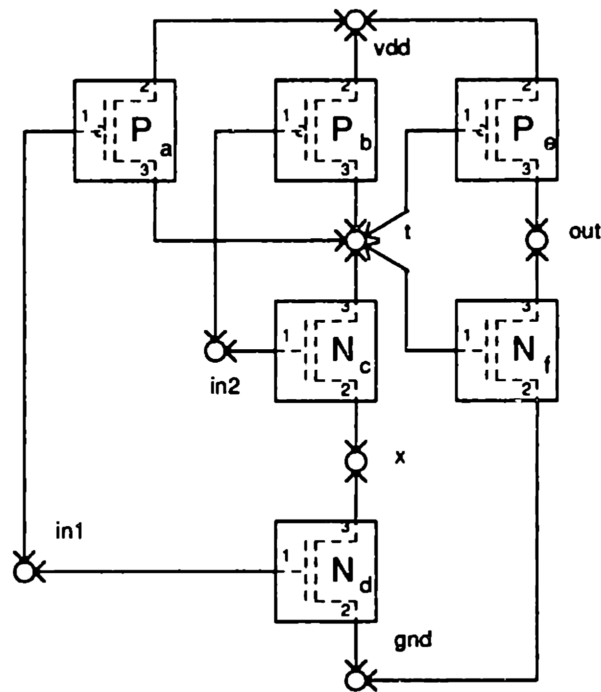


FIGURE 3-14: NAND Gate and Inverter Circuit

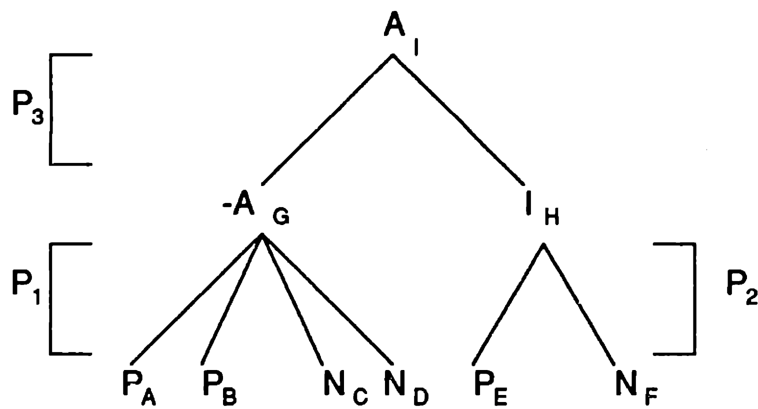


FIGURE 3-15: NAND Gate and Inverter Parse Tree

reductions can result in a situation where further reductions on the resulting circuit are not possible even though the initial circuit obeys the circuit methodology being checked for.

Consider the string grammar G whose productions are:

$$S \rightarrow AC \qquad A \rightarrow ab$$
$$C \rightarrow cd \qquad W \rightarrow bc$$

Then $S \Rightarrow AC \Rightarrow abC \Rightarrow abcd$ so $abcd \in L_G$ however $aWd \Rightarrow abcd$ but $S \not\Rightarrow aWd$. Therefore even though bc can be reduced into W this reduction should not be performed because it leads to aWd from which no more reductions are possible even though the string is in the language of the grammar. The same holds for circuit grammars and hence a mechanism for deciding which of any possible reductions to apply is necessary.

If for any circuit that obeys the design methodology there is a procedure for deciding which networks to reduce such that the sequence of reductions is guaranteed to result in the start symbol S_G, then the grammar is said to be *deterministic*. For grammars that are not deterministic, many reductions must be tried and then later undone when it can be shown that they lead to *dead ends*. This typically results in parse times that are exponential in the number of modules in the circuit.

Given a subnetwork, GRASP decides whether or not to reduce it by examining its *connected neighborhood*. In this *neighborhood* GRASP looks for conditions that guarantee that the subnetwork can be reduced. These conditions are the presence or absence of certain modules referred to as *condition* modules. The modules whose presence is required are referred to as *presence condition* modules. The modules whose absence is required are referred to as *absence condition* modules. *Presence* and *absence condition* modules correspond to *look-ahead* and *pushdown automata state* [3] for string grammars.

Presence Condition Modules

Checking for the *presence condition* modules comes down to identifying the augmented networks N'_E which consist of the networks N_E isomorphic to some R_{RHS} plus the required *presence condition* modules. The network formed by the RHS of the production R_{RHS} plus the presence condition modules is called the augmented production RHS. With the above criterion, deterministic parsing corresponds to identifying the augmented network(s) N'_E, but only the modules in N_E are removed from the circuit and replaced by a new composite module; the *presence condition* modules in N'_E are left in the circuit. In the rule in figure 3-16 the augmented subnetwork N'_E consists of the two transistors which constitute

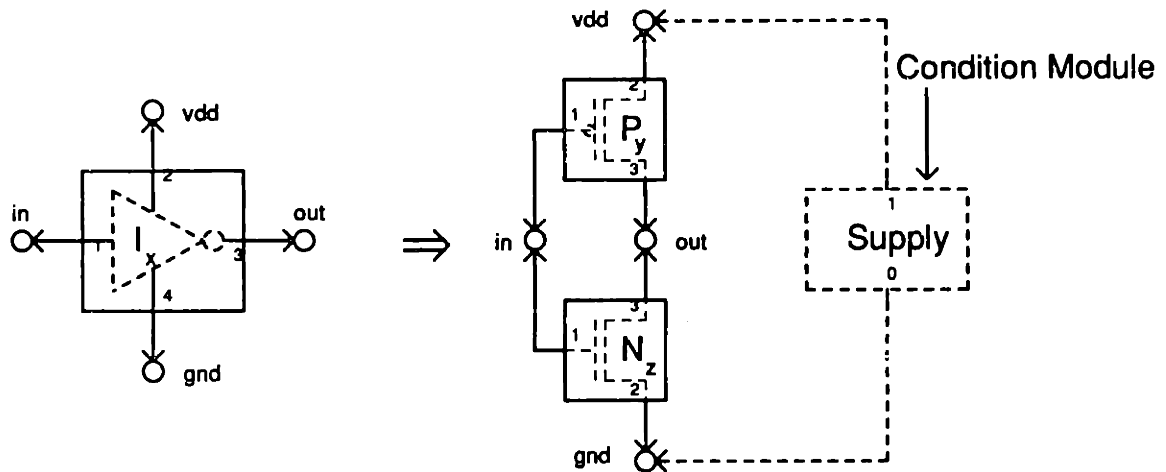


FIGURE 3-16: Presence Condition Modules

the associated subnetwork N_E and the *power supply* module which is a *presence condition* module. During the application of this rule only the transistor modules in N_E get reduced. The *power supply* module, which was necessary for the rule to be applied, is left in the circuit.

Absence Condition Modules

Checking for *absence condition* modules comes down to identifying the networks N_E and verifying that specific pins numbers of specific module types do not connect to certain external nets of the network N_E. Figure 3-17 shows a hypothetical[3] case where the production P produces an inverter and the corresponding reduction R requires that the output of the inverter not be connected to the input of a latch. This requirement is referred to as an *abscence condition* and module z in figure 3-17 is referred to as an *absence condition module.* A better example will be given in section 4.2.1.

Absence conditions are used much more rarely than *presence conditions.* They are generally used to enforce production precedence which blocks certain productions from firing until some other productions (of higher precedence) cannot be fired first.

Consider the string grammar G whose productions are:

$$\begin{array}{lllcllll} S & \rightarrow & AB & & & & \\ A & \rightarrow & aA & & A & \rightarrow & a \\ B & \rightarrow & Bb & & B & \rightarrow & b \end{array}$$

During a parse of the string $\gamma = \alpha AB\beta$ where α and β are strings, the production

[3]This production is for explanatory purposes only and has no electrical meaning.

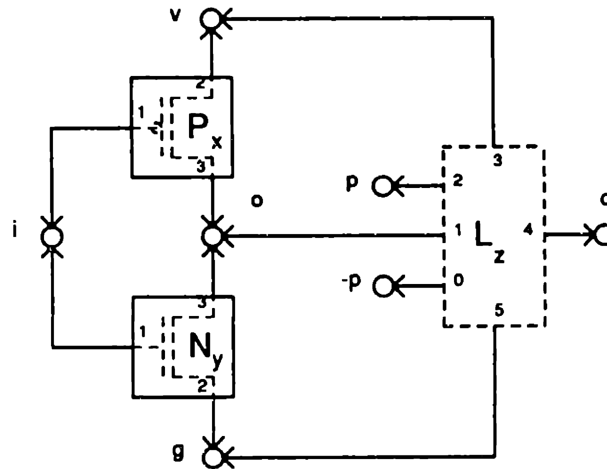


FIGURE 3-17: Absence Condition

$S \rightarrow AB$ should be used only if there is no a to the left of the A and no b to the right of the B.

In a circuit there is no natural *left to right* ordering of modules and hence no standard *left to right* parsing techniques such as LALR [3] parsing are applicable for circuit grammars. Networks in the circuit get reduced in *pseudo random* fashion and therefore the situation described above for strings does (occasionally) occur.

Absence condition modules suffer from the serious problem that the absence module for reduction R may not be present at the time the reduction R is applied but may appear later from the result of some other reduction. For example, consider the circuit of figure 3-18 which is similar to figure 3-17 except that the transistors constituting the latch have not yet been reduced into a latch module. Thus the attempted reduction of circuit figure 3-17 will be performed. Sometime later the latch transistors will have been replaced by the latch module but the reduction R will already have been performed.

In order to correctly prevent R from being performed not only must pin 1 of a latch moduletype on net "o" be an *absence condition* but so must any module susceptible of being reduced into a latch module. In the case of figure 3-18 it would appear that pin 0 of any module of type n-tran or p-tran should also be an absence condition module. However adding those two *absence conditions* will incorrectly prevent the circuit on the left hand side of figure 3-19 from being reduced. Only those cases which necessarily lead to a latch module have pin 1 connected to net o and no other condition must inhibit the reduction from being performed. A safe and easy way to use *absence conditions* is when the *absence module* is a primitive module type.

In order to use *non terminal* absence modules and avoid the problems described above, a special case of *absence conditions* with *well behaved* properties is used. In this scheme instead of listing which modules should not be present the modules that are allowed to

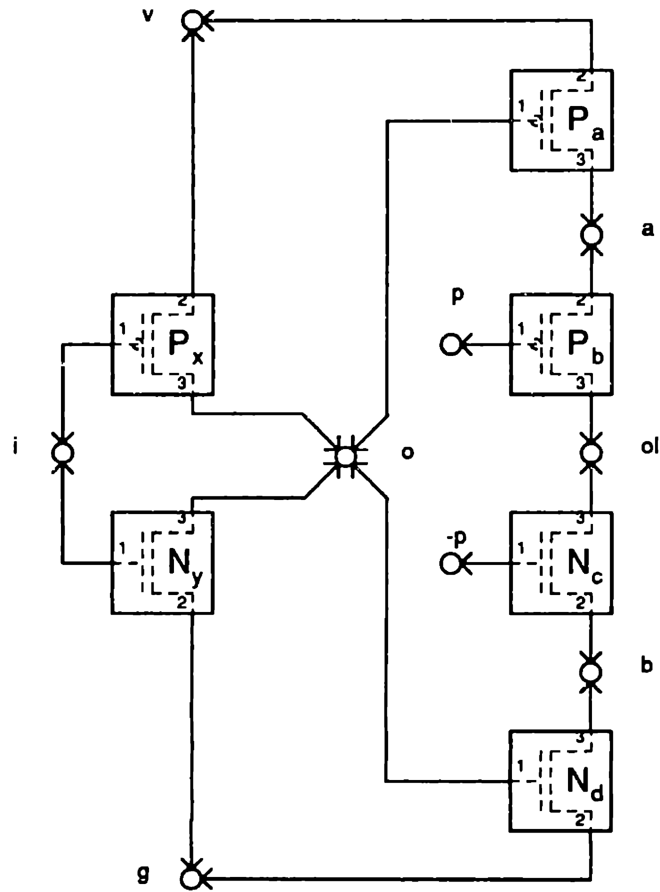


FIGURE 3-18: Expanded Absence Module

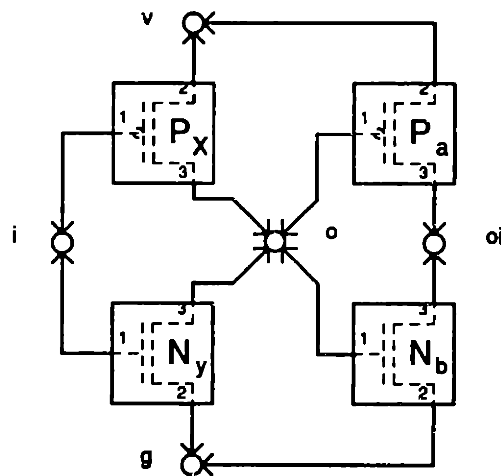


FIGURE 3-19: Expanded Non-absence Condition Module

be present are listed. Only specific pins of specific moduletypes are allowed to connect to certain external nets of the network being reduced. All other connections (excluding the connections by the modules in the network being reduced) will inhibit the reduction from being performed. Section 5.1.5 describes how GRASPs *event driven* algorithm is used to schedule the reduction when all of the *absence conditions* are finally met.

In the case of figure 3-17, a reasonable *absence condition* could be "allow no connections other than pin 0 of an inverter type on net *o*". Only when no other module except perhaps pin 0 of an inverter connects to net *o* can the reduction be performed. In section 5.1.5 a technique for using GRASP's event driven parsing algorithm to *schedule* the reduction after the inverter has been created is described.

Ambiguous Grammars

Ambiguous grammars are grammars for which there are several different parse trees for a given schematic. A grammar can be both deterministic and ambiguous at the same time. Determinism guarantees that if the schematic is in the range space of the grammar, the sequence of reductions will yield the start symbol of the grammar. This sequence of reductions is characterized by the parse tree. Determinism does not require the parse tree to be unique. If the grammar is ambiguous[4], there may be several different parse trees for a given schematic. Any one of them can be used to prove the membership of the schematic to the range space of the grammar. Therefore ambiguous grammars pose no problem[5] for the grammar based verification method proposed in this chapter.

3.1.8 Waveform Generators

The correct function of a circuit depends not only on its structure but also on how it is connected to external waveform generators such as power, ground and clocks. Even the circuitry connected to the chips input and output pins can be thought of as external waveform generators. By considering these waveform generators as modules that are part of the circuit, circuit methodologies can be expressed by a grammar whose alphabet consists of both the circuit (e.g., transistor) and the generator (e.g., power and clocks) module types. Recognizing that a net is, for instance, a *vdd net* is achieved by observing that it is properly connected to the *power generator* module.

[4]Many of the grammars described in chapter 4 are highly ambiguous.

[5]During incremental design, grammatical ambiguity can in fact be advantageous. Because of the larger number of reduction options available, a greater part of the parse tree can be constructed from a partially finished circuit and therefore errors in the design can be detected earlier.

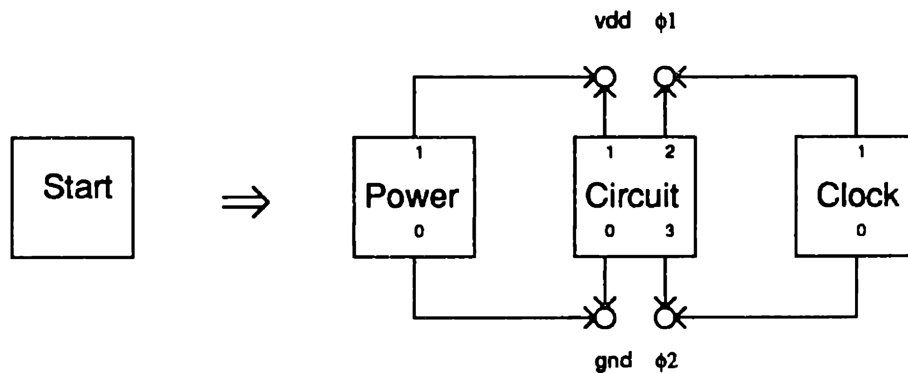


FIGURE 3-20: Waveform Generator Production

In terms of the grammar formalisms developed so far these waveform generators are treated as *presence condition* modules. Since these waveform generator modules are now truly part of the circuit to be verified, typically the production(s) that expand the start symbol module of the grammar have the form shown in figure 3-20. In this type of production the start symbol module M_S is expanded into the *actual circuit* to be verified represented by module M_C and the waveform generator modules M_{power} and M_{clock}.

3.1.9 Net Bundles

Overview

A circuit to be verified by GRASP typically consists of modules of type *transistor*, *clock* and *power supply*. As parsing proceeds, networks which correspond to the RHS of a rule are replaced by new composite modules which correspond to the LHS of the rule. These new modules form new smaller networks, which are in turn replaced by other composite modules using the same process. After several such reductions, each resulting composite module M_i will typically correspond to a rather large network N_i of the initial circuit. The size of these networks N_i and the number of connections they may have to the rest of the initial circuit is unbounded.

By virtue of being reductions of the networks N_i, the composite modules M_i must necessarily connect to all the nets in common between N_i and the rest of the initial circuit. Because the new composite modules M_i must also belong to one of the module types in A_G, the maximum number of pins they can have is fixed by A_G and is bounded. Therefore, since the number of nets a composite module must connect to is unbounded, whereas the number of pins it can have is not, the concept of a *net bundle* is introduced. Net bundles allow composite modules with a small set of pins to represent arbitrarily large networks with an arbitrary number of connections to the outside world.

Because net bundles are central to allowing the grammatical techniques described in

this thesis to be applied to design style verification, the following subsections describe in more rigor and detail the problems described above. Section 3.1.9 describes *net bundles* and how their use solves the above problems. Definitions that will be used in the following sections are first provided.

Definitions

- $pins(M)$ is the number of pins of module M.
- R_G^{max} is the maximum number of modules in any rule of grammar G.
- C_0 is the set of all circuits and given a circuit C, N_C is the set of all networks included in C.
- M_C^G is the set of all modules which occur during a parse of circuit C with grammar G ($M \in M_C^G$ if and only if $M \stackrel{*}{\Rightarrow} N$ where $N \in N_C$).
- For any circuit C (respectively network N) $sizeof(C)$ (respectively $sizeof(N)$) denotes the number of modules in C (respectively N).
- For a module M which is a multistep reduction of a network N in the initial circuit ($M \stackrel{*}{\Rightarrow} N$ and N has only terminal modules) $sizeof(M) = sizeof(N)$ is defined.
- For network N, ν is a boundary net if and only if ν is connected to at least one module in N and at least one module not in N. The set of boundary nets of N is denoted by $boundaryset(N)$.
- For the trivial network N consisting of the sole module M $boundaryset(M) = boundaryset(N)$ is defined.
- The number of connections a network N has to the rest of the circuit is denoted by $connections(N)$ and is defined as the number of boundary nets for N.
- If M is a module which is a multistep reduction of N ($M \stackrel{*}{\Rightarrow} N$) then $connections(M) = connections(N)$ is defined.

Maximum Module Size

Circuits for most useful design methodologies can be of arbitrary size. Therefore the range space of any useful circuit grammars must include circuits of arbitrary size. Since at the end of a successful parse only the start symbol module M_{S_G} remains, $sizeof(M_{S_G}) = sizeof(C)$ can be arbitrarily large.

Maximum Number of Connections

In this subsection it is sought to establish that it is not possible to design a grammar such that for any module M_i created during parsing ($M_i \in M_C^G$), connections(M_i) is bounded by a constant independent of C as will be shown in theorem 3. If the reader is convinced of this he can safely skip the remainder of this subsection and go on to section 3.1.12. Theorems 1 and 2 are used to prove theorem 3. Finally theorem 4 is used to prove equation 3.6 which shows that if each pin connects to only one net, an infinite alphabet set A_G and mapping set M_G is necessary.

Theorem 1 *During the parse of any circuit C for any positive number n less than sizeof(C) it is always possible to find a composite module M_i such that sizeof(M_i) is within a factor of R_G^{max} from n. More formally stated:*

$$\begin{gathered} \forall C \in C_0, \forall \alpha, \beta \in \Re^{\star +}, \beta < 1, \tfrac{\beta}{\alpha} > R_G^{max}, \exists M_i \in M_C^G \textit{ such that} \\ \alpha \cdot sizeof(C) \leq sizeof(M_i) \leq \beta \cdot sizeof(C) \end{gathered} \tag{3.1}$$

Proof 1 *Since at the end of a successful parse only one module remains, there is at least one module M_S, such that $sizeof(M_S) > \beta \cdot sizeof(C)$. Let M_F be the first module created such that $sizeof(M_F) > \beta \cdot sizeof(C)$. Let N_F be the network M_F was derived from in one step ($M_F \Rightarrow N_F$) and $M_{N_F}^{max}$ the module with the largest size in N_F. By definition of M_F, $sizeof(M_{N_F}^{max}) \leq \beta \cdot sizeof(C)$ because $M_{N_F}^{max}$ was created before M_F. Also:*

$$\begin{gathered} \alpha \cdot sizeof(C) \leq \tfrac{\beta}{R_G^{max}} \cdot sizeof(C) \leq \tfrac{sizeof(M_F)}{R_G^{max}} \leq sizeof(M_{N_F}^{max}) \\ \textit{therefore: } \alpha \cdot sizeof(C) \leq sizeof(M_{N_F}^{max}) \leq \beta \cdot sizeof(C) \end{gathered}$$

therefore $M_i = M_{N_F}^{max}$ can always be chosen.

Theorem 2 *In order to divide a circuit C obeying methodology M into two very roughly comparable parts the size of the cut set required cannot be bounded by a constant independent of C. More formally stated:*

$$\begin{gathered} \forall n \in \aleph, \exists C \in C_0, \alpha, \beta \in \Re^{\star +}, \beta > 1, \tfrac{\alpha}{\beta} > R_G^{max} \textit{ such that} \\ \forall N \in N_C, \alpha \cdot sizeof(C) \leq sizeof(N) \leq \beta \cdot sizeof(C) \\ \textit{It is true that: } connections(N) > n \end{gathered} \tag{3.2}$$

Proof 2 *Connections(M) is the number of connections that have to be cut to separate N from C. Since the ratio $\frac{sizeof(C)}{sizeof(N)}$ is an element of a closed interval in $]0,1[$ this number is also the bisection width of the graph formed by circuit C. The bisection width of many*

graphs for many useful circuits such as FFTs is known to grow at least linearly with sizeof(C) [49], ($O(\frac{n_{mod}}{T})$ for the FFT). Hence connections(M) $\geq O(sizeof(C))$.

Given the two previous theorems it will now be proved that for any $n \in \aleph$ a circuit $C \in C_0$ and a module $M_i \in M_C^G$ such that $connections(M_i) > n$ can be found. In other words, $connections(M_i)$ cannot be bounded by a constant independent of C.

Theorem 3

$$\forall n \in \aleph, \exists C \in C_0, M_i \in M_C^G \text{ such that } connections(M_i) > n \tag{3.3}$$

Proof 3 *Given n let $\alpha, \beta \in R^{*+}, C \in C_0$ satisfy the requirements on them specified in theorem 3.2 and M a module satisfying the conditions in theorem 3.1. Let M_i be a module then satisfying the requirements of equation 3.1 ($\alpha \cdot sizeof(C) \leq sizeof(M_i) \leq \beta \cdot sizeof(C)$). It is known that such a module exists because of theorem 3.1. Let N_i be the network in the initial circuit that M_i was derived from. N_i must necessarily satisfy $\alpha \cdot sizeof(C) \leq sizeof(N) \leq \beta \cdot sizeof(C)$ because M_i was derived from N_i and M_i satisfies the constraints in theorem 3.1. Because N_i satisfies the constraints in theorem 3.2 it must be the case that $connections(N_i) > n$. Therefore $connections(N_i) = connections(M_i) > n$.*

3.1.10 Equality of Boundary Sets

In this section it will be established that boundary sets are preserved under network reduction operations.

Theorem 4

$$\forall C \in C_0, N' \in N_C, \text{ If } N \Rightarrow N' \text{ then } boundaryset(N) = boundaryset(N') \tag{3.4}$$

Proof 4 *Let ν be a boundary net for N. ν connects to a module $\overline{M}$ not in N and therefore obviously not in N'. ν also connects to a module M in N. If M was not involved in the $N \Rightarrow N'$ expansion then M is in N' and therefore ν connects to a module in N'. Otherwise M was replaced by a network N_M which must necessarily also have a module connected to ν by definition of a well formed expansion defined in section 3.1.3. Since N_M is a subnetwork of N', ν connects to at least one module in N'. Therefore it can be concluded that boundaryset(N) $\subseteq$ boundaryset(N')*

Let ν' be a boundary net for N'. ν' connects to a module $\overline{M}$ not in N' and therefore obviously not in N. ν' also connects to a module M' in N'. If M' is not in N_M then M is

also in N and ν' is therefore a boundary net for N. Otherwise ν' cannot be an internal net for the network N_M because it connects to $\overline{M}$ and therefore also connects to M $(M \Rightarrow N_M)$ by definition of a well formed expansion. Therefore $boundaryset(N') \subseteq boundaryset(N)$. Therefore it can be concluded that $boundaryset(N) = boundaryset(N')$.

3.1.11 Minimum Number of Pins

It will now be proved that $pins(M) \geq connections(M)$, i.e. the number of nets a module M connects to is equal to the number of connections the network N it was derived from had with the rest of the circuit. By induction on theorem 4 it can easily be shown that if $N \stackrel{*}{\Rightarrow} N'$ it is necessarily true that $boundaryset(N) = boundaryset(N')$. In the case where N consists of one sole module M, $boundaryset(M) = boundaryset(N) = boundaryset(N')$. The boundary set for M consists of those nets connected to M that are also connected to other modules. If each pin of M connects to exactly one net then:

$$\begin{aligned} Pins(M) &\geq Cardinality(boundaryset(M)) \\ Cardinality(boundaryset(M)) &= Cardinality(boundaryset(N')) \\ &= connections(N') \\ &= connections(M) \end{aligned} \tag{3.5}$$

Using the inequality of equation 3.5 and the results from theorem 3 it can be concluded that:

$$\begin{gathered} \text{If each pin connects to exactly one net} \\ \forall n \in \aleph, \exists C \in C_0, M \in M_C^G \text{ such that } Pins(M) > n \end{gathered} \tag{3.6}$$

Equation 3.6 poses a serious problem. It says that if each pin of every module connects to only one net then the number of pins a module must have is unbounded. The number of pins for each module is determined by its module type which is an element of the symbol set A_G. Allowing the maximum number of pins a module can have to be unbounded requires that the alphabet set A_G be infinite. From section 3.1.4 recall that in order for G to be a grammar, A_G must be a finite set. This clearly conflicts with the requirements of equation 3.6.

The problems associated with having an infinite alphabet set A_G are more deep rooted than simply requiring a departure from standard grammar methodology. An infinite alphabet set A_G requires that M_G contain an infinite number of mappings. These mappings must then be specified in *parametric form* so that a *finite representation* of the mappings can express the infinite set M_G. Allowing *too much* freedom in the specifications of these

mappings causes a substantial departure from the formalisms and techniques applicable to grammars and can result in ad hoc and unwieldy methods.

Section 3.1.12 defines a technique which effectively deals with the problem posed by equation 3.6 while preserving the grammatical formalisms of section 3.1.2.

3.1.12 Net Bundle Definition

For many types of modules (mostly composite) that occur in practice, pins can be grouped into a few small classes. For example, for any static gate module, pins can be grouped into 4 basic classes: input, output, power and ground pins. In a well-formed circuit each class of pins of a module should be connected (via a net) to other specific module types and pin classes. For example, the *power* pin of a module should be connected to the *vdd* pin of a 5V supply, and the input pin of a static gate should be connected to output pins of other static gates etc.. Permuting the connections of two pins (of the same module) in different classes will very likely cause the circuit to become ill-formed. For example, permuting the nets connected to the output and power pins of a static module will cause the output pin to be connected to the power supply and as a result the circuit will become ill-formed. By contrast, permuting the connections of two pins in the same class may change the functionality of the circuit but will not cause the circuit to become ill-formed. This observation will be used as the definition for pin equivalence class.

Definition 12 *For a given methodology $\mathcal{M}$ and module type $\mathcal{T}$, two pins are in the same pin equivalence class if and only if in any circuit C which obeys methodology $\mathcal{M}$ and any module M in C of type $\mathcal{T}$, the new circuit C' obtained by permuting the connections to the corresponding pins of M, still obeys the methodology $\mathcal{M}$.*

In equation 3.6 it has been shown that it is not possible to have an alphabet set with modules having an a-priori bounded set of pins. However it is possible to construct useful alphabet sets where each module symbol has a finite set of pin classes. Common methodologies, such as static CMOS and NORA, are designed to be used by human designers capable of handling only a few pin classes at a time. Therefore the module blocks used to build circuits in these common design styles have less than half a dozen pin classes. Useful module symbols with a finite set of pin classes include:

- Combinatorial block module. Pin classes are: input pins, output pins, power pin(s), and ground pin(s).

- Precharged block module. Pin classes are: input pins (they typically come from other precharged blocks after passing through an inverter), output pins (they go to other output pins after passing through an inverter), precharge clock pin(s), evaluate clock pin(s), power pin(s) and ground pin(s).

- Latch block module . Pin classes are: input pins, output pins, *latch_open* pin(s), $\overline{latch_open}$ pin(s), power pin(s) and ground pin(s).

It is possible to define the mappings in M_G in terms of the pin classes. The mappings need not distinguish between pins in a given class because permuting the connections of those pins does not change whether the circuit is well-formed or not. Furthermore, since the mappings are derived from methodology considerations and typically define conditions that each pin class must satisfy, the form of the mappings is almost always independent of the number of pins in each class. For example, if there is a mapping which involves static modules with 4 inputs there is usually a related mapping involving static modules with 5 inputs. In this case it is possible to describe the mappings independent of the number of pins in each class. In the three module block types described above the composition rules can be described *independent* of the number of pins in each class. Keeping these properties in mind the concept of a *net bundle* is introduced.

Definition 13 *Given a circuit C and a methodology $\mathcal{M}$, for any module M the set of nets $\mathcal{B}$ which connects to all the pins of M of a particular class is called a* **net bundle**.

For a given module, a pin class can be said to *connect* to a net bundle because each pin in the pin class connects to a net in the net bundle. It is therefore tempting to replace the concept of pin with pin class and net with net bundle.

By modifying some of the definitions for: Module type, Module, Pin, Net, Network and Circuit defined in section 3.1.1 almost all the formalisms developed in previous sections can be maintained. The only modification to the definitions of section 3.1.1 that need to be performed is to replace *pins* with *pin classes* and *nets* with *net bundles.* With these new definitions any module symbol can potentially represent a module with any number of pins. Now for any grammar G, the alphabet set A_G is finite but actually represents an infinite set of module types. Similarly the set of mappings M_G is finite but represents an infinite set of mappings. The pictorial representations for networks remain the same except that small circles now represent net bundles and the connections to those net bundles are now accomplished by pin equivalence classes.

Pin classes will henceforth be referred to simply as pins and net bundles as nets except where it becomes necessary to draw the attention to attributes of pin classes or

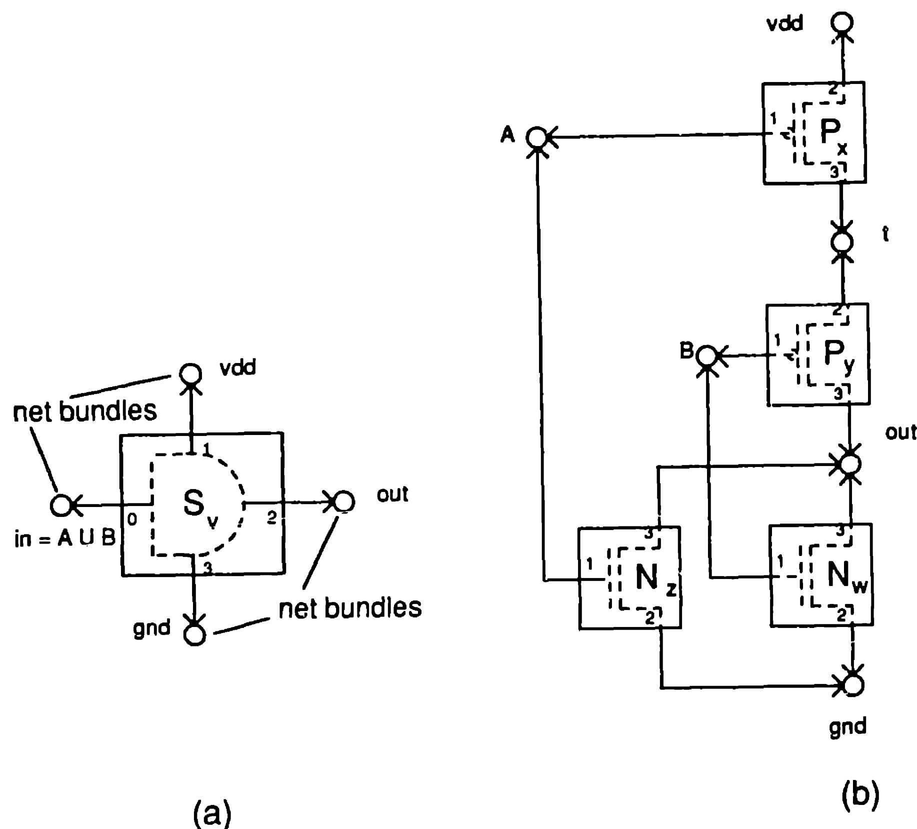


FIGURE 3-21: Example of Net Bundles

net bundles not found in pins or nets. Almost all of the formalisms developed in the previous sections will carry through except where specified.

3.1.13 Examples of Net Bundles

Figure 3-21 (a) shows a static gate symbol which represents all static gate modules irrespective of the number of inputs as described in section 3.1.12. This gate can be derived by reducing a network such as the one in figure 3-21 (b). The input and output pins 0 and 2 (actually pin classes) of the module now connect to net bundles which typically contain many individual nets. The power and ground pins 1 and 3 also connect to net bundles which happen to be singletons[6] containing only one individual net.

3.1.14 Creation of Net Bundles

A circuit to be verified by GRASP typically consists of primitive modules connected together via singleton net bundles (also called individual nets). The circuit to be verified

[6] A set with only one element in it is called a singleton.

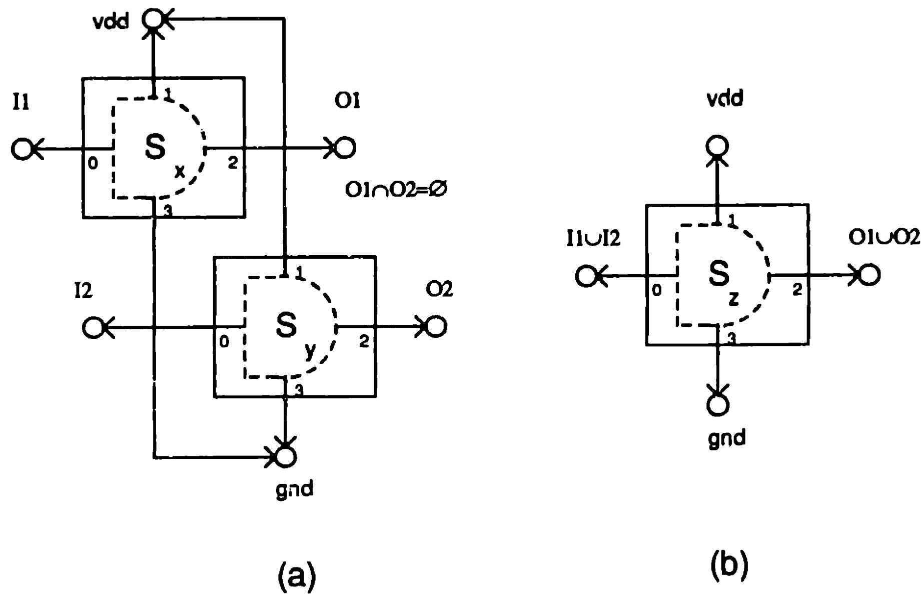


FIGURE 3-22: Net Bundle Creation

can be described using these simpler building blocks because the problems related to equation 3.6 occur only when large composite modules are created. It is during the process of creating new composite modules that new net bundles are created in terms of the existing net bundles. The new net bundles are typically the set union, set intersection and set subtraction of existing net bundles. For example, in figure 3-22 assuming $O_1 \cap O_2 = \emptyset$ the two static block modules of figure 3-22 (a) can be combined into the static block module of figure 3-22 (b). Two new net bundles are created during the reduction of the network of figure 3-22 (a) into the static block module of figure 3-22 (b). They are $I_3 = I_1 \cup I_2$ and $O_3 = O_1 \cup O_2$. A better example of the creation of new net bundles is described later in section 4.1.

3.1.15 Conditions on Net Bundles

In the previous section the reduction of figure 3-22 (a) is legal only if $O_1 \cap O_2 = \emptyset$. This is because the output pins (pins 2) of the two module blocks x and y cannot have any individual nets connected to the output of both x and y. Thus before performing the reduction it is necessary to check that $O_1 \cap O_2 = \emptyset$. This condition on the two nets O_1 and O_2 can be specified in the productions.

3.1.16 The Disjoint Network Problem

It is rarely the case that the set of nets connected to one particular pin class of a given module happens to be the same set of nets connected to another pin class of another

module. Therefore modules rarely *share* net bundles. As *larger* modules and *larger* net bundles get created, the chances of two modules being connected to the same net bundle diminishes. Many of the modules in the circuit end up being connected to their own set of net bundles which are not shared with other modules and the circuit begins to *appear* disconnected.

For example, the circuit of figure 3-23 (a) contains three static gates. After a few reductions the resulting circuit may look like figure 3-23 (b)[7]. The connected circuit of figure 3-23 (a) is transformed into a *seemingly* disconnected circuit in figure 3-23 (b).

In order for a grammar G to be parsed effectively, the network on the RHS of each mapping in M_G must be connected. This is because finding the candidate network to be reduced is an operation performed on a locally connected neighborhood of the circuit (as described in section 3.1.7). The parser moves from module to connected module in search of a network to be reduced. When the network appears disconnected from module S_x, as in figure 3-23 (b), the parser cannot access any other modules because S_x does not share any nets with other modules.

In order for the parser to be able to access other nets besides those connected to S_x and be selective in that only *relevant* nets be examined, operations to move from one net bundle to other associated net bundles must be provided. Associated net bundles of a given net bundle η may be any other net bundles which have nets in common with η.

In the implementation of the GRASP parser, given a net bundle η three operations for accessing other nets from η are provided. These three operations have been shown to be sufficient for describing common methodologies such as NORA, static CMOS etc.. They are:

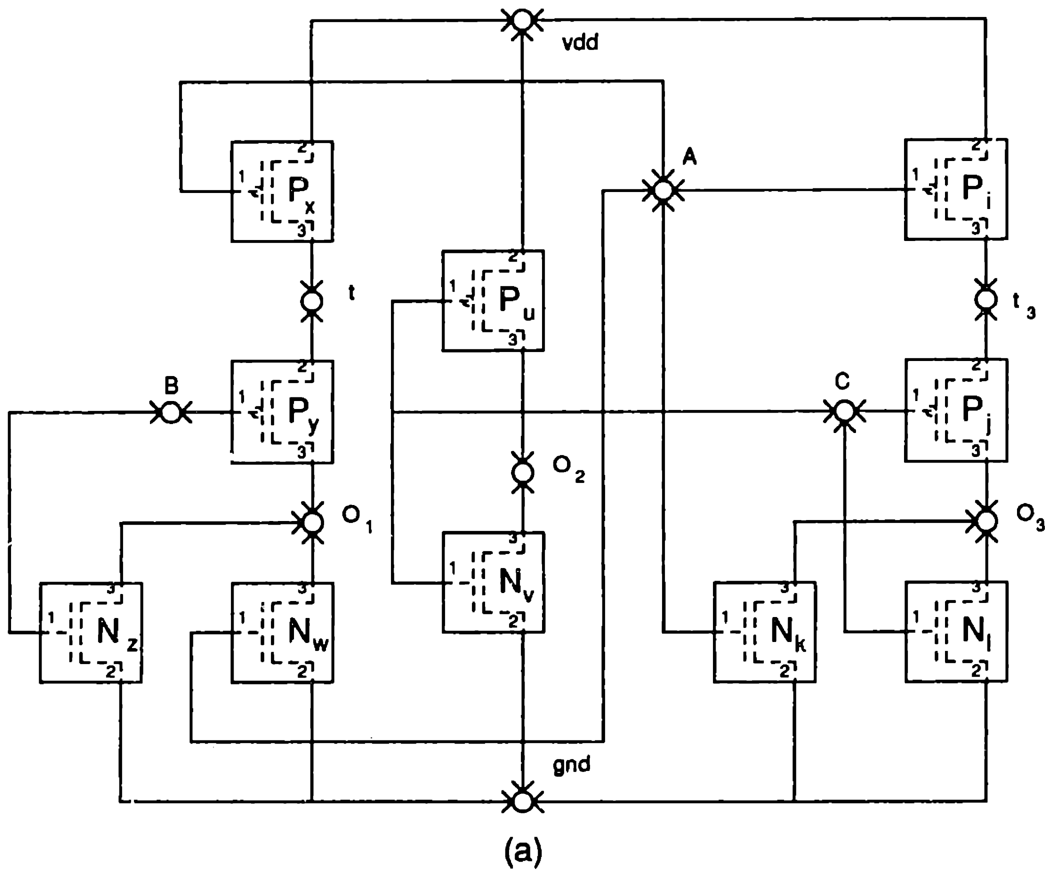
1. Yield the set of net bundles contained in η. This set is referred to as the set of *inferior nets* of η.

2. Yield the set of net bundles which contain η. This set is referred to as the set of *superior nets* of η.

3. Yield the set of net bundles which have at least one individual net in common with η. This set is referred to as the set of *adjacent nets* of η.

These operations yield only the net bundles that are currently connected to pins of existing modules. For example, the first operation will not yield net bundles that were connected to modules which no longer exist.

[7] For explanatory purposes the connections to power and ground are ignored in figure 3-23 (b).



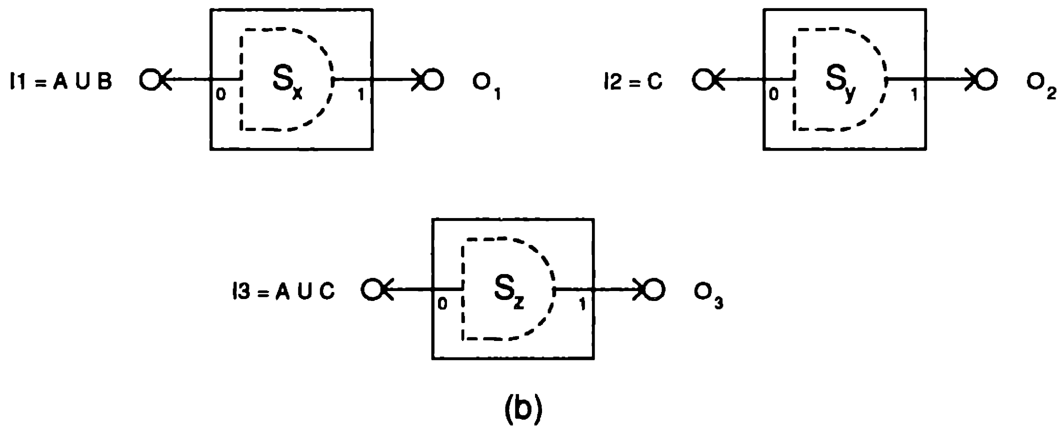


FIGURE 3-23: Disjoint Network Problem

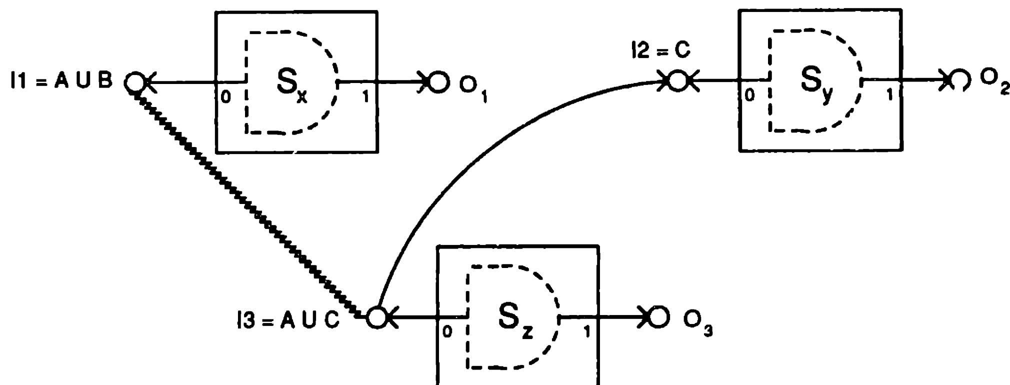


FIGURE 3-24: Relations between Nets

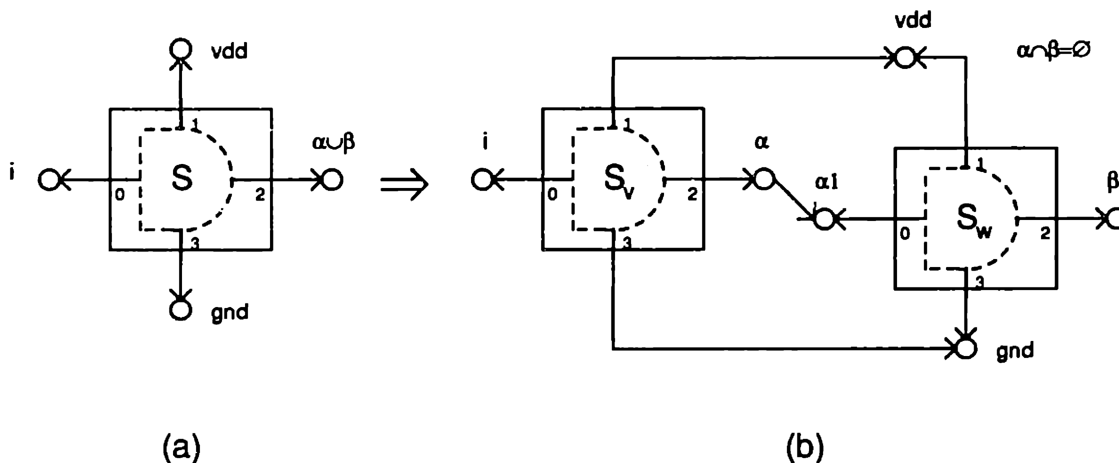


FIGURE 3-25: Complex Circuit Mapping

In a pictorial representation of a network, a small arrow is drawn from each net bundle to its *inferior* bundles. A *wiggly* line between two nets signifies that they are *adjacent.* Figure 3-24 shows the relations between the nets for the circuit of figure 3-23 (b).

For productions that operate on circuits consisting mainly of composite modules and non singleton net bundles, the RHS of the mapping might look like figure 3-25 (b) and the LHS like figure 3-25 (a). The parser uses these relationships between net bundles and the three operations described above to move from module to module in search of the network of figure 3-25. The relations between nets appear implicitly as *conditions* (see section 3.1.15) on the net bundles. Before reducing the network the parser checks to see that all the relations between nets are verified.

3.1.17 Reducibility Condition Revisited

If a non singleton net bundle η is an internal net for a network $\mathcal{N}$ to be reduced, it is not sufficient that η be connected only to modules in $\mathcal{N}$. It is possible that η has an *adjacent net* which connects to modules outside $\mathcal{N}$. The reducibility condition for

net bundles follows the *spirit* of the reducibility condition defined in section 3.1.5. The individual nets in η that are not in one of the external nets of $\mathcal{N}$ cannot be in a net connected to a module not in $\mathcal{N}$.

Formally stated, if $\mathcal{E}$ is the set of external nets for network $\mathcal{N}$, and η' is a net connected to a module not in $\mathcal{N}$, then:

$$\eta \cap \eta' \subseteq \bigcup_{\eta_i \in \mathcal{E}} \eta_i \tag{3.7}$$

In the implementation of the parser a stronger condition which requires that η have no adjacent nets connected to modules not in $\mathcal{N}$, is used. This is equivalent to replacing $\bigcup_{\eta_i \in \mathcal{E}} \eta_i$ with $\emptyset$ in equation 3.7 which forces $\eta \cap \eta' = \emptyset$ which means that no such η' can exist.

3.2 Behavioral Verification

Overview

Behavioral verification is the process of proving or disproving that the actual behavior of a circuit matches a user supplied specification of its intended behavior. In [1] the techniques described in this chapter are extended to perform behavioral verification. Through the use of parsing techniques, the behavioral verification process is facilitated and verification speed is increased.

Relation to GRASP

In [1] the Denotational Semantics [43], [36] paradigm is used to perform behavioral verification. During behavioral verification in [1] a parse tree for the circuit is generated using GRASP. A single grammar whose range space is not restricted to any particular design methodology is used to parse the schematic. The range space of this grammar covers most circuits of practical value. In the same way as the parse tree of a program text captures the structure of underlying algorithm implemented by the program, the parse tree of the schematic captures the structure of computational process implemented by the schematic.

The behavior of a composite module M (nonterminal) in the parse tree can be computed in terms of the behavior of the modules in the network N_M whose reduction produced M. The modules of N_M appear as children of M in the parse tree. The behavior of the modules in N_M can in turn be computed in terms of their children in the parse tree. The behavior of the entire schematic can be determined by recursively computing

the behavior of the root of the parse tree. Finally, the derived behavior is matched with a procedural specification of the intended behavior of the circuit.

Benefits of the Method

By operating on the parse tree, which is a structured representation of the schematic, instead of directly on the collection of interconnected modules in the schematic, the behavioral verification process is substantially facilitated. The parse tree organizes information in the schematic into a format suitable for extracting its behavior. It provides a hierarchical specification of the schematic in which each composite module in the hierarchy is derived from a (small) finite set of *familiar* circuit configurations. For each such configuration a procedure for computing the behavior of the composite module in terms of the behavior of its constituent modules is provided. Using one of these procedures, the behavior of any composite module can be computed in terms of the behavior of its children in the parse tree.

All behavioral verification techniques in one form or another extract the behavior of the schematic from its structure. By separating the structure finding phase from the rest of the verification the verification algorithm is partitioned along a natural boundary. By casting the structure finding phase into grammatical parsing using a deterministic context free circuit grammar, efficient parsing techniques for deterministic context free grammars described in section 5.1 can be applied.

Implementation

A computer program called SEMANTICIST [1] which implements the behavioral verification techniques described in this section has been written and tested on various circuits. SEMANTICIST uses the GRASP parser to create the parse tree, then recursively extracts the behavior of the circuit from the parse tree as described and finally compares this behavior with a user specification of the intended behavior.

4

Examples

This chapter provides examples of grammars for common design styles, parse sequences on actual circuits and how circuit grammars catch various simple design errors. These examples are especially useful as they attempt to provide the reader with an intuitive feel of how the formalisms described in the previous chapters apply to real circuits. Many of the formalisms defined in the previous chapters are but one of the many possible ways of *packaging* the problem of design style verification into grammatical parsing. It is necessary to have an intuitive understanding of how grammatical parsing catches design errors and the impact of the formalisms of the previous chapter to grasp the insights required for further extending these techniques.

Section 4.1 gives an example of a grammar for classical CMOS capable of reducing n and p channel transistors into classical CMOS gates. Section 4.2.1 describes a grammar for domino gates. Section 4.2.2 gives an example of a grammar for the two phase CMOS clocking methodology which is capable of verifying a *transistor level* description of classical CMOS gates, dynamic gates (e.g. domino) and latches combined in accordance with the CMOS two phase clocking requirements. Finally section 4.3 concludes with examples of how grammars catch various simple design errors such as open circuits, short circuits and illegal loops.

4.1 A Classical CMOS Grammar

In this thesis, classical CMOS gates are defined as static CMOS gates in which the pulldown structure is formed by series-parallel connections of n-channel transistors and the pullup structure is the DeMorgan complement of the pulldown structure. In this section, a grammar capable of parsing a netlist of transistors into a classical CMOS gate is described. This grammar can be used as a *component* of a grammar in a more complex

circuit methodology that incorporates classical CMOS gates. It also provides good examples of presence condition modules and net bundles. Similar grammars which reduce transistors into modules have been designed for static NMOS, domino logic, precharged NMOS, latches etc.

Module C_0 in figure 4-1 (a1) is of type *C-blk* (short for *Complementary Block*). For modules of this type, the internals of the module are such that the internal electrical path between pins 1 and 2 is composed of series-parallel connections of p-channel transistors. The electrical path between pins 3 and 4 is composed of series-parallel connections of n-channel transistors and is the conjugate of the p-channel path between pins 1 and 2. Pin 0 connects to the net bundle corresponding to the connections to the gates of all the transistors on both paths.

A *C-blk* module can be created by reducing a p and an n-channel transistor as shown in figure 4-1 (a1). Two *C-blk* modules can then be composed into a new *C-blk* module by connecting the two p-channel paths in series and the n-channel paths in parallel as shown in figure 4-1 (a2). The resulting module C_3 in figure 4-1 (a2) is also of type *C-blk* and is such that its p-path (between pins 1 and 2) is the series combination of the two p-paths of C_1 and C_2 and the n-path (between pins 3 and 4) is the parallel combination of the two n-paths of C_1 and C_2. The input pin (pin 0) of C_3 now points to the union of the two bundles i_1 and i_2.

Figure 4-2 shows the two productions of figure 4-1 as well as all the other productions in the classical CMOS grammar. A companion production of the production of figure 4-1 (a2) or figure 4-2 (a2) or which composes the p-paths in parallel and the n-paths in series is shown in figure 4-2 (a3). The productions in figures 4-2 (b1), (b2) and (b3) are similar to the productions in figures 4-2 (a1), (a2) and (a3) except that they have a module of type *G-blk* (short for *Gate Block*) on their LHS. Modules of type *G-blk* are almost identical to modules of type *C-blk*. The electrical path between pins 1 and 2 is composed of series-parallel connections of p-channel transistors and is the conjugate of the n-channel path between pins 2 and 3. Pin 0 is again the pin corresponding to the gates of the transistors in both paths. Modules of type *G-blk* differ from modules of type *C-blk* only in that they have 4 pins instead of 5 because both the p-channel and the n-channel paths *share* pin 2. A final production shown in figure 4-2 (c) with a *presence condition* module[1] *P0* of type *power*, similar to the production in figure 3-16 reduces a *G-blk* into the classical CMOS gate *S0*.

It can be proved that the range space of the grammar defined by the productions in

[1] The *presence condition* modules for productions (a1), (a2), (a3) and (b1), (b2), (b3) are not shown.

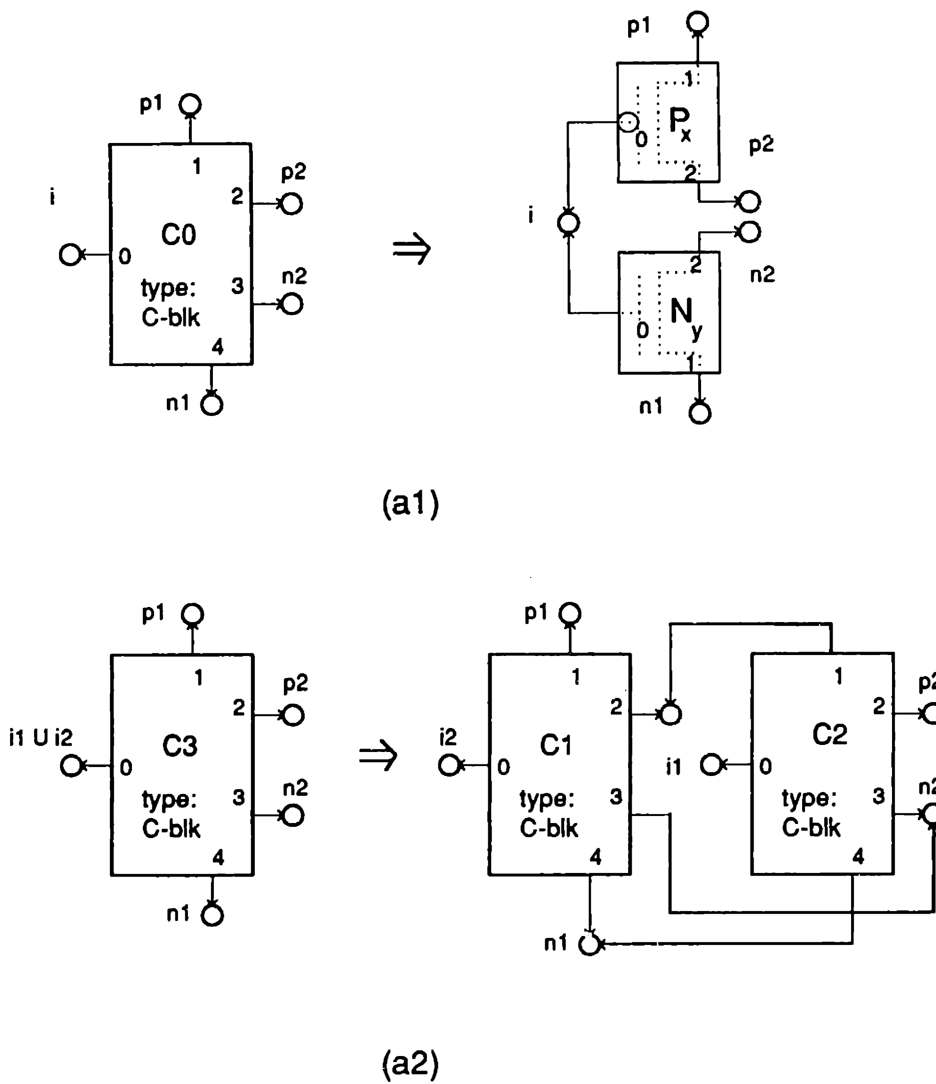


FIGURE 4-1: Classical CMOS Productions

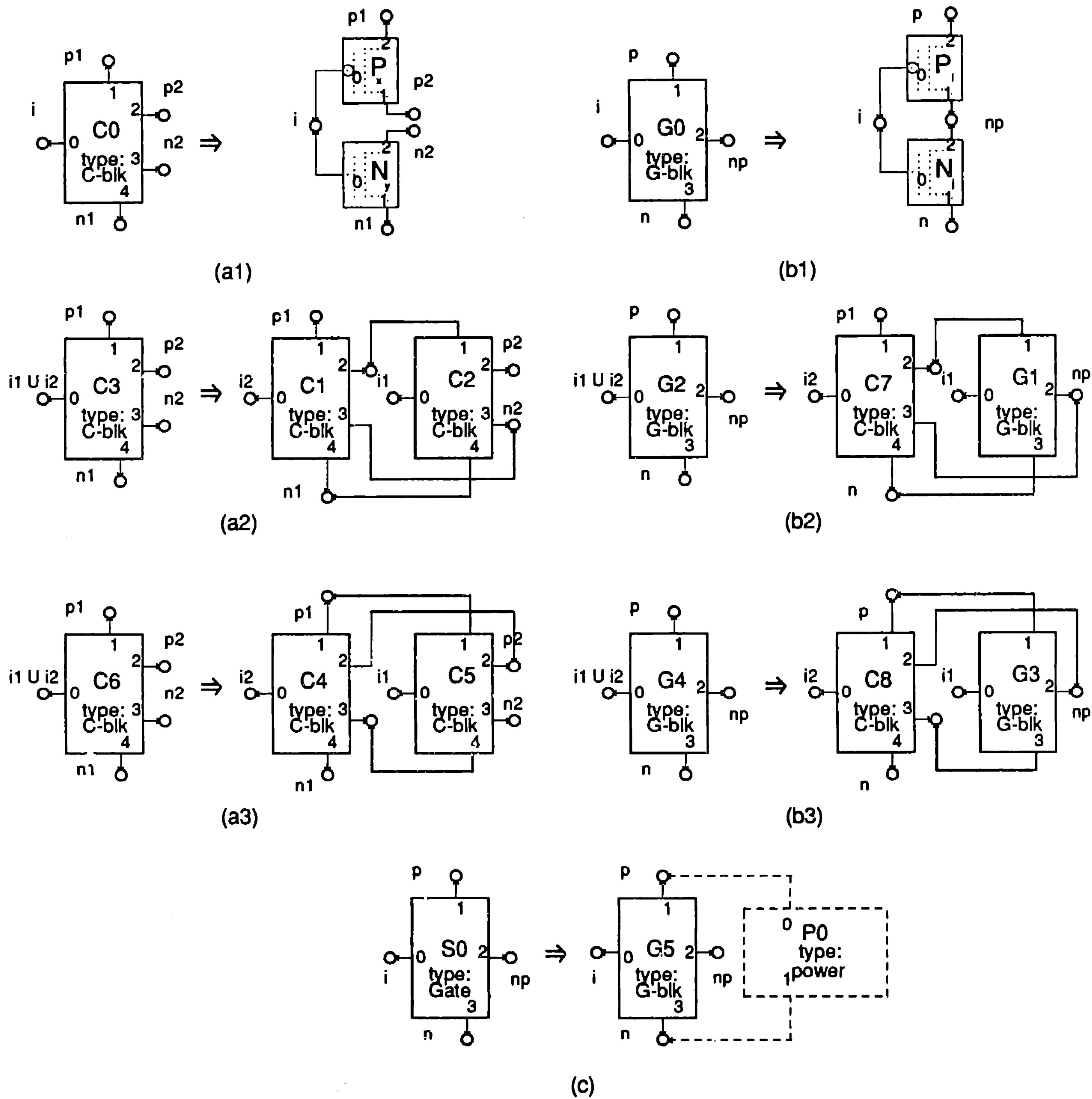


FIGURE 4-2: Classical CMOS Grammar

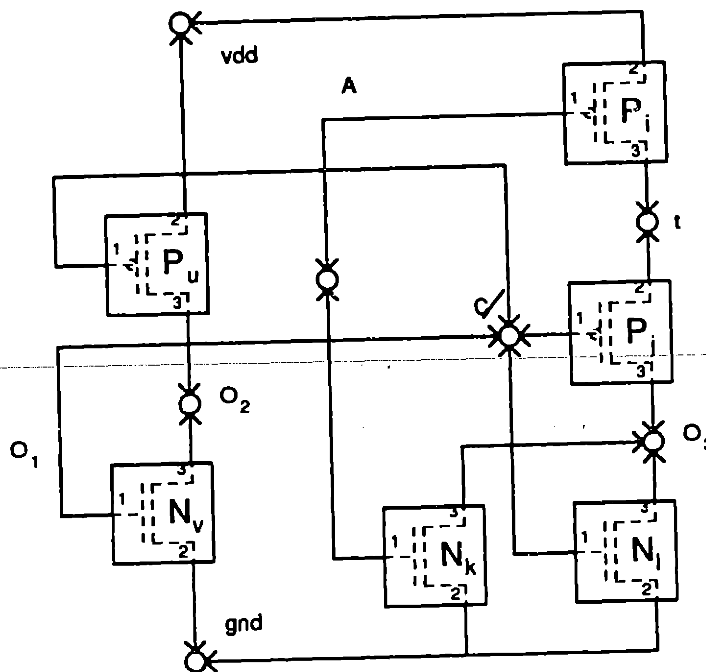


FIGURE 4-3: Transistors in Different CMOS Gates

figure 4-2 exactly describes the set of complementary CMOS gates characterized in [53]. However proving that the range space of a grammar coincides with an independently characterized set of circuits is beyond the scope of this thesis.

Condition Modules

In figure 4-1 the condition modules for productions (a1), (a2), (a3) and (b1), (b2), (b3) are not shown. Production (a1) requires one of several presence modules in order to be deterministically applicable. Consider the case of figure 4-3. The network formed by modules v and j (or modules u and l) is isomorphic to the RHS of production (a1) in figure 4-1. The intent of this production is to group corresponding n and p-channel transistors within a static gate. However transistors v and j belong to different gates. Performing the reduction of production (a1) on transistors v and j leads to a circuit which cannot be reduced into two classical CMOS gates although figure 4-3 does indeed consist of two well-formed classical CMOS gates.

Because net *np* in figure 4-2 is connected to the drain of both the n and p-channel transistors these transistors necessarily belong to the same CMOS gate. Therefore production (b1) does not require a presence module to ensure that the transistors belong to the same gate. It can be shown that the RHS of production (b1) appears in every classical CMOS gate circuit.

In order to ensure that the n and p-channel transistors for production (a1) belong to

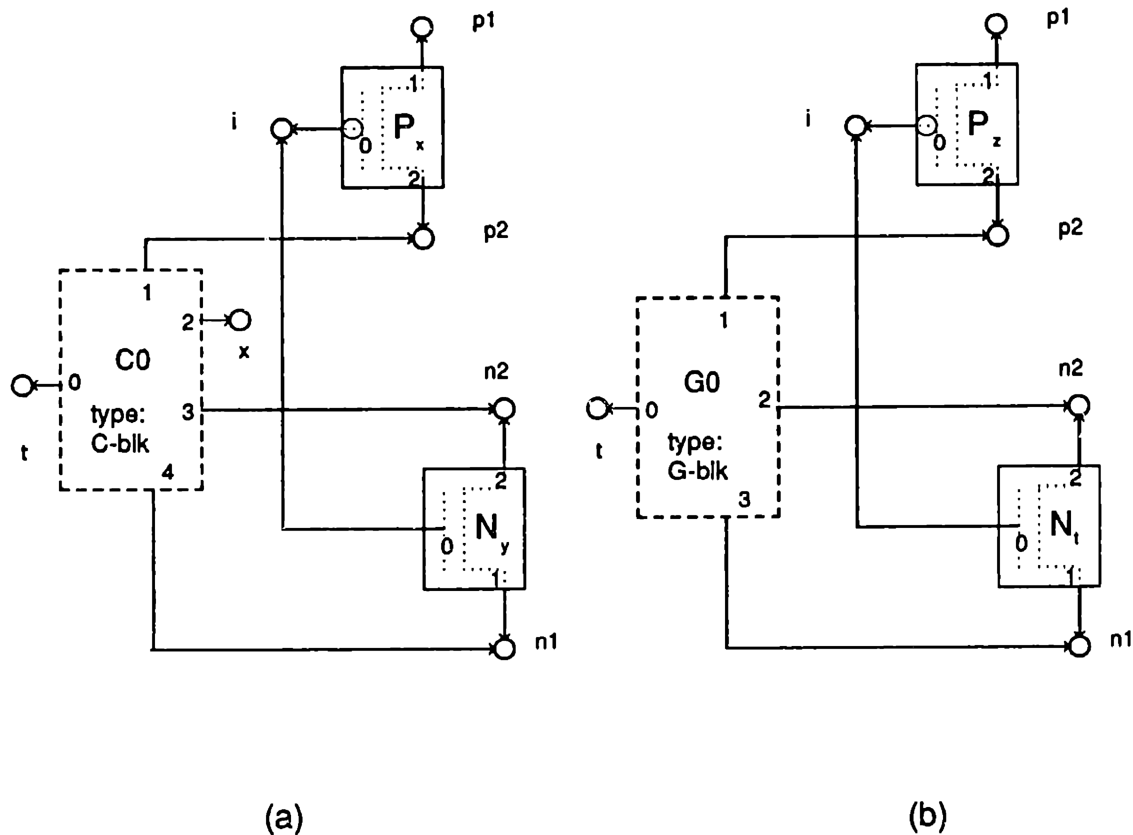


FIGURE 4-4: CMOS Presence Condition Modules

the same gate, the presence condition modules of figures 4-4 (a) and (b) are introduced. The presence of the augmented network of figure 4-4 (a) anywhere in the circuit guarantees that the n and p-channel transistors belong to the same gate. A similar augmented network with a *G-blk* condition module shown in figure 4-4 (b) is also required. Finally, *companion* augmented networks for both figures 4-4 (a) and (b) with the p-channel paths in series (instead of in parallel) are needed.

Parse Sequence

Figure 4-5 shows the evolution of a CMOS NOR gate circuit during parsing. In the first step, production (a1) is applied to reduce transistors y and z of figure 4-5 (a) yielding the circuit of figure 4-5 (b). Then production (b1) is applied to reduce transistors x and t in figure 4-5 (b) yielding figure 4-5 (c). During this reduction module G0 is used as a presence condition module in a fashion analogous to figure 4-4 (a). Production (b2) is then used to reduce C0 and G0 in figure 4-5 (c) yielding the circuit of figure 4-5 (d). Notice that during this reduction a new net bundle $C = A \cup B$ is created. Finally, if nets *vdd* and *gnd* are appropriately connected to the power supply module then module G1 can be reduced to a gate module via the production of figure 4-2.

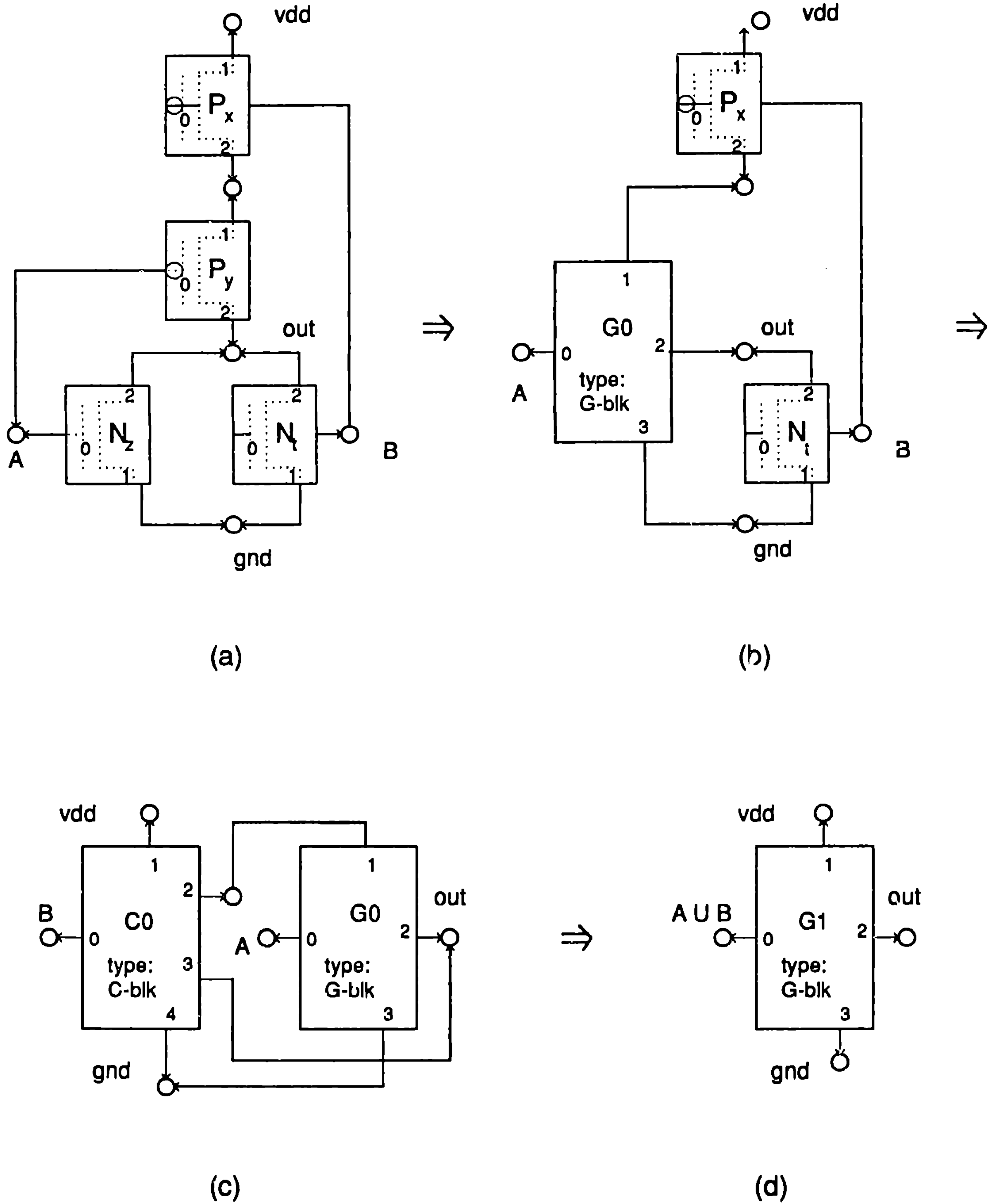


FIGURE 4-5: Parse Sequence

4.2 Two Phase Clocking Methodology Grammar

In this section a grammar for verifying a CMOS two phase clocking methodology is described. The grammar combines latches with precharged and static gates in accordance with the CMOS two phase clocking rules. These modules are assumed to have been created by (sub) grammars (such as the grammar of section 4.1) capable of reducing netlists of transistors into modules. This is an example of how grammars can be combined to yield new grammars.

First a grammar for verifying domino logic [53] is described. Then a grammar for the two phase clocking methodology similar to NORA [18] that builds on the domino grammar is introduced. For simplicity's sake only p-channel precharge and n-channel evaluate domino gates are used.

4.2.1 Domino Grammar Productions

The grammar for domino gates described in this section implements just one of the various strategies for checking domino gates. Each of these strategies exhibits different trade-offs between parsing efficiency, how early errors are detected, conceptual simplicity and number of productions. A grammar that exercises several of the concepts described in the previous sections is chosen.

The domino grammar described in this section verifies that the inputs to all domino gates are connected to the output of an inverter driven by the output of another domino gate or a latch. For the circuit to function properly, the domino gates and latches must be connected to the proper clocks. For example, a domino gate precharged on ϕ_1 and evaluating on ϕ_2 can only be fed by the output of an inverter driven by a domino gate connected to the same clocks or a latch open on ϕ_1. This is easily verified by making the clock nets $\phi_1, \phi_2, \overline{\phi_1}$ and $\overline{\phi_2}$ and possibly the clock module (used as a presence condition module) appear in the productions. In this section for simplicity's sake it will be assumed that all modules are consistently connected to the clocks and hence all the clock connections will be omitted. This will greatly simplify the figures and focus attention on the pertinent parts of the grammar. For the same reason, power and ground connections are also omitted.

Module y on the RHS of figure 4-6 is of type *precharged gate.* These modules consist of a p-channel precharging transistor capable of charging the output high and a pull down path to ground consisting of a series-parallel path of n-channel transistors. Pin 0 is the input to the precharged gate and pin 1 is the output. The connections to the clocks, power and ground have been omitted. These precharged gate modules are generated

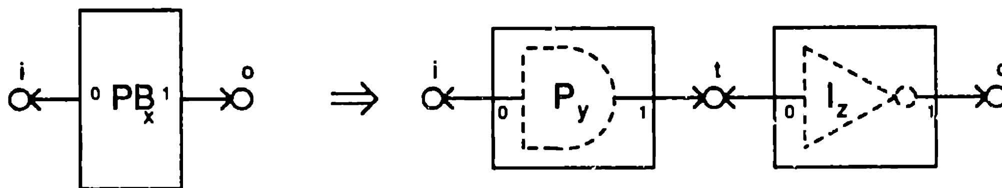


FIGURE 4-6: Production 1

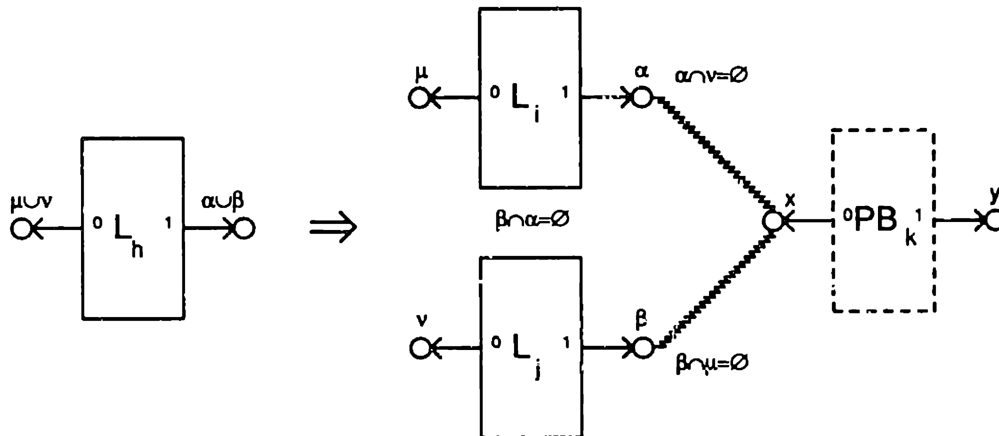


FIGURE 4-7: Production 2

by a grammar capable of recognizing precharged gates in a transistor level netlist and reducing them to a precharged gate module.

Module x on the LHS of figure 4-6 is of type *precharged block* (PB for short). A precharged block can be derived from a precharged gate and an inverter via production 1 shown in figure 4-6. The input to a precharged block such as module x can be taken from the output of another precharged block or the output of a latch module. The productions 3, 4 and 5 (described later in this section) combine precharged blocks into *larger* precharged blocks which exhibit the same properties.

Modules h, i and j in figure 4-7 are parallel combinations of inverting latches (referred to simply as latches). The transistor level representation for an inverting latch (which is considered to be a *degenerate* parallel combination of inverting latches) has been shown in figure 3-6. The connections to the clocks, power and ground have been omitted. Pin 0 represents the input of the latch and pin 1 the output. The latches can be combined in parallel to form new composite latches.

Production 2 shown in figure 4-7 combines in parallel two latches which have some of their outputs connected to the input of a common PB module. The *wiggly* line between nets β and x and nets ν and x signifies that each of these pairs of net bundles are *adjacent* (see section 3.1.16) which means that they have some individual nets in common. The strategy of productions 1, 3, 4 and 5 (described later) is to group all precharged gates and inverters into one *large* PB module. Assuming the strategy works, only one PB module

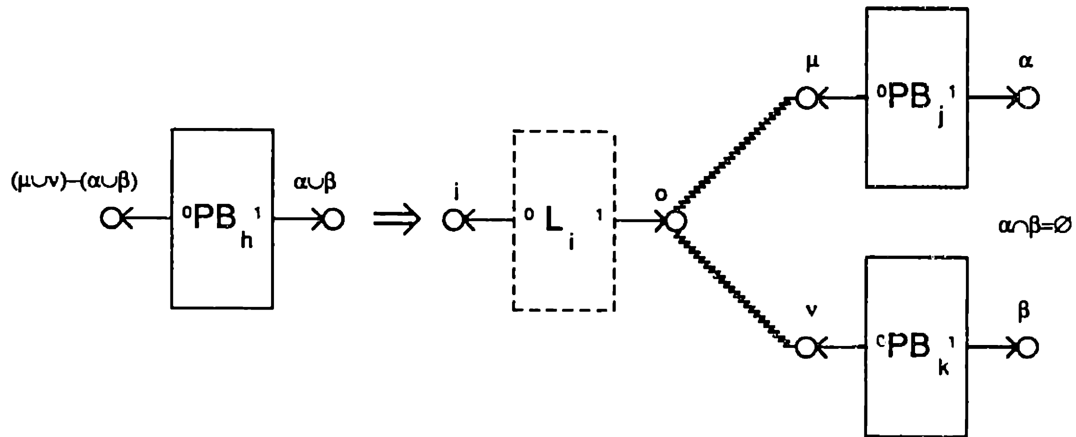


FIGURE 4-8: Production 3

will exist in the end and hence the outputs of all the latches will be adjacent to the inputs of the same (*large*) PB module. Production 2 will therefore combine all latches into one large latch module[2].

Before combining the two latches in production 2 it is necessary to verify that $\alpha \cap \beta = \emptyset$ (two outputs cannot connect together) as depicted in figure 4-7 so that the internals of the new latch will be well-formed. Because *feedback* from the output of a latch to the input of any other latch (connected to the same clocks) is disallowed it is also necessary to verify that $\alpha \cap \nu = \emptyset$ and $\beta \cap \mu = \emptyset$.

Production 3 shown in figure 4-8 is similar in spirit to production 2. It's purpose is to combine PB modules. As in production 2 it must be the case that $\alpha \cap \beta = \emptyset$. It is however legal for the outputs of module j to be the inputs of module k and vice versa (in this grammar signal feedback loops in domino logic are allowed). These nets are *driven* nets and hence must be removed from the inputs of the new composite PB module. Therefore the input net bundle of module h is $(\nu - \alpha) \cup (\mu - \beta)$ (which is also equal to $(\mu \cup \nu) - (\alpha \cup \beta)$ since $\alpha \cap \beta = \emptyset$).

Production 4 shown in figure 4-9 is identical to production 3 except that instead of a latch module a PB module is used as a condition module to group PB modules together.

Production 5 shown in figure 4-10 combines two PB modules in series. All the inputs of module z must be connected to the outputs of the same PB module y. Therefore net β must be superior to net μ (see section 3.1.16) which means that all the individual nets in μ must be contained in β. Feedback from the output of module z to the input of y is allowed but the input of the new PB module must be $\alpha - \nu$ so it does not contain any individual nets that are connected to some *driven* output.

Production 6 shown in figure 4-11 combines the latch and PB module into a $\phi_i$section

[2]Except those latches whose output is not connected to anything.

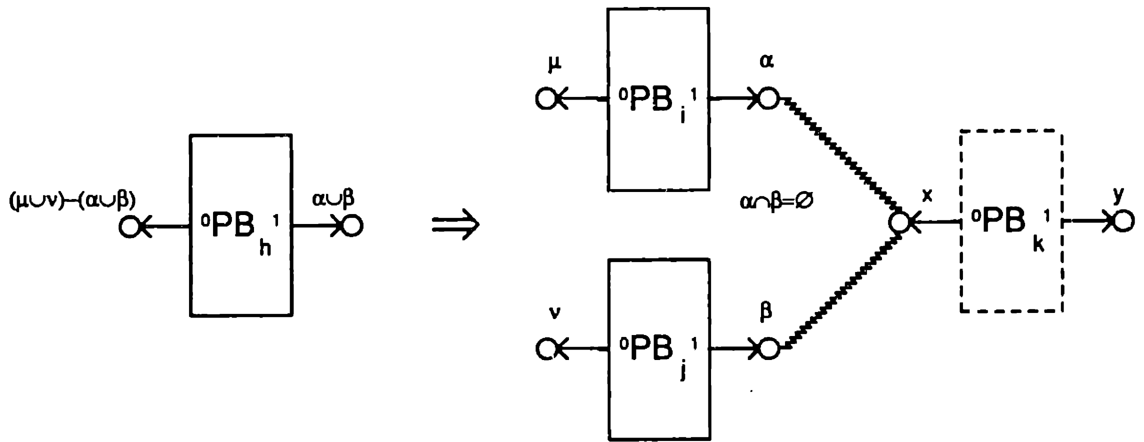


FIGURE 4-9: Production 4

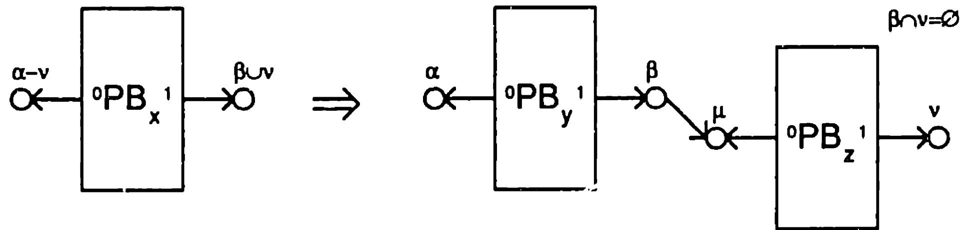


FIGURE 4-10: Production 5

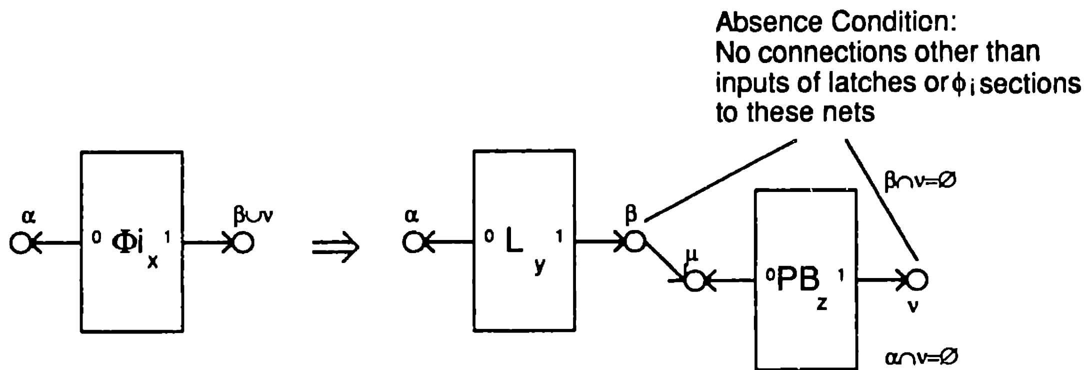


FIGURE 4-11: Production 6

module which is a basic building block of the two-phase clocking methodology described in section 4.2.2. Since the output of the PB module cannot feed the inputs to the latch, $\alpha \cap \nu = \emptyset$. Before this reduction is performed it must be ensured that no more PB or latch modules remain to be combined via productions 3, 4 or 5. For this purpose absence conditions are used in production 6[3].

The outputs of a $\phi_i$section block created by production 6 can connect only to the inputs of a latch or another $\phi_i$section block (connected to the alternate phase clocks) as described in section 4.2.2. Hence if the latch and PB modules are *complete* the nets adjacent to nets β or ν can connect only to the inputs of a latch or $\phi_i$section. Therefore an absence condition requiring that there be no connections to adjacent nets of β and ν by pins other than the input pins of a latch or a $\phi_i$section is required to guarantee that both the latch and PB module cannot be reduced into *larger* modules of the same type. Because of these absence conditions additional productions are required as described later in section 5.1.5.

It is important to note that the domino grammar accepts more than just series parallel combinations of PB modules. The domino grammar will reduce the circuit of figure 4-12 which cannot be expressed in terms of series parallel combinations of PB modules into a single PB module.

4.2.2 Two Phase Clocking Methodology Grammar Productions

The two phase clocking methodology grammar is similar to the domino grammar except that the $\phi_i$section blocks are more complex and can themselves be composed into

[3]In this case the absence condition can be avoided by rearranging the productions.

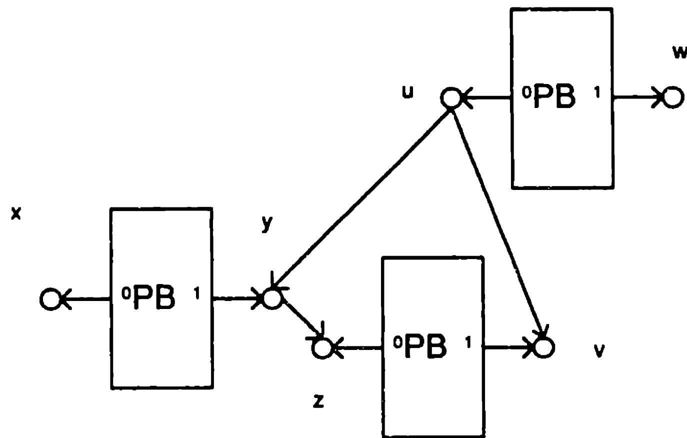


FIGURE 4-12: Non Series Parallel Domino Blocks

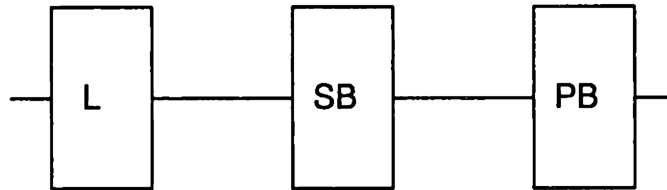


FIGURE 4-13: $\phi_i$section Block Diagram

larger ϕsection blocks. First the new, slightly more complex $\phi_i$section is described, then productions for the ϕsection are described.

Building $\phi_i$section Blocks

Each $\phi_i$section block consists of a latch L, a static block SB and a precharged block PB as shown by the block diagram of figure 4-13[4]. The static block SB consists of combinations of static gates. The productions for building the SB blocks are such that the signal flow within each SB block is guaranteed not to have any loops. This is achieved by using the *loop detecting* grammatical productions described in section 4.3.3 which produce a combined latch static-block module of type LSB in which there is no signal loop path through static gates. The net effect of these productions is to yield a LSB module equivalent to that generated by the production of figure 4-14. The clock nets are not shown in the figures of this subsection. The LSB and PB modules are then combined to yield a $\phi_i$section block as shown in figure 4-15. The productions of figures 4-14 and 4-15 are related to the domino production of figure 4-11. Absence conditions associated with both productions are needed but are not shown.

Care must be taken in defining the grammar productions for classical CMOS so that inverters which are needed in the domino grammar do not get reduced into static gates.

[4]Another SB block can also be added after PB block.

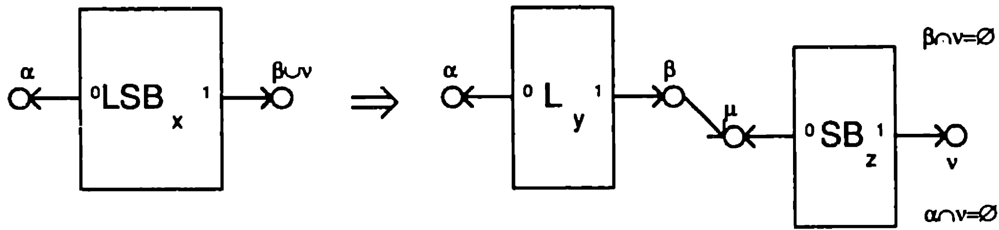


FIGURE 4-14: LSB Production

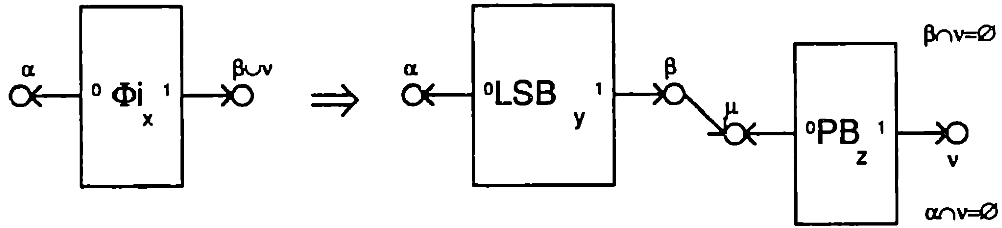


FIGURE 4-15: Complex $\phi_i$section Production

In a more general version of the domino grammar, NAND and NOR gates as well as inverters (which can be thought of as degenerate NAND and NOR gates) can be used between precharged gates. A slightly more complex classical CMOS grammar could easily be made to differentiate between NAND, NOR, inverter gates and other static gates.

Building ϕsection Blocks

A $\phi_i$section precharged on ϕ_2 and evaluating on ϕ_1 is called a $\phi_1$section. The same block precharged on ϕ_1 and evaluating on ϕ_2 is called a $\phi_2$section. In the figures of this subsection the clocks ϕ_1 and ϕ_2 are shown to differentiate between ϕ_1 and ϕ_2 sections. The complement of these clocks $\overline{\phi_1}$ and $\overline{\phi_2}$ and power and ground are not shown. Figure 4-16 shows how a $\phi_1$section and $\phi_2$section block can be combined to yield a ϕsection block. Absence conditions which guarantee that each $\phi_i$section is *complete* (similar to the absence conditions of section 4.2.1 figure 4-11) before this production is applied are needed but are not shown.

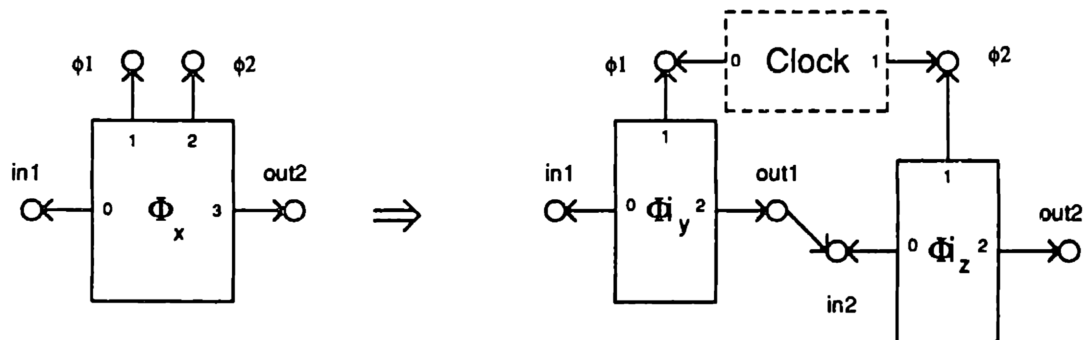


FIGURE 4-16: ϕsection Production

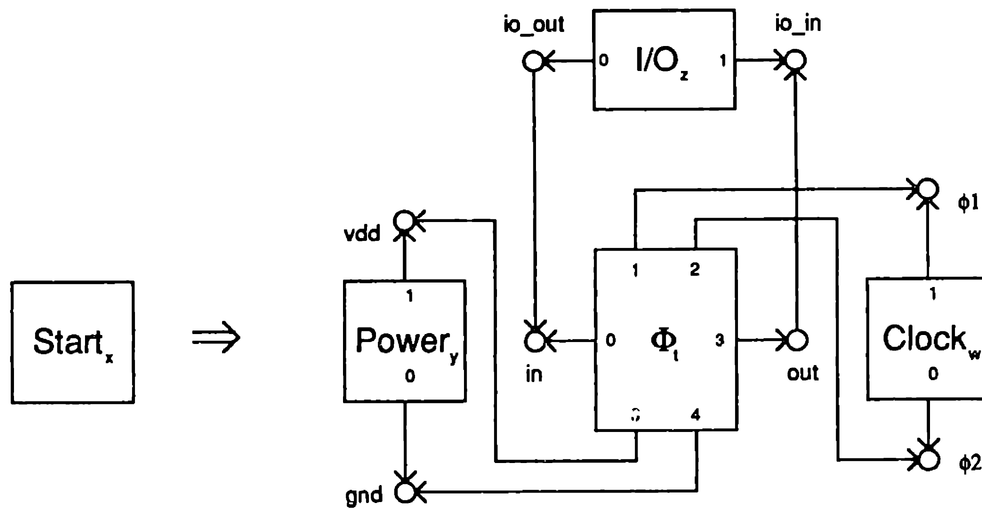


FIGURE 4-17: Start Symbol Production

The ϕsection blocks can themselves be composed into *larger* ϕsection blocks using productions similar to those used to combine PB blocks in figures 4-9 and 4-10 (which combine PB blocks not necessarily in a series parallel fashion). Finally the production of figure 4-17 which involves the I/O waveform generator module which simulates the inputs and outputs to the circuit is used to yield the start symbol of the grammar. The power and ground nets are shown in this figure.

4.3 Detecting Common Errors

Unlike the techniques in [26], [30] and [52], GRASP's verification strategy checks for the *right* structure rather than any particular type of error. The set of context free circuit grammars is not closed under range space intersection. Let G_α (respectively G_β), be a context free grammar with range space L_α (respectively L_β) for design error type α (respectively β). The grammar $G_{\alpha,\beta}$ whose range space is $L_{\alpha,\beta} = L_\alpha \cap L_\beta$ and is capable of simultaneously verifying both design error types α and β, is not necessarily context free. Thus for comparing the capabilities of grammatical parsing with that of existing techniques, it is not meaningful to show that context free circuit grammars can be designed for catching any specific error type. This is because even if that were feasible, it may not be possible to construct a context free grammar which simultaneously verifies all of the error types[5].

When the structural constraints can simultaneously be expressed in terms of a hierarchy of permissible module configurations (as often occurs with common design method-

[5]It is of course possible to have a different parser associated with each grammar G_α.

ologies), context free grammars can be used to capture the structural constraints.

Although GRASP's parsing strategy is not centered around catching individual error types, this section attempts to provide some intuition as to the way in which parsing catches a few of the common design errors. Section 4.3.1 (respectively section 4.3.2) describes how, irrespective of the circuit grammar, parsing identifies **any** open (respectively short[6]) circuits that cause an otherwise structurally correct circuit to become ill-formed. Finally section 4.3.3 describes a particular grammar whose sole purpose is checking for illegal loops.

4.3.1 Detecting Open Circuits

Suppose a circuit C obeys methodology M. The effect of creating an open circuit in C by removing some connections between two modules M_1 and M_2 in C is investigated. In particular the difference in behavior of the parser on the two circuits C and C', identical to each other except for the open circuit, is examined.

Since only one module remains at the end of a parse of C there is necessarily a reduction R involving two modules M_1^0 and M_2^0 such that M_1 is in the expanded circuit of M_1^0 and M_2 is in the expanded circuit of M_2^0.

The difference between the two circuits can manifest itself in one of the following two ways during reduction R.

1. Some pin of M_1^0 and some pin of M_2^0 which used to be connected to the same net in C are no longer in C'.

2. Two nets on the RHS of R which were in relation to each other in C (by one of the three relations described in section 3.1.16) are no longer in C'.

Either of these two conditions can prevent R from firing and the open circuit (if it were truly the case that the circuit has become ill-formed), will stop C' from being successfully parsed.

4.3.2 Detecting Short Circuits

Suppose a circuit C obeys methodology M. The effect of creating a short circuit in C by merging the connections of two nets η_1 and η_2 into one net η is investigated[7]. In

[6]This is different from identifying a particular kind of short circuit such as power to ground shorts.
[7]Both η_1 and η_2 are singleton net bundles.

particular the difference in behavior of the parser on the two circuits C and C', identical to each other except for the short circuit, is examined.

Since only one module remains at the end of a parse of C one of the following two situations holds true.

1. There is a reduction which involves both η_1 (or one of its superiors) and η_2 (or one of its superiors).

2. There is a reduction which does not involve η_1 (or one of its superiors) and has as an internal net η_1 (or one of its superiors) or vice versa.

In the first case the short circuit could cause pins that were connected to different nets to be connected to the same net or could cause some of the disjointed relationships (the nil intersection requirements of section 3.1.15) to no longer be satisfied. For each production the set of nets that must be disjointed so that the new composite module is well-formed must be described and hence the short circuit will be caught if it leads to the circuit being ill-formed.

In the second case the short circuit will cause the reducibility condition to no longer be satisfied.

Either of the two conditions can halt the corresponding reduction from being fired and can halt C' from being successfully parsed.

4.3.3 Detecting Loops

Loop detection is often necessary in design style verification. For example, the SB module of section 4.2.2 consists of static gate modules which must be composed in such a way that there are no signal loops which would cause the circuit to oscillate.

A regular grammar [24] can be used to detect loops. The set of regular grammars is a subset of deterministic context free grammars and therefore loop detection can described in terms of our grammatical formulations. During the parse of any circuit using a regular grammar at most one composite module can exist at any given time.

A regular grammar can be used to reduce static gates and latch modules into one combined Latch and Static block LSB of section 4.2.2. This grammar can be used as a component of the NORA grammar. The classical CMOS grammar (which reduces transistors into static gates) as well as several other component grammars are context free so the NORA grammar as a whole will be context free.

Given a LSB module (obtained by reducing a latch and a static gate module) whose internals are such that there are no loops, figure 4-18 incorporates a static gate into

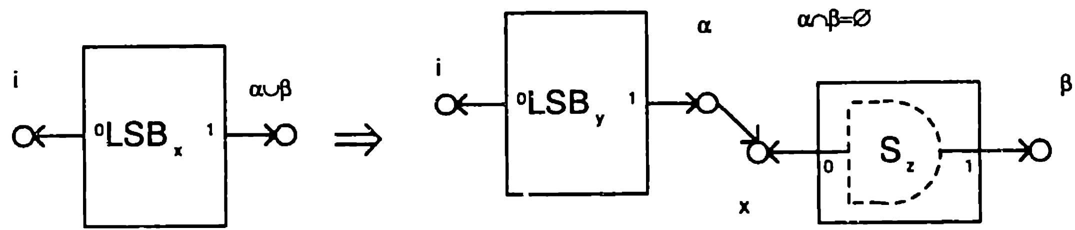


FIGURE 4-18: Loop Checking Production P_{loop}

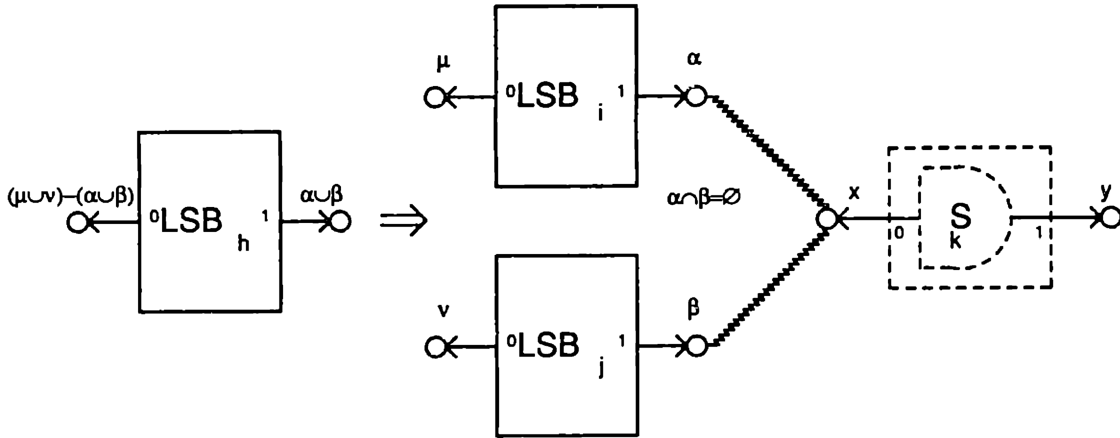


FIGURE 4-19: Parallel LSB Composition

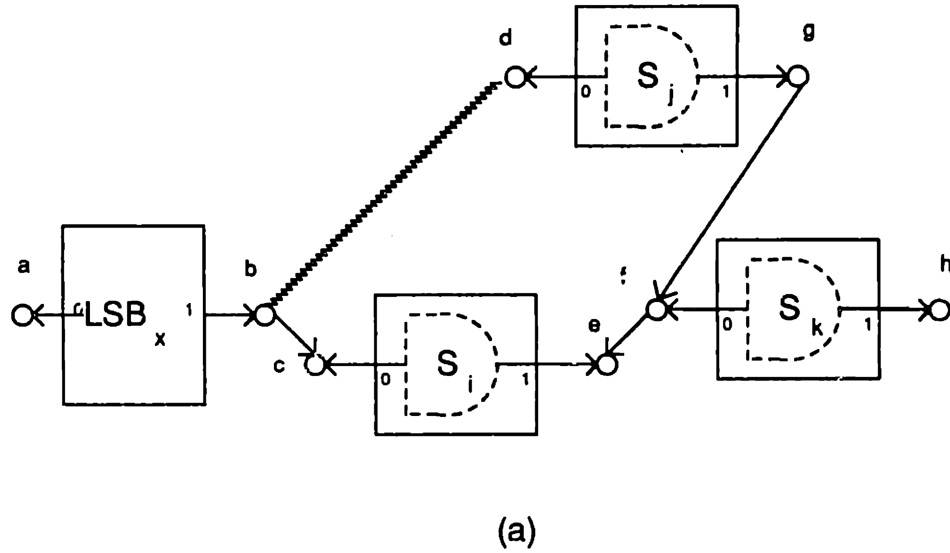
the LSB module. The new LSB module is guaranteed not to have any loops because the static gate is driven solely by the LSB module and does not drive any circuitry in the LSB module.

The grammar induces an implicit (partial) ordering of the static gates. This ordering is such that if the output of a static gate S_x influences an input of a static gate S_y, then $order(S_x) < order(S_y)$. During parsing gates are reduced in order i.e. if the output of S_x influences the input of S_y then S_x is reduced before S_y.

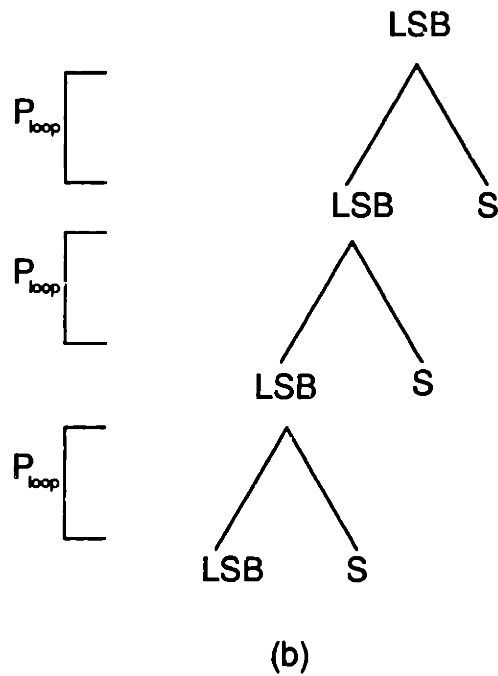
The production of figure 4-19 is used to compose LSB blocks in parallel. This production causes the grammar for building LSB blocks to become non-regular. However there is no induced ordering between a static gate in LSB_1 and a gate in LSB_2 because the production guarantees that no output of LSB_1 influences any static gate of LSB_2 and vice versa. The *part* of the grammar which performs the ordering of the modules however is regular.

Regular grammars can cause the parse tree to become unbalanced. Unbalanced parse trees can become a problem during incremental modifications to the circuit because more than $O(\log n_{mod})$ reductions may have to be undone. A parse tree for the circuit of figure 4-20 (a) is shown in figure 4-20 (b). The height of the tree is greater than the length of the longest signal path through static logic. In practice the length of any such chain is limited to a few static modules to minimize signal delay between latches.

Parse trees for the LSB grammar appear as subtrees of the parse tree for the entire NORA circuit. Since the height of each of these subtrees is limited the parse tree for the entire circuit will not become too unbalanced.



(a)



(b)

FIGURE 4-20: Unbalanced Parse Tree

5

Schematic Verification Algorithm & Implementation

5.1 Event Driven Parsing Algorithm

5.1.1 Overview

The operation of the algorithm for GRASP described in this chapter as well as the algorithms for layout verification and schematic vs. layout comparison described in chapters 7 and 10 is organized into actions that the algorithm must perform but has not yet processed. The execution of an action may result in the creation of new actions that the algorithm must also process. At any given point during verification, the collection of actions that remain to be executed is an unordered set. The actions in this set can be performed in any sequence without changing the outcome of the verification process. Computation time however can depend on the order in which the actions are executed. The actions will henceforth be referred to throughout this thesis as *events* without inferring any associated time at which they occur or when they should be processed. The term *event driven* will be used throughout this thesis for algorithms whose operation is organized around the creation and processing of events.

GRASP uses an *event driven* bottom up parsing algorithm to parse circuits. Parsing effort is expended in an area of a circuit only when a change is detected in that area. An *event queue*[1] is maintained to schedule GRASP's activities. Each event in the queue is associated with a module in the circuit. The *event queue* is initialized when the circuit

[1] A *first created, first processed* event ordering strategy has been found to be computationally efficient on average.

to be verified is read into GRASP by inserting into the queue an event for each module in the circuit.

As parsing proceeds, each event is in turn removed from the queue and serviced. Servicing an event involves firing any applicable reductions in which the module M associated with that event appears either as a module to be reduced or as a *presence condition* module. New events associated with each new composite module created by this process are generated and put on the event queue. The algorithm terminates when the event queue becomes empty.

5.1.2 Servicing an Event

Servicing an event with associated module M requires finding a network N_{aug} containing M isomorphic to the RHS of an augmented production (see section 3.1.7). The RHS of these augmented productions consist of the RHS of a production plus any presence condition modules. Once such a network is found, the corresponding production RHS (see section 3.1.7) is replaced by the LHS of the production if the reducibility condition, any possible absence conditions and all intersection conditions on nets are satisfied.

Events that correspond to modules which have already been reduced by previous events are removed from the queue and ignored. For example, suppose that A and B are two modules corresponding to two events in the queue with the event for A being scheduled for servicing first. Let us presume that during servicing the event for A, a reduction is performed which causes module B to be removed (abstracted) from the circuit. When the event for B gets serviced, B no longer exists and the event is therefore discarded.

In order to determine if the *event module* M of type T belongs to an augmented production RHS N_{aug} the following operation is performed. For each position in which a module of type T appears in an augmented production RHS N_{aug}, the parser checks if the circuit module M appears in a network isomorphic to N_{aug} at the same position. The parser maintains a list of positions in which modules of type T appear in an augmented production RHS and sequentially tries each position in the list for a potential match. The concise notation *M appears in position P* will be used to denote that module M appears in the circuit within a network isomorphic to some augmented production RHS in position P.

When a match in the list of positions is found and all other conditions are satisfied, the corresponding reduction is applied and an event associated with the new composite module created is placed on the event queue. If the event module M is a condition

module, it remains in the circuit after the reduction. In this case M may also be used in another reduction (typically also as a condition module). The position list must therefore be retraversed from the beginning in order to find another potential match. If M is not a condition module it *disappears* from the circuit after reduction and the parser begins processing the next event.

Example

The C_{blk} module type described in section 4.1 appears in the RHS of the productions of figures 4-2 (a2), (a3), (b2) and (b3) (which require no presence condition modules). Modules of type C_{blk} can also appear as presence condition modules for the production of figure 4-2 (a1) as described in section 4.1. The corresponding augmented production RHSs are shown in figures 5-1 (a), (b), (c), (d), (e) and (f). Two modules of type C_{blk} appear in the augmented production networks of (a) and (b) and only one in (c), (d) (e) and (f). The list of positions for module type C_{blk} contains the eight positions for modules of type C_{blk} in figures 5-1 (a), (b), (c), (d), (e) and (f). For each of the positions in the list the parser checks if the event module M appears in a network isomorphic to the corresponding augmented production RHS at that position. If a match is found for the positions of figures 5-1 (e) or (f), then M is a condition module and the positions list is traversed again for a new potential match.

5.1.3 Determining Production Applicability

Determining if a module M is in position P is performed using the search algorithm described in this section. This algorithm implements an efficient depth-first search of the circuit surrounding the event module. The algorithm is divided into two parts. In the first part an augmented production RHS and a position in that network are transformed into a sequence of instructions. These instructions are a sequence of elementary operations that need to be performed for determining if M is in position P. This operation is performed only once when GRASP starts up and the productions are read into GRASP. Given a module M in the circuit and a sequence of instructions for a position P (generated by part 1), the second part of the algorithm interprets these instructions to verify that M appears in position P in the circuit. By dividing the algorithm into two parts, coding and debugging the code is greatly facilitated. Execution speed is also improved because much of the work is done just once when the production is *compiled* into a set of instructions. The rest of this subsection describes the details of this search algorithm.

For each augmented production RHS N_{aug}, the parser assigns consecutive numbers

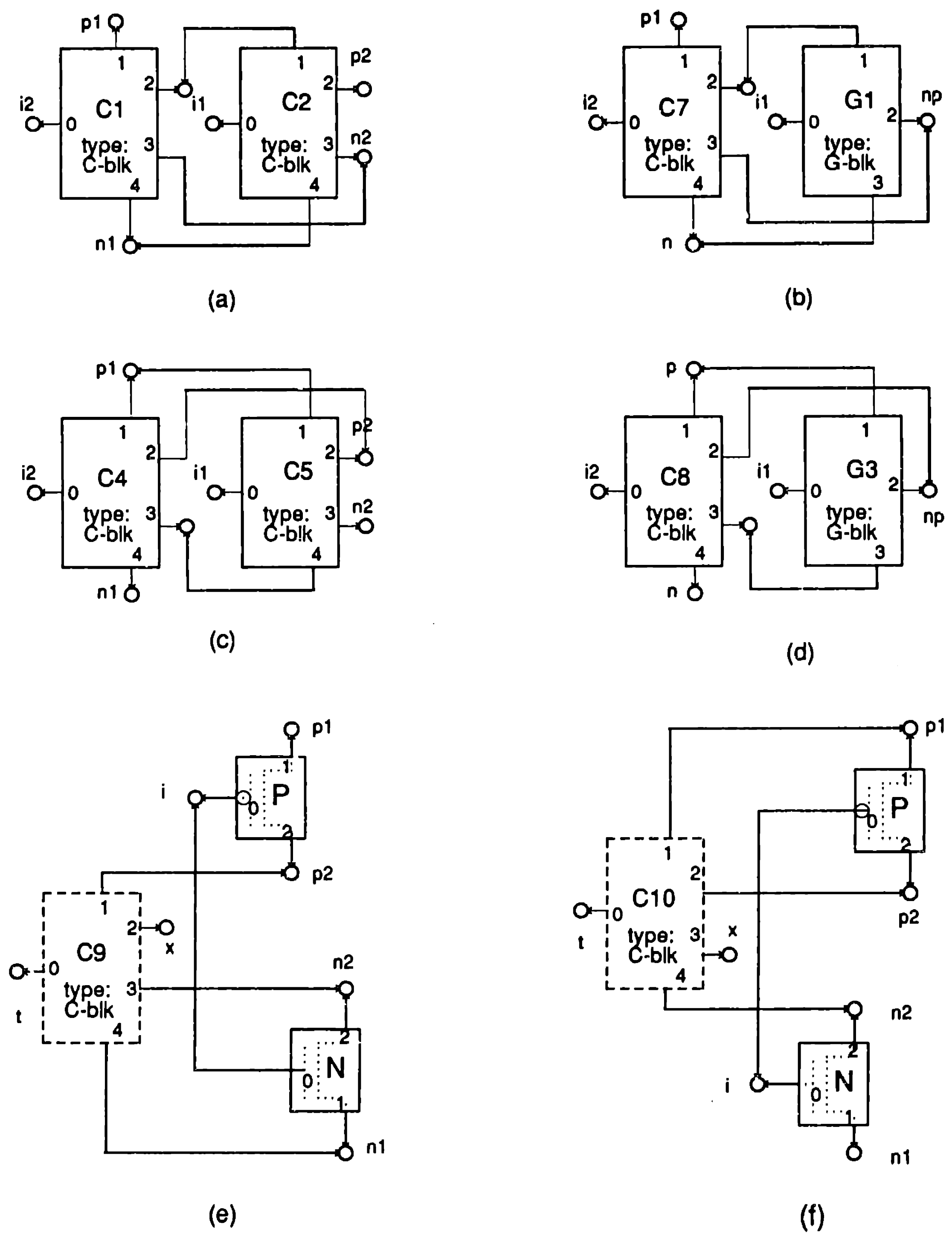


FIGURE 5-1: Positions of Modules C_{blk}

starting from 0 to the modules and nets in N_{aug} (as in figure 5-2 (a)). The parser maintains a set of module and net slots and it's goal is to fill these slots with modules and nets from the circuit so that the network formed by these modules and nets is isomorphic to N_{aug}. The slots are filled in such a way that the module in slot i corresponds to module i in N_{aug} and the net in slot j corresponds to net j in N_{aug}.

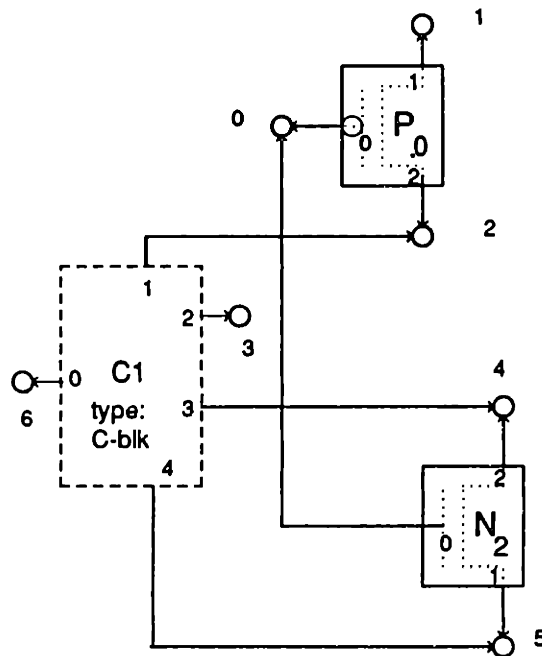
The instructions for position P describe how to find the modules in N_{aug} starting from the event module M and put them in their corresponding slots. Each instruction describes which type of module to search for next, the net to perform the search on and which slot to put the newly found module in. The form of an instruction is:
$I = (I.net_slot, I.net_relation, I.pin_number, I.module_type, I.module_slot)$. The interpretation of such an instruction is as follows: scan the pins of all nets related to the net in slot *I.net_slot* by relation *I.net_relation* and find a pin whose pin number is *I.pin_number* and belongs to a module of type *I.module_type*. Having found such a pin, insert its module in slot *I.module_slot*. The net relation field I.net_relation is one of: *identity*, *superior*, *inferior* or *adjacent* as described in section 3.1.16. The instructions are generated by a simple (depth-first) traversal of the modules in N_{aug}.

Figure 5-2 (a) is the same circuit as figure 4-4 (a) except that the modules and nets have been relabeled with consecutive numbers. Figure 5-2 (b) shows the set of instructions for the position of the module of type C_{blk} in the augmented production RHS of figure 5-2 (a). A description of the meaning of each instruction is provided alongside the instructions in figure 5-2. Figure 5-3 shows the state of the module and net slots for the production of figure 5-2 (a) after a network isomorphic to the production's RHS (the network formed by the modules X, Y and Z) has been found.

Figure 5-4 (a) is the same circuit as figure 4-12 with the modules and nets relabeled with consecutive numbers. Figure 5-4 (b) shows the set of instructions for the position of module 0 in the augmented production RHS of figure 5-4 (a). These instructions require that superior and inferior nets be searched.

Figure 5-5 describes in *pidgin algol* the recursive procedure for executing instructions. This procedure calls another procedure `insert_module_in_slot` which inserts a module into a given slot number. Before a module M can be inserted into a slot however, various checks and operations must be performed. If any of the checks fail, the value "*failure*" is returned and all the slots revert back to the state when `insert_module_in_slot` was called. The operations and checks that need to be performed by insert_module_in_slot are:

1. Verify that M is not already inserted in some other module slot. This is necessary in productions that combine modules of the same type in parallel in order to make



(a)

Instructions for module 1

1	(Begin,1)	*Put the event module in slot 1*
2	(2, Identity, 2, **P**, 0)	*Search net in slot 2 for a connected pin numbered 2 of a module of type* **P** *and put the module in module slot 0*
3	(4, Identity, 2, N, 2)	*Search net in slot 4 for a pin numbered 2 of a module of type* N *and put the module in module slot 2*
4	(Done)	*All modules have been found and put in their proper slots. Fire reduction after checking if all other conditions are met*

(b)

FIGURE 5-2: Network and Corresponding Instruction

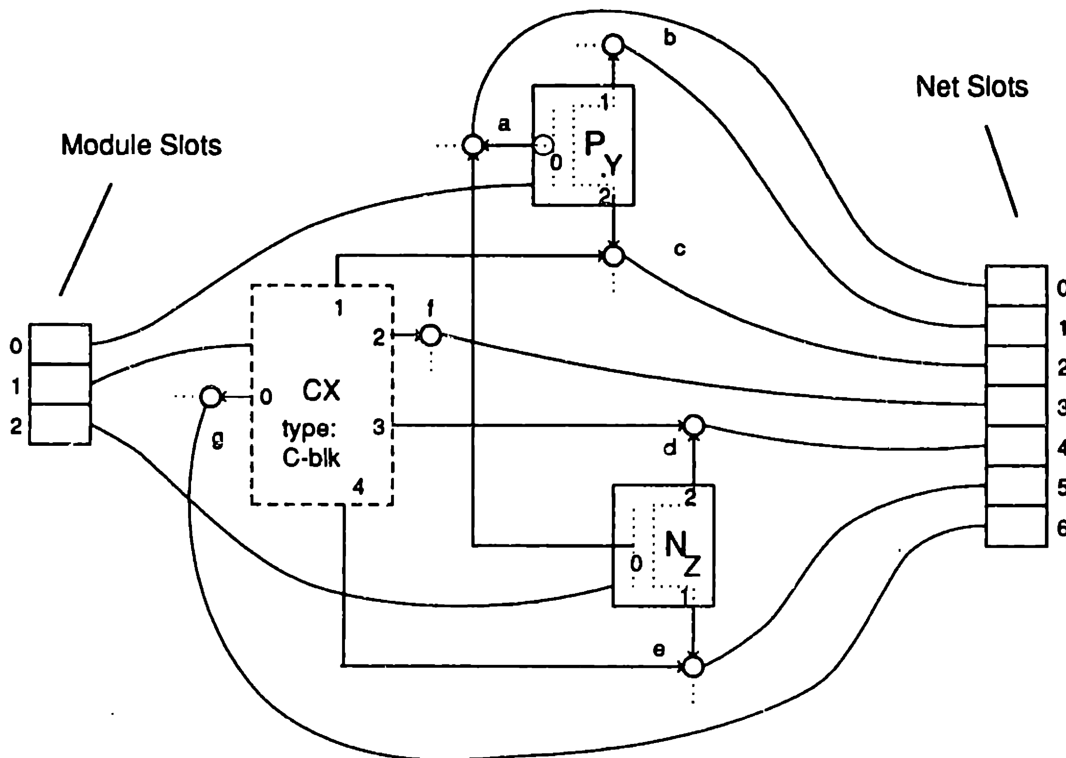


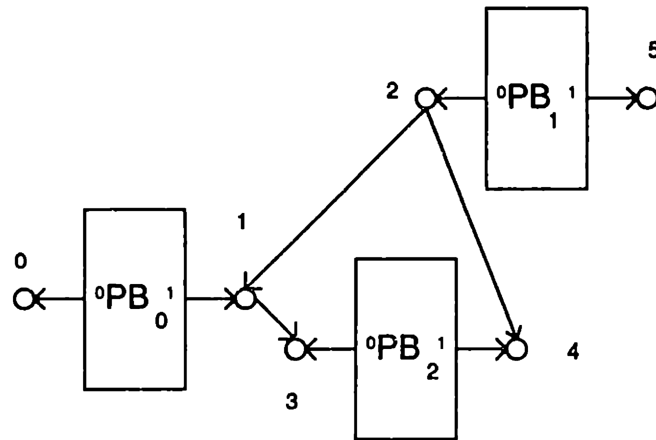
FIGURE 5-3: Module and Net Slots

sure that the same module (which can be considered in parallel with itself) does not appear twice.

2. Insert the nets connected to all the pins of M in their appropriate slots. These slots can be determined by examining the connections of the ith module (the module corresponding to M) in N_{aug}. If pin x of the ith module in N_{aug} connects to net j, then the net connected to pin x of M must go into net slot j. Before inserting a net η into a net slot j the following verifications must be performed.

   - Verify that η is not already inserted in another slot. Were this verification not performed, the algorithm would not be able to detect *short circuits* and would end up firing almost any reduction on the *bogus* circuit consisting of a large number of modules all of whose pins are connected to just one net.

   - If slot j is already full it must contain η. Were this verification not performed the algorithm would not be able to detect *open circuits* and would end up firing almost any reduction on the *bogus* circuit consisting of a large number of modules all of whose pins are connected to different nets.



(a)

Instructions for module 0

1 (Begin,0) — *Put the event module in slot 0*

2 (1, (Inferior,3), 0, **PB**, 2) — *Search all the inferiors of net in slot 1 and find one which is. connected to a pin numbered 0 belonging to a* **PB** *module. Put the newfound net in slot 3, and module in slot 2*

3 (4, (Superior,2), 0, **PB**, 1) — *Search all the superiors of net in slot 4 and find one which is. connected to a pin numbered 0 belonging to a* **PB** *module. Put the newfound net in slot 2, and module in slot 1*

4 (Done) — *All modules have been found and put in their proper slots. Fire reduction after checking if all other conditions are met*

(b)

FIGURE 5-4: Network and Instructions for Superior and Inferior Nets

```
Procedure execute_instruction(instruction)
 if is_first_instruction(instruction)
    insert_module_in_slot(the_event_module,instruction.module_slot)
    execute_instruction(next_instruction(instruction))
 else if is_last_instruction(instruction)
   fire_reduction()
   return(success)
 else For each net n in
      (instruction.net_relation(net_slots[instruction.net_slot]))
         For each pin p in connections(n)
           If ((pinnumber(p) == instruction.pin_number) and
               (typeof(moduleof(p)) == instruction.module_type)
             If is_successful(insert_module_in_slot(moduleof(p),
                             Instruction.module_slot))
                If is_successful(execute_instruction
                               (next_instruction(instruction))
                   return(success)
      return(failure)
```

FIGURE 5-5: Procedure Execute Instruction

5.1.4 Parsing Complexity

Number of Events

The total number of events is equal to the number of modules n_{mod} initially in the network plus the number of new modules created during parsing. The number of modules created during parsing is equal to the number of reductions applied. Generally, the number of modules in the circuit is reduced by at least one every time a reduction is fired because most productions have at least two modules (which are not condition modules) on the RHS. In this case the number of reductions applied is less than n_{mod}. Hence the number of events is linear (less than $2n_{mod}$) in the number of modules in the circuit.

The above result holds even when there are productions with only one module on the RHS provided certain conditions on these productions are met. Let $P_0 \cdots P_j \cdots P_n$ be a sequence of productions with one module on the RHS such that the module type of the LHS of P_i is the same as that of the module on the RHS of P_{i+1}. If there is no sequence such that the module type on the LHS of P_n is the same as that of the RHS of P_0 then the maximum length of any such $P_0 \cdots P_j \cdots P_n$ sequence is bounded. It is quite safe to assume that such a sequence does not exist otherwise it would be possible to fire

a sequence of reductions corresponding to $P_0 \cdots P_j \cdots P_n$ and end up with a module of the same module type as the RHS of the first reduction (albeit perhaps connected to the circuit differently).

Let n_{max} be the maximum length of any $P_0 \cdots P_j \cdots P_n$ sequence. For explanatory purposes the productions are divided into two categories: "*regular reductions*" which have two or more modules on the RHS and "*transformations*" which have only one module on the RHS. Firing a sequence of reductions $R_0 \cdots R_i \cdots R_n$ (corresponding to a sequence $P_0 \cdots P_j \cdots P_n$ as described above) where the module resulting from applying R_i is the RHS of R_{i+1} can be thought of as applying a series of "*transformations*" to the RHS of R_0. Using the same arguments as in the first paragraph it can be shown that during the parse of a circuit with n_{mod} modules not more than n_{mod} "*regular reductions*" are fired. Therefore not more than $2n_{mod}$ modules existed initially in the circuit or were created by a "*regular reduction*". To each of these modules not more than n_{max} transformations can be applied. Therefore a total of $n_{max}(2n_{mod})$ "*transformations*" could have been executed. Each applied "*transformation*" results in the creation of a new event. Therefore the total number of events is less than $n_{max}(2n_{mod}) + 2n_{mod} = 2n_{mod}(n_{max} + 1)$ and the total number of reductions is less than $n_{max}(2n_{mod}) + n_{mod} = n_{mod}(2n_{max} + 1)$. Therefore the number of events is linear in the number of modules initially in the circuit.

In practice $n_{max} \approx 1$ and few "*transformation*" type reductions get applied. Also many of the productions have more than two modules on the RHS. Finally about half of the events correspond to modules that no longer exist because they have been reduced by previous events (as described in section 5.1.2). Therefore in practice the number of *effective* events (those for which the event module is still in the circuit at the time the event is processed) is approximately n_{mod}.

Servicing an Event

Servicing an event requires scanning through the list of positions for that module type and possibly firing a reduction. This process may have to be repeated several times in the case of condition modules. Each time this procedure is repeated a reduction is necessarily fired as described in section 5.1.2. Since the total number of repetitions is less than the total number of reductions applied, the former is less than $n_{mod}(2n_{max}+1)$. The total number of scans, which is the total number of events plus the number of repeat scans of a position list (possibly followed by a reduction), is therefore:

$2n_{mod}(n_{max} + 1) + n_{mod}(2n_{max} + 1) = n_{mod}(4n_{max} + 3)$

Let L_{max} be the maximum length of a position list i.e. the maximum number of times any given module type appears in an augmented production RHS. Let α_{max} be

the maximum amount of time required to process an element in the list and β_{max} the maximum amount of time required to fire a production. In the worst case all the lists have to be traversed completely. The total parse time is comprised of the time taken to traverse the lists plus the time required to fire the reductions and hence is:

$$n_{mod}(4n_{max}+3)L_{max}\alpha_{max}+n_{mod}(2n_{max}+1)\beta_{max}$$
$$= n_{mod}((4n_{max}+3)L_{max}\alpha_{max}+(2n_{max}+1)\beta_{max})$$

Since typically the number of *effective* events and the number of reductions fired is n_{mod} and since list retraversal is a comparatively rare event, the parsing complexity is typically less than $n_{mod}(L_{max}\alpha_{max}+\beta_{max})$.

Processing an Event in the Positions List

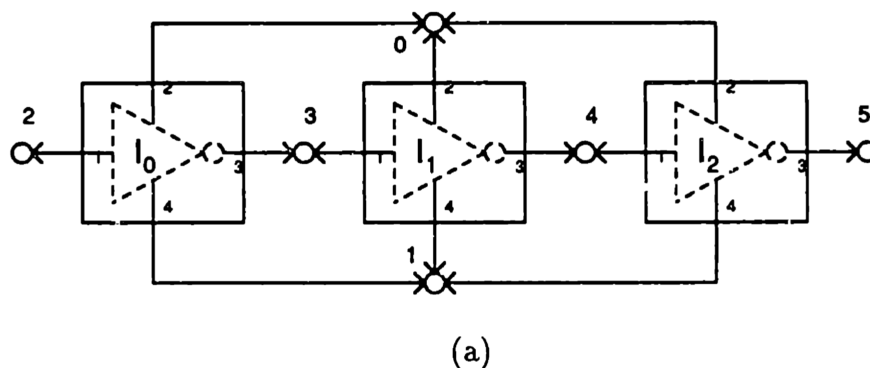
The time taken to process an element in the positions list is the time taken for the first call to `execute_instruction` to return. α_{max} is defined as an upper bound on this execution time.

During a call to `execute_instruction` let us suppose that a maximum of m pins are examined, that is to say the variable p in procedure `execute_instruction` gets bound to a new value a maximum of m times. In the worst case each assignment of p will result in a new call to `execute_instruction`. If the number of modules in the current augmented production RHS is k then in the worst case only every k^{th} recursive call to `execute_instruction` will result in failure. At that time a maximum of m^k (actually $\sum_{i=0}^{k} m^i$) pins will have been examined. If the maximum number of modules in any augmented production RHS is R_{max} then not more than $m^{R_{max}}$ pins are examined.

The number of pins p examined during a call to `execute_instruction` depends not only on the circuit and the production RHS but also on the instructions generated for that production in part 1 of the algorithm. Figures 5-6 (a) and (b) show two different possible instruction sets for the position of the leftmost inverter in figure 5-6 (a). In the instructions of figure 5-6 (b) nets x and y are scanned in search of the next inverter module. In figure 5-6 (c) the `vdd` net is scanned instead. In the worst case all modules[2] connected to vdd must be tried and rejected by procedure `insert_module` which checks to see that the other connections of the newly found inverter are correct.

In a circuit consisting of a huge chain of n_{mod} inverters the maximum number of pins examined during a call to `execute_instruction` for the instructions of figure 5-6 (a) is $m_1 = 2$. On the other hand for figure 5-6 (b) this number is $m_2 = n_{mod}$ because all the n_{mod} modules can be connected to `vdd` (net 0). In the case of figure 5-6 (a), the number

[2]In the worst case all the modules in the circuit are inverters.



(a)

Instructions for module 0

	(b)		(c)
1	(Begin,0)	1	(Begin,0)
2	(3, Identity, 1, I, 1)	2	(0, Identity, 2, I, 1)
3	(4, Identity, 1, I, 1)	3	(0, Identity, 2, I, 1)
4	(Done)	4	(Done)

(b) (c)

FIGURE 5-6: Efficient and Inefficient Instructions

of pins examined is approximately $m_1^2 \approx 1$. For figure 5-6 (b) this number is $m_2^2 = n_{mod}^2$. In the worst case $m = n_{mod}^{R_{max}}$. Instructions should therefore be generated in such a way that nets which typically have the least number of connected pins (usually *internal nets*) are scanned.

Instructions such as those in figure 5-4 (b) require that every net in relation (via the three basic relations) to the net in slot i be scanned. The criteria for selecting an appropriate net is to choose the one which has the fewest number of nets in relation with it each of which has a small number of pins connected to it.

By carefully choosing which nets to scan, the average time required to process an element in the positions list (which is proportional to m the number of pins examined) is largely independent of circuit size.

Some augmented production RHSs contain modules such that in order to access them it is necessary to scan nets with a large number of connections. For example, if the power supply module appears as a condition module of an augmented production's RHS, then it might be necessary to scan the vdd (or gnd) net to access the power supply module. By having each net capable of quickly accessing pins only connected to certain module types without having to scan all the pins connected to it, the time required to scan the net is considerably reduced.

Operation	Worst Case	Typical
Number of events	$O(n_{mod})$	$O(n_{mod})$
Number of reductions	$O(n_{mod})$	$O(n_{mod})$
Number of list traversals	$O(n_{mod})$	$O(n_{mod})$
Per element in List	$O(n_{mod}^{R_{max}})$	$O(2^{R_{max}})$
Per reduction	-	$O(1)$
Total	$\approx O(n_{mod}^{R_{max}+1})$	$\approx O(n_{mod})$

Table 5-1: Parsing Complexity Summary

Firing a Reduction

Firing a reduction involves deleting some modules from the circuit, creating a new module and inserting it into the circuit. This operation is independent of circuit size. During this operation new net bundles also may have to be created. The time complexity of creating these new net bundles depends on the implementation of the net bundle data structures. In practice, for circuits up to several hundred transistors (even for a very crude implementation of net bundles) the time required to create a new bundle is largely independent of circuit size.

Interaction with Virtual Memory

GRASPs event driven algorithm is *well behaved* with respect to virtual memory management schemes. Memory references made during the servicing of an event occur in a *connected neighborhood* of the event module. If care has been taken to generate efficient parsing instructions (see section 5.1.4), the number of modules (respectively nets) accessed during an event is typically 10 (respectively 15) and rarely exceeds 100 (respectively 150). Page faults during the servicing of an event are rare because the amount of memory accessed is usually a few dozen kilobytes which is small enough to fit in RAM memory or even in large cache memories.

Parsing Complexity Summary

Table 5-1 summarizes the basic asymptotic results of this section. The right set of instructions can have a profound impact on the time complexity of the parsing algorithm and the implementation of this algorithm described in section 5.2 allows the user to specify which nets to scan making the typical parsing complexity linear in the number of modules in the circuit.

5.1.5 Rescheduling due to Absence Conditions

In section 3.1.7 a subset of absence conditions with well-behaved properties was introduced. These absence conditions for a production P list the only module types and pin numbers (other than those in the network N_E being reduced) allowed to connect to certain external nets of the network N_E. The presence of modules (or pin numbers) not in this list will inhibit the reduction from being fired. It is possible for these modules to be reduced later into modules in the allowed list of this production. It could be the case that at that time no events for any of the modules of N_E remain on the event queue and hence the reduction does not have a chance of firing even though all its requirements are satisfied.

For each module M_i in the allowed list, a new production identical to P but with M_i as a condition module needs to be created. In this case when modules not in the allowed list are reduced to modules M_i in the allowed list, during the processing of the event for an M_i the reduction for N_E (augmented by M_i as a condition module) will again be attempted. This ensures that only when modules in the allowed list remain will the reduction be applied.

5.1.6 Incremental Update

GRASP's algorithm allows for incremental additions and deletions to the circuit. The *parse tree* data structure built during parsing maintains a log of which networks were reduced, which new composite modules were created in the process and of the grammar production involved. For each reduction the condition modules (if any) involved are also recorded[3]. The parse tree plus the condition module information is called the ***augmented parse tree***. The augmented parse tree is used when, due to modifications to the circuit by the user, some of the reductions must be undone.

Undoing a Reduction

Undoing a reduction R consists of replacing the resulting reduced module M for that reduction with the modules M_i it was derived from. These modules must be put back on the event queue so that they can be reduced again (presumably the rest of the circuit has also changed so that the same reduction is not applied again when these events are processed). Processing of events can then be enabled.

[3]This causes the *parse tree* to actually be an acyclic graph.

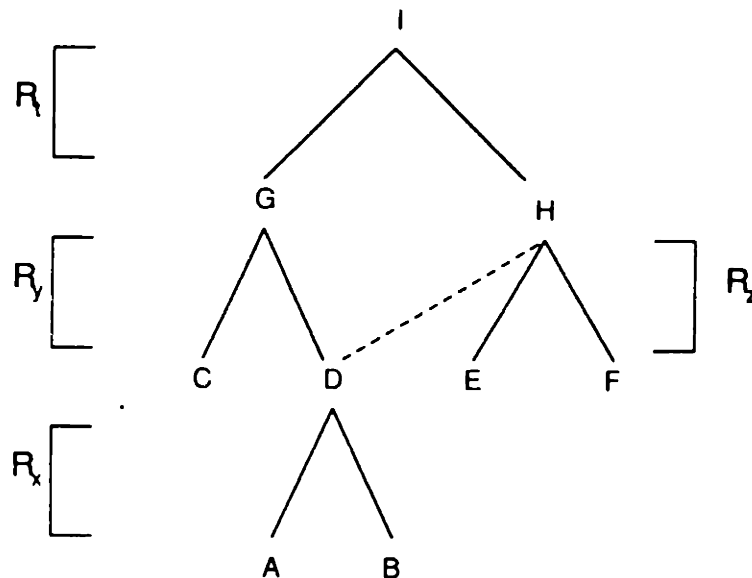


FIGURE 5-7: Augmented Parse Tree

Before reduction R can be undone, all reductions R_i which use module M on the RHS, even as a condition module, must first be undone. This in turn requires that for each R_i, all reductions $R_{i,j}$ that use the module resulting from the reduction R_i be undone first. Thus in order to undo a reduction R with resulting reduced module M all reductions on all paths between M and the root of the augmented parse tree must first be undone in top down fashion. If the circuit has not been completely parsed then there may be several *roots* in the augmented parse tree[4]. For each root that can be reached from M all reductions on all paths from M to that root must be undone. If the augmented parse tree (parse forest) is balanced, the number of reductions that must be undone is $O(\log n_{mod})$ where n_{mod} is the number of modules in the initial network .

Figure 5-7 shows an augmented parse tree. The dependence of a reduction on condition modules is shown by a dotted line. If reduction R_x is to be undone then reductions R_t, R_z, R_y and R_x must be undone in that order. Events for modules H, C, A and B (actually events for G, H, C, D, A and B but modules G and D get deleted) must be created and placed on the event queue.

Deleting a Module

Before deleting a module M from the (initial) circuit, any reductions in which M is used must first be undone (causing all other necessary reductions to be undone). Only after the reduction has been undone can the module be deleted. Enabling the processing

[4]At this time the augmented parse tree is actually a parse forest.

of events will then rebuild the parse tree efficiently.

Sectioning the Circuit

The initial circuit can be divided into two disjoint parts in such a way that much of the parsing information for each part can be retained. This can be achieved by undoing those reductions whose RHSs consist of modules of both parts. The two parts of the circuit can then be *cut* apart resulting in two separate augmented parse trees (parse forests). This procedure is more efficient than having to parse each part from scratch.

Adding a Module

Before adding a module M from the (initial) circuit, two kinds of reductions must be undone:

1. Reductions for which any of the nets connected to M are internal nets. This is because for those reductions the reducibility condition is no longer satisfied.

2. Reductions which have an absence condition that will be violated when module M is inserted. Absence conditions are of the form: if this reduction is to be fired then net x cannot be connected to pin y of some module type Z. Each net records the negative conditions associated to it as well as the pertaining reduction. For each net to be connected to M those reductions which become violated must be undone.

After the appropriate reductions have been undone the module M can be added to the circuit. An event associated with M must also be created and placed on the event queue.

Merging Two Circuits

Two fully or partially parsed circuits can be merged together in such a way that the parsing information for each part is combined into one augmented parse tree. This operation is much more efficient than rebuilding the parse tree for the combined circuit from scratch.

First, those reductions in both parts which become illegal because of the new connections between the two parts (see previous subsection) must be undone. The events generated by the undoing of the reductions as well as the events from the event queues of both parts that have not yet been processed must be placed on one common event queue. The two circuits can then be connected. The effects of these connections must be

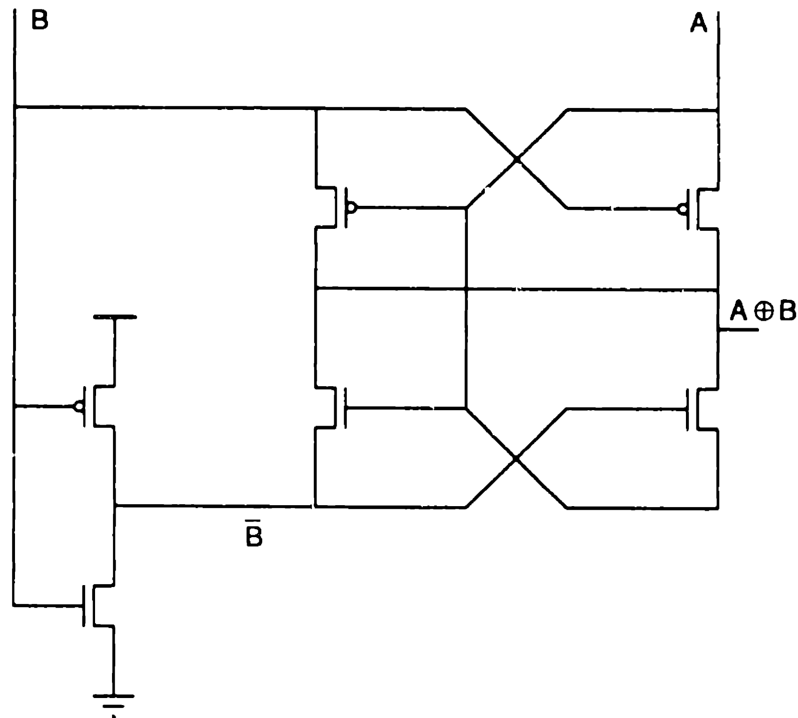


FIGURE 5-8: Six Transistor XOR Gate

propagated back up both parse trees. That is to say the connections to all the net bundles and the relations between the net bundles must be updated to reflect the connections between the two circuits.

Those modules in each of the reduced circuits (after some of the reductions have been undone) which are connected to net bundles, whose connections have changed or whose relations have changed since the merger, must be placed on the event queue. The reason for this is that these modules can now be used in a reduction with modules from the other part. Enabling the processing of events will then build the joint parse tree appropriately.

Library Modules

The merging techniques described in the previous subsection are especially useful when dealing with library modules. Library modules such as *full adder* or *xor* modules which consist of many transistors can be *preparsed* before being used. *Preparsing* library modules consists of parsing the circuit for that module. When these library modules are used in a circuit the parsed version of the module and of the circuit are merged together as described previously in this section. Because parsing effort is not expended to reduce the circuit of each library module instance into its parsed version, a considerable savings in parse time can be achieved especially for large library modules.

Preparsing library modules can be used as a way to incorporate subcircuits that use

special non standard design techniques into the circuit and still have the parser recognize the circuit as error free. For example the XOR gate circuit of figure 5-8[5] can be *manually parsed* into a complementary CMOS gate. The significance of this is that for the purposes of design style verification the circuitry of the XOR gate is equivalent to that of a classical CMOS gate. When this library module is instantiated in a circuit the parser uses the preparsed version and hence is *fooled* into thinking that circuits which use this XOR gate obey the design style.

Mixing Grammars

Merging techniques can also be used to parse different portions of a schematic with different grammars. This might become necessary when different portions of the schematic are designed using different incompatible design methodologies. The schematic S is divided into sub-schematics S_i and each S_i is parsed using a grammar G_i. The partially parsed schematics S_i' derived from the S_i are then merged together into a single schematic S'. A partial parse tree T' for the combined circuit is also built from the parse trees of the S_i'. The parse of S' can then be completed using a grammar G' (capable of combining the composite modules in each S_i') and the partial parse tree T'.

5.1.7 Error Reporting

When the circuit to be verified does not obey the design style it is not enough for the verification system to merely report that the circuit is incorrect. Feedback must be given to the user as to the type and location of the error so that he may fix it. GRASP provides valuable feedback to the user which can help him locate the source of the error.

When the circuit to be verified is not in the range space of the grammar, GRASP will fail to produce a sequence of reductions resulting in the start symbol of the grammar. Typically after having applied several reductions to the circuit there will come a point where no more reductions are possible. This manifests itself by the event queue becoming empty while there remains more than one module in the circuit.

At this point the resulting circuit consisting of the new composite modules must be examined. Locating errors is greatly facilitated by the fact that most of the circuit is now expressed in terms of large composite modules (such as combinatorial blocks, precharged blocks etc..) whose internals are known to be well-formed. Errors are usually localized

[5]For clarity's sake this figure is shown using the traditional pictorial representation of figure 3-1 (a) instead of the representation of figure 3-1 (b) used in this thesis in which the nets appear explicitly as circles.

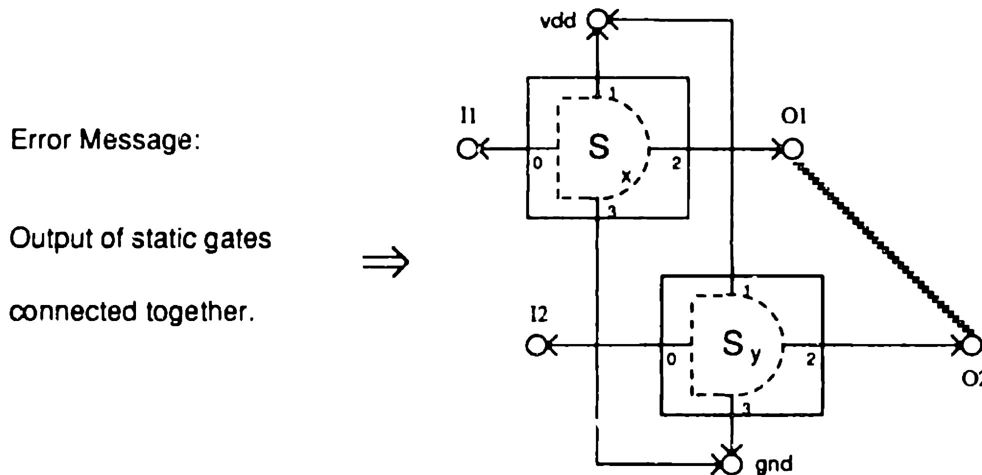


FIGURE 5-9: Example of an Error Production

in regions where few reductions have been performed and the parse tree can be used to quickly locate these regions.

GRASP's structure finding capability can also be used to help locate errors by using special *error productions* which look for illegal configurations in terms of the composite modules in the reduced circuit. The RHS of the error productions consist of a network just as for regular productions however their LHS is an error report that needs to be sent to the user when a RHS match is found. Because the error productions are expressed in terms of composite modules and operate on the partially parsed circuit they make use of the existing parsing information and hence are a much more powerful error finding mechanism than simply looking for illegal configuration in the initial circuit.

Figure 5-9 shows an example of an error production which fires when the output of two static gates are connected together. An error report is sent back to the user along with the modules involved in the production. Catching this sort of error from the partially parsed version of the circuit is much easier than identifying this error directly from the initial transistor netlist.

5.2 Implementation

Source Code

GRASP is written in C and runs on a HP 9000 series model 350 workstation running UNIX and Xwindows. The source code is approximately 15000 lines out of which the core of the parsing algorithm takes approximately 3000 lines. The code makes extensive use of data abstractions and is written much like a program in C^{++}[44].

The core parsing algorithm accesses abstract data types such as modules, pins, and

net bundles through procedural interfaces and never manipulates the underlying data structures for these data types directly. By providing these procedural interfaces to the data types the core parser code can be incorporated *as is* into an existing CAD tool such as a schematic editor.

The parsing algorithm code is extremely efficient and as such the sophistication and efficiency of the code implementing the abstract data types has a large impact on run time. In one of the earlier versions of GRASP these data types allocated memory using a simple C library memory allocator. By simply using a more efficient memory allocator a speedup of 3× in parse time was achieved. Similar optimizations will most likely result in a further speedup especially for large circuits.

GRASP first reads in the grammar productions, then the circuit netlist and then proceeds to parse the circuit. The productions are described using a Lisp like syntax. A YACC[4] front end reads the text descriptions and transforms them into an internal representation for productions which consists of module, net and pin data types. After each production is read, the instruction generator is called to generate instructions for all positions in the production.

Input Files

Figure 5-10 shows a textual representation of the classical CMOS grammar production of figure 4-1 (a2).

The circuit to be parsed is read from a netlist file whose format is similar to [48]. Figure 5-11 shows an example of such a file consisting of a 4 input NAND gate and an inverter.

Graphics Display

An Xwindow interface built into GRASP can display on a graphics window either a grammatical production or a portion of the circuit surrounding the event module. The display shows the circuit just before and after a reduction is performed. The event module appears at the top of the screen. The modules participating in the reduction and the condition modules are distinguished by different colors. The network is displayed with the modules on the LHS and the nets on the RHS. Superior and inferior relations between the displayed nets are also shown.

Figure 5-12 shows what an Xwindow display might look like. A *grab screen* feature built into GRASP was used to generate this figure. The figure represents a snapshot of the netlist of figure 5-11 during parsing. In this snapshot the event module Module 19

```
(production C_from_2C_n_parallel (2 7 1)
          % Production has: 2 modules, 7 nets (0-6)
          % and creates 1 new net (net 7)
      (LHS_type C)
          % Module type of the LHS is ''C''
      (LHS_connections 7 1 2 3 4)
          % Connections of the LHS module
          % Pin 0 connects to net 7, pin 1 to net 1, etc....
       (moduletypes C C)
          % module types of the RHS. The first module is referred to as
          % module 0, the second one as module 1 etc..
      (rulemodules 0 1)
          % Both modules 0 and 1 get reduced when the production is fired.
          % Modules not in this list are presence condition modules.
       (module_conn (0 (6 1 2 5 4))  % Connections of module 0
                    (1 (0 1 2 3 5))) % Connections of module 1
       (eq_groups)
          % If for example net 0 is defined as superior to net 1 then
          % (eq_groups (0 1)) allows them to also be equal (same net)
       (superiors (6 7) (0 7))
          % Net bundle 7 is superior to 6 and to 0.
       (adjacents)         % No adjacent relations. Superiors and
          % inferiors are automatically declared adjacent.
       (non_intersecting)
          % No empty intersection requirements between the nets.
       (unwanted)          % Absence condition list.
       (hintlist 5 2 1)    % Optional precedence list for nets.  Allows
          % optimized instructions by having the user indicate which nets
          % to scan preferentially
       print_production    % print production  debugging information
       print_instructions  % print instruction debugging information
)
```

FIGURE 5-10: Textual Representation of a Production

```
Number_of_nets: 12
% Contents: Nand Gate + Inverter + Power Supply
p_trans 0 2 7
n_trans 0 3 7
n_trans 7 6 4
p_trans 10 2 4
n_trans 8 5 6
p_trans 11 2 4
p_trans 8 2 4
supply 2 3
p_trans 7 2 4
n_trans 10 9 5
n_trans 11 3 9
```

FIGURE 5-11: Circuit Input Netlist

Circuit	Transistors	Parse Time
4 input CMOS NAND Gate	8	0.05s
4×4 Systolic Multiplier	978	14s
8×8 Systolic Multiplier	3746	168s

Table 5-2: Parse Times

of type $C2$ is about to be replaced by a static gate module as per the production of figure 4-2 (c). The power supply module, module 7 is a condition module and is outlined in dashed line. The third module is not involved in the production and is outlined in dotted line. Net 7 is contained in net 9 and the relation between these two nets is shown by a thin arc directed from net 9 to net 7.

The display interface is a useful tool for debugging the code and demonstrating the operation of the parsing algorithm. It is not intended as a schematic editor and provides no features for incremental additions and deletions to the circuit.

5.3 Experiments

Grammars for static CMOS and NMOS gates, domino gates, NMOS precharged gates and the two phase non-overlapping clocking methodology have been designed.

To evaluate the parsing speed of GRASP the grammar for the CMOS two phase clocking methodology [53] introduced in section 4.2 has been tried out on a large bit-systolic multiplier circuit [21]. The grammar consists of about 50 small grammar productions (there are usually 2-3 modules in each augmented production RHS) and takes about 3

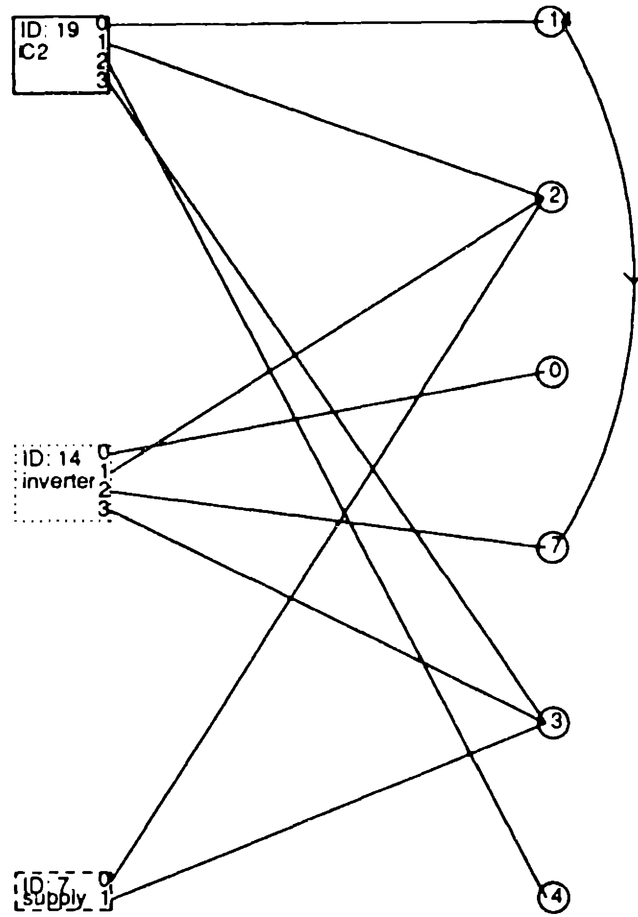


FIGURE 5-12: Xwindow Graphic Display

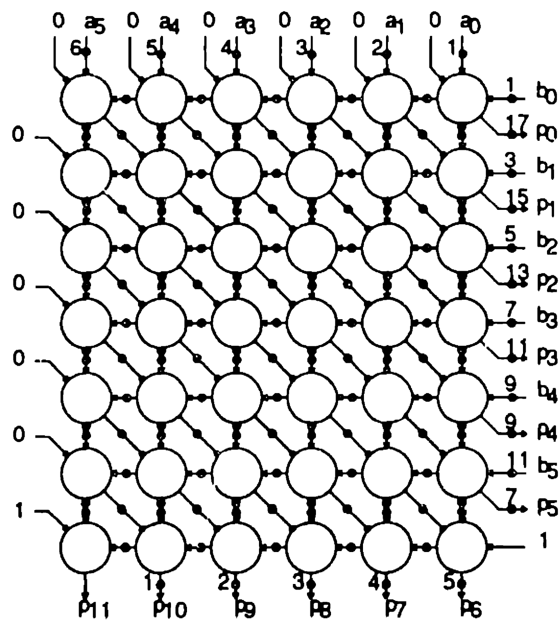


FIGURE 5-13: Systolic Multiplier

seconds[6] to load into the GRASP program.

Figure 5-13 shows the bit-systolic multiplier for the 6×6 case. Each circle represents a precharged full adder cell. The dots represent CMOS latches whose number and location are obtained by the retiming theorem [28]. Each column operates on an alternate clock phase. Table 5-2 summarizes the parsing time required for various circuit sizes. The computation times are shown for the case where each circuit is input as a flat netlist file of transistors and waveform generator modules (power, ground, and clocks). Parse time is roughly 35 modules/second which compares favorably to the 1 module/second VAX 8800 required by [40]. The computation times can be substantially reduced by preparsing the basic multiplier cell as described in section 5.1.6.

As the circuit gets large an increasing proportion of time is spent creating new bundles. This accounts for the slightly worse than linear parse times for large circuits. In the current implementation of GRASP, net bundles are implemented in a rather brute force fashion. For each net bundle a separate list of all its superiors, inferiors, adjacents as well as all the *individual nets* contained in it is maintained. When each new net bundle is created these lists for many other net bundles must also be updated. The rational for this is that most of the productions that are tried are not applied and hence creating a net bundle is a comparatively rare event. By contrast adjacent, superior and inferior net access is a common operation and hence needs to be optimized. When the net bundles become large however, the number of nets in relation with each other become large. In this case not only do the relations list for many net bundles have to be updated but each list is also large and hence can take time to update[7]. This accounts for much of the nonlinearity in the parse times of table 5-2.

[6] Includes the time required for generating the instructions.

[7] The lists are kept sorted so insertion and deletion time increases as the list size grows.

6

Layout Verification

6.1 Introduction

Layout design rule verification (abbreviated as DRC for design rule checking) is the process of verifying that a given set of mask patterns to be used during wafer fabrication belongs to a set of permissible geometries. The set of permissible geometries can be described by geometrical constraints that all patterns in this set must satisfy. Usually these constraints, called design rules, take the form of minimum and maximum allowable values between features of the various patterns. The task of verifying that a mask pattern obeys a set of design rules has received considerable attention [7], [6].

In a typical layout the number of geometrical features that need to be verified is large and hence the verification process requires large amounts of computation time. Various techniques for reducing processing time include special purpose hardware [37], exploiting parallelism [15] or incremental verification [47]. When the mask patterns are described using hierarchically defined cells, hierarchical DRC techniques [54] can be used to verify the mask patterns of subcells only once thus saving computation time.

In each of the techniques described above when a layout expressed entirely in terms of library cells needs to be design rule verified, the mask geometries of the layout are first generated and then DRC verification is performed on them. The number of geometrical values needed to describe the mask pattern is typically several orders of magnitude greater than the number of library cell instances used. Thus verification based on geometrical values requires manipulation of a vastly greater number of parameters than the number of cell instances. Also errors are reported back to the user in terms of violations of the minimum and maximum permissible values of geometric features. The burden of identifying which cell instance placement is at fault then rests with the user.

In this chapter a technique for verifying the DRC correctness of a layout built from library cells is presented. The method operates directly on the library cells and their placements and never accesses or manipulates mask patterns. It is applicable to a continuum of cell sizes ranging from cells consisting of a single polygon (though this can lead to a huge number of templates) to cells containing several hundred polygons. The method is not restricted to any particular layout style and allows a great deal of flexibility in the library cells and their placements.

Because mask geometries are not manipulated the verification algorithm has to deal with comparatively few objects, hence verification time and memory requirements are greatly reduced. Errors are reported to the user in terms of placement conflicts between the cell instances considerably facilitating the task of identifying the cause of the error and fixing it. Finally in chapter 8 a method for verifying the correspondence between a schematic and a layout is introduced. This method requires that the layout be first verified using the techniques of this chapter.

6.2 Overview

The set of layouts L_c consisting solely of cell instances from a set of library cell types is considered. A technique for defining a subset L_i of L_c consisting of *well-formed* layouts, similar in essence to the techniques of chapter 3 is presented. The method makes use of user defined *templates* consisting of small *fragments* of layouts to specify L_i. During verification, the system attempts to cover the layout with these templates in order to prove its membership in L_i. The layout and the templates are modeled as graphs [10] and all verification operations are performed on the graphs.

The formalisms in the layout verification method are geared toward verifying the *design rule* (DRC) correctness of the layout and are not concerned with the underlying electrical meaning of the layout. Electrical errors are caught by examining the schematic not the layout representation of a design instance.

In schematic design style verification errors are not necessarily localized in the sense that any finite region of the schematic might be error free and the circuit as a whole might still have a design methodology error. For example, in the two phase clocking methodology, signal loops through static gates are not allowed. Since the length of the loop is unbounded an arbitrarily large region of the circuit might have to be examined to find the loop. During parsing, portions of the circuit that might initially be *far* apart are progressively brought *closer* due to network reductions. At some point all the circuitry necessary to detect the error will appear in a locally connected neighborhood of the

reduced circuit at which time the error can be detected by some production.

Unlike schematic design style verification, layout verification can be accomplished without hierarchical parsing. Verifying the DRC correctness of a region of layout can be accomplished by examining its geometric neighborhood. Regions of layout that are geometrically far apart do not interact with each other and therefore in order to verify the correctness of the whole layout it suffices to verify the correctness of each of the regions independently.

Section 6.3 describes the basis for our layout correctness criteria. Section 6.4 describes the graph representation for the layout. Section 6.5 describes how the graph representation is used to verify the layout correctness.

6.3 Layout Correctness

6.3.1 Criteria for Layout Correctness

Let the absolute bounding box of an instance be defined as the smallest rectangle enclosing all the mask polygons associated with the instance. Let the bounding box of an instance then be defined as the rectangle obtained by expanding its absolute bounding box by half the maximum design rule size. A design rule error can occur only between two mask polygons (of which one may be itself) which are separated by less than the maximum design rule size. Therefore a design rule error can occur at most one half the distance from the nearest polygons. Hence any design rule error involving polygons of instance C_i necessarily appears in the bounding box of C_i.

A layout in L_c is DRC correct if all the regions contained within the bounding box of each of the instances C_i in L_c are DRC correct[1]. The layout within the bounding box of instance C_i consists of layout from the instance C_i itself and layout from the neighboring instances[2]. Each instance C_i and its neighbors form a pattern of cells called template. The bounding box of C_i forms a DRC error free zone for that template and is called the *interior* of the template. If more than one instance in the template has a DRC error free zone then there may be more than one *interior* instance per template.

Example

Consider the layout of the PLA shown in figure 6-1. This PLA is built from instances of library cells as shown in figure 6-2. From the layout of this PLA several useful templates

[1] There are no DRC errors in the bounding box regions

[2] Those instances whose bounding box intersects with the bounding box of C_i.

can be identified. Figures 6-3 (a) and (b) show two possible templates. The interior of each template is shown by the shaded region. The interiors of these two templates also appear as the shaded regions in figures 6-1 and 6-2.

6.3.2 Layout Verification using Templates

If a pattern of cells *equivalent* to a template *appears* in any layout and the instance D_i corresponding to the interior of the template interacts only with those instances corresponding to the template, then the bounding box of D_i, the template, is necessarily a DRC error free zone. Instance D_i is said to be the interior of this occurrence of the template.

The templates that can be derived from figure 6-2 (plus a few others that do not appear in this layout) can be used to verify layouts of PLAs similar to the one in figure 6-2 but of any size and encoding. The PLA example of figure 6-2 is chosen because it is easy to visualize how the various cells in a PLA fit together. The usefulness of this scheme is certainly limited for PLAs which are usually procedurally generated. However templates can be used to describe how cells from more general cell libraries are put together and can hence be used to verify the more general class of layouts assembled from cells in the library. These templates allow the user to define a subclass of layouts L_i of L_c deemed acceptable.

In order to verify that a layout built from library cells is correct according to a user defined set of templates $\mathcal{T}$, each instance in the layout must be the interior of at least one template occurrence. By definition of a template the bounding box regions for each of the instances will then be a DRC error free zone and the layout as a whole will therefore be DRC error free. The layout is said to be in the *range space* of the templates $\mathcal{T}$.

6.4 Review of RSG Connectivity Graphs

Connectivity graphs first introduced in [8] provide a simple yet powerful representation for layouts. The connectivity graph is especially useful when dealing with the placement of the cells in the layout. It is a better representation than the layout itself for assembling cells into larger cells and verifying that they have been assembled *correctly*. In the layout assembler RSG [8], all cell assembly operations are carried out on the graph instead of the layout. When the graph is finally complete a fast and simple graph to layout transformation is applied to produce the final layout. Section 6.4.1 reviews the basics of RSG connectivity graphs.

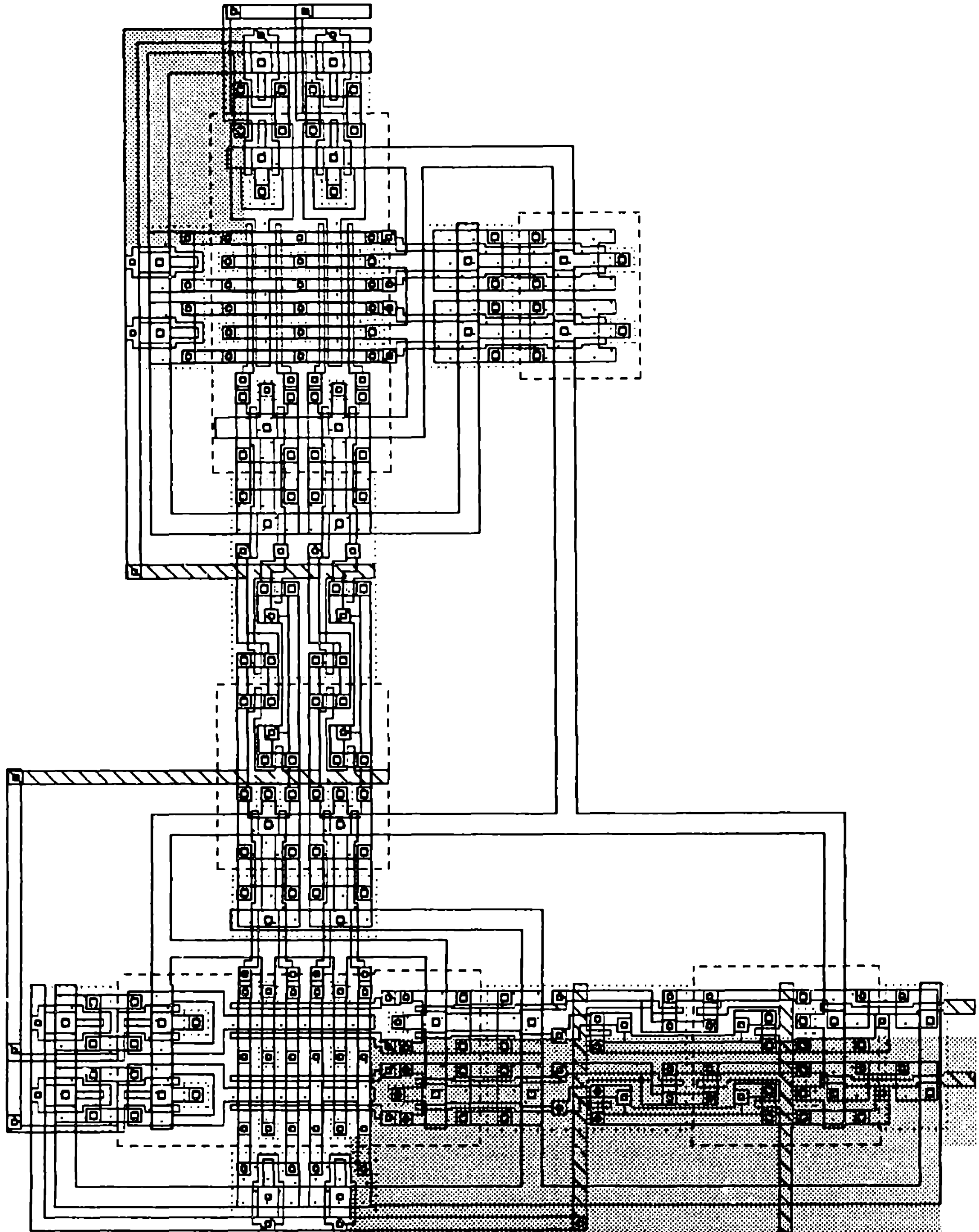


FIGURE 6-1: PLA Layout

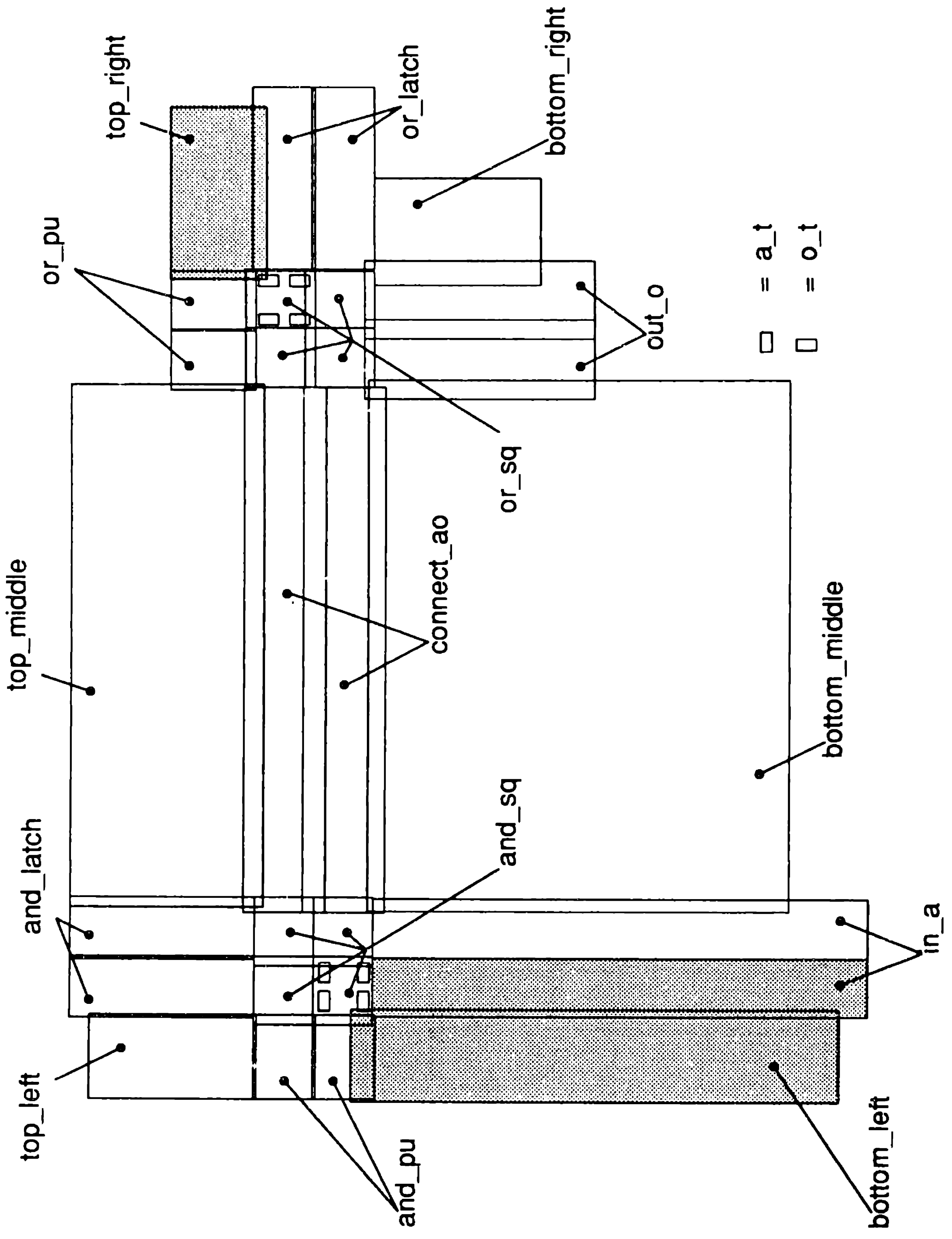


FIGURE 6-2: PLA Instances

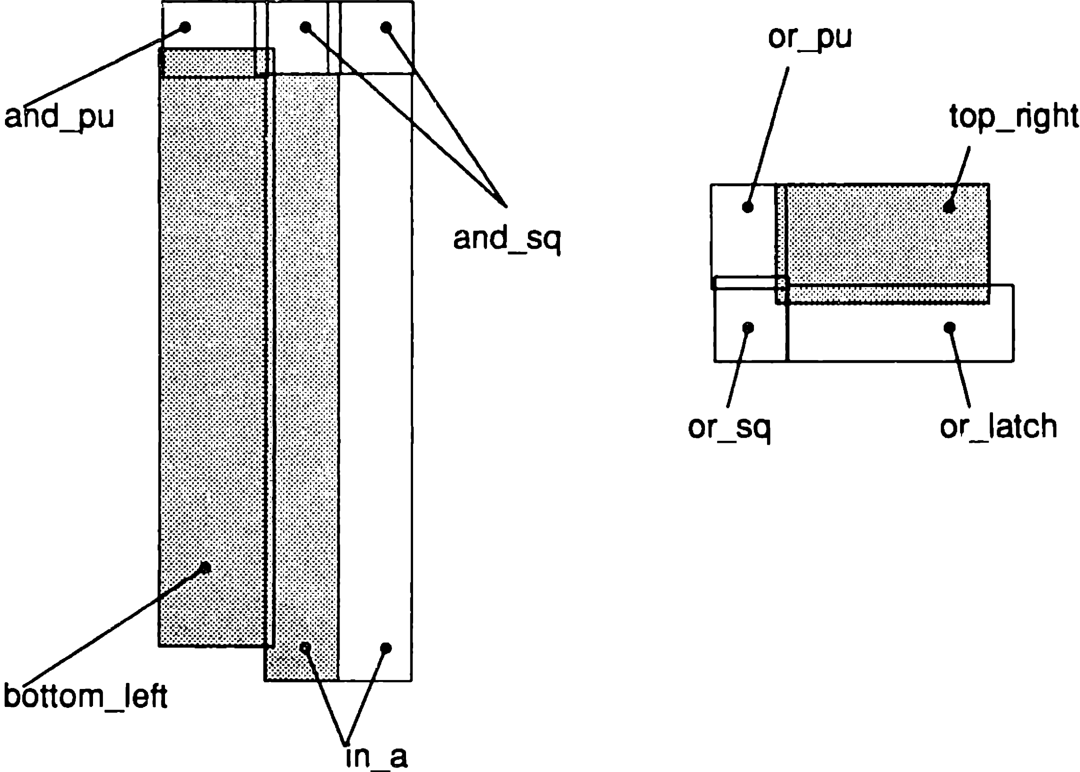


FIGURE 6-3: Examples of Templates

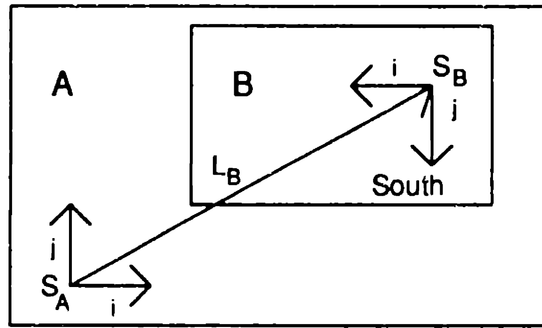


FIGURE 6-4: Instance of Cell B in Cell A.

6.4.1 Cells, Interfaces and Connectivity Graphs

Cells

A cell A consists of a collection of objects whose locations are specified in terms of the coordinate system of the cell C_A with origin S_A. The objects can be polygons, points, wires or instances of other cells. Each instance B_i of a cell B in cell A is a pair $(T^r_B, <$ *cell definition of* $B >)$[3]. $T^r_{B_i}$ is an affine isometry called transform of B_i in A and $<$ *cell definition of* $B >$ is a reference to the cell definition of B. Any affine transformation T^r_B can be decomposed into two parts $T^r_B = (L^r_B, O^r_B)$. L^r_B is called the location of B in A and O^r_B is called the orientation of B in A (vectorial isometry of T^r_B). The effect of calling B in A at transform $T^r_B = (L^r_B, O^r_B)$ is that of performing the vectorial isometry O^r_B on B, placing the origin of B at location L^r_B within the coordinate system of A, and finally adding to A the collection of objects in B (see figure 6-4).

Interfaces

If instances of cells B and C are called together in the same coordinate system (same cell) then B and C have an interface between them. The interface is an affine transformation that defines the relationship between the transforms of T^r_B of the instance of B and T^r_C of the instance of C. The interface defined by the transforms T^r_B and T^r_C is $I_{BC} = (T^r_B)^{-1}T^r_C$ (since T^r_B is an isometry its inverse is well defined).

Cells B and C may have more than one defined interface between them. In this case the interfaces are labeled $I^1_{BC}, I^2_{BC}, I^3_{BC}$ *etc.* When the index number of the interface is omitted as in I_{BC} the first interface I^1_{BC} is assumed. In order to keep the number of defined interfaces between cells low, only interfaces between cells in which the bounding boxes of the instances intersect are considered. With this scheme the number of defined interfaces between any pair of cells is usually not more than two or three. Interfaces

[3]The r superscript denotes that the transform is relative to the calling coordinate system.

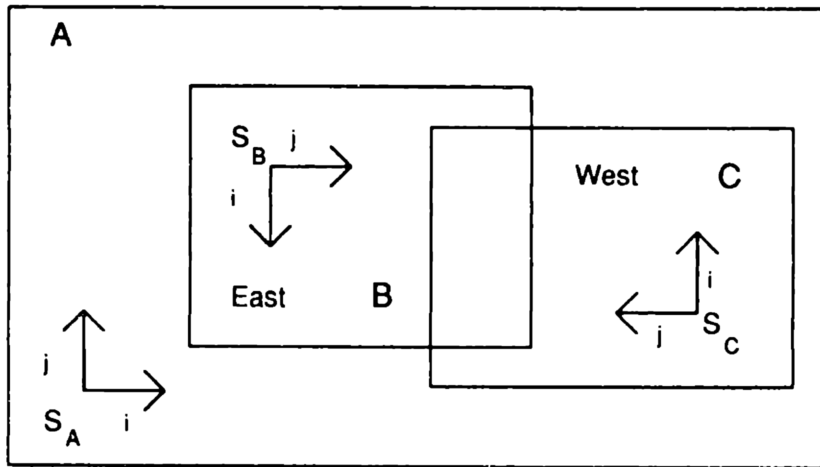


FIGURE 6-5: Interface between A and B

between cells B and C can be defined `by-example` simply by creating instances of B and C in some other cell A.

The relative positions of instances in a well-formed template are necessarily *legal* (and useful) interfaces. It is therefore assumed that all the relative positions of instances in templates of $\mathcal{T}$ are defined interfaces.

Example

Figure 6-5 shows instances of cells B and C called within cell A. The transform of B is $T_B^r = (L_B^r, East)$ and the transform of C is $T_C^r = (L_C^r, West)$. This defines an interface I_{BC} between cells B and C. The interface I_{BC} is the transform C would have if B and C had the same relative position but B was called with the identity transform $I_{BC} = (T_B^r)^{-1} T_C^r = (V_{BC}, O_{BC})$ which turns out to be $V_{BC} = rotate_west(L_B^r - L_C^r)$ and $O_{BC} = South$.

Connectivity Graphs

A connectivity graph $G = (V_G, E_G, W_G)$ ($E_G \subseteq V_G \times V_G$ and W_G is a mapping from edges in E_G to interfaces) is a directed graph in which the vertices V_G represent instances of cells and the edges E_G represent predefined interfaces between them. Each edge e is labeled with $W_G(e)$ which is a symbol representing a transform which must be the transform of one of the predefined interfaces between the cells of the vertices[4].

For each *well formed* connectivity graph there exists one corresponding layout. Figure 6-6 shows the graph and layout equivalents in the case of a three instance connectivity

[4]The edges of the graph must be directed. The transform is directed from the source to destination in the direction of the arrow of the edge.

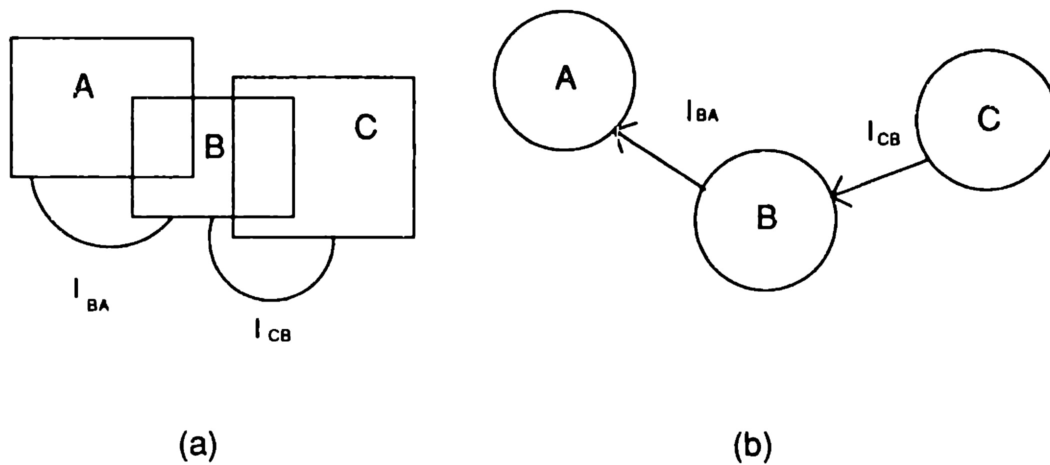


FIGURE 6-6: Graph and Layout Equivalents

graph. The celltype of the vertex is indicated inside each vertex circle. A unique identifier for each vertex is usually omitted except when absolutely necessary.

A connectivity graph is transformed into its layout equivalent by choosing a *root* vertex in the graph and placing the corresponding instance at some arbitrary transform (usually the identity transform). The graph is then traversed and the instances corresponding to each of the vertices are placed relative to one another. The transform T_{B_i} of an instance B_i can be computed from the transform T_{A_j} of a neighboring vertex instance A_j which has already been traversed and the interface I_{AB} which labels the edge between the two vertices in the graph[5] by the relation $T_{B_i} = T_{A_i} I_{AB}$. By traversing all the vertices in the graph all the instances are properly placed relative to each other.

It is assumed that no vertex has two edges with the same label connected to it. Suppose vertex v_i of celltype A has two edges both labeled I_{AB} to vertices v_j and v_k respectively (necessarily of type B). The layout of v_j and v_k will necessarily coincide by falling exactly on top of each other. Hence the layout of either v_j or v_k can be eliminated. Making sure that no vertex has two edges with the same label can easily be checked during graph generation.

The concise notation "two connectivity graph vertices *intersect*" will be used to denote that the bounding boxes of the instances corresponding to those vertices intersect.

If I_{BC} is an interface between B and C then $I_{CB} = I_{BC}^{-1} = (T_C^r)^{-1} T_B^r$ is an interface between C and B. One of these two interfaces is chosen and all graphs are expressed solely in terms of that interface. For example, if I_{BC} is chosen then a edge directed from B to C labeled I_{BC} is always used in lieu of an edge directed from C to B labeled I_{CB}. Two relations *subgraph* and *graph isomorphism* are defined for connectivity graphs.

[5]Assumes that the direction of the edge is from A to B otherwise $I_{BA} = (I_{AB})^{-1}$ is used.

Definition 14 *Given two connectivity graphs* $G_{sub} = (V_{sub}, E_{sub}, W_{sub})$ *and* $G = (V_G, E_G, W_G)$, G_{sub} *is a* **subgraph** *of* G *if and only if* $V_{sub} \subseteq V_G$.

Definition 15 *Two graphs* $G_1 = (V_1, E_1, W_1)$ *and* $G_2 = (V_2, E_2, W_2)$ *are* **isomorphic** *if and only if there is a one to one mapping* $f_G : V_1 \rightarrow V_2$ *such that* $e_{ij} = (v_i, v_j) \in E_1 \Leftrightarrow e_{kl} = (v_k, v_l) = (f_G(v_i), f_G(v_j)) \in E_2$ *and* $W_1(e_{ij}) = W_2(e_{kl})$.

6.5 Connectivity Graph based Layout Verification

In this section, the verification of layouts by templates is formalized using connectivity graphs. Given a connectivity graph, this method can be used to verify whether the layout corresponding to the connectivity graph belongs to the range space of a set of templates $\mathcal{T}$. The templates themselves are also manipulated in terms of their connectivity graph representation. The connectivity graph for the layout to be verified can be input directly to the verification system from a cell assembler such as RSG or can be automatically extracted from a layout.

6.5.1 Differences between Layouts and Connectivity Graphs

There is no one to one correspondence between the set of connectivity graphs and the set of layouts. Several differences between connectivity graphs and layouts are to be noted. There are layouts in L_c for which there are no corresponding connectivity graphs, there are connectivity graphs for which there are no layouts and there are layouts for which there are several connectivity graphs.

Layouts for which there are no Connectivity Graphs

All connectivity graphs have predefined interfaces on their edges. Therefore layouts in L_c for which the relative position of some of the instances cannot be expressed in terms of one of the predefined interfaces to other instances cannot be represented by connectivity graphs. However, since all the relative positions of instances in a set of templates $\mathcal{T}$ are defined, such a layout is necessarily not in the range space of $\mathcal{T}$. If such a layout is presented to the verification system, the system will fail to extract a connectivity graph which indicates that the layout is not in the range space of $\mathcal{T}$.

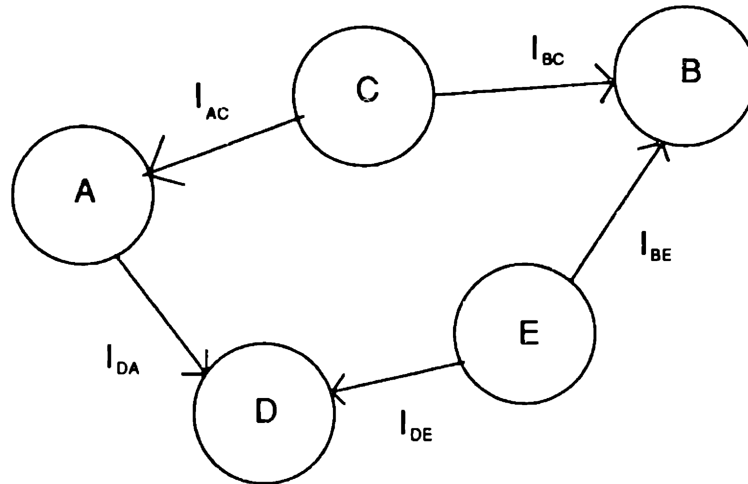


FIGURE 6-7: Cycles in the Graph

Cycles in the Connectivity Graph

Cycles in the connectivity graph[6] can contain conflicting instance placement constraints and cause the graph to have no layout equivalent. Consider the case of figure 6-7. If T_A is the transform of A then T_B the transform of B can be computed from the top path by $T_B = I_{BC}(I_{AC})^{-1}$ and from the bottom path by $T_B = I_{BD}(I_{ED})^{-1}I_{DA}$. This requires that $I_{BC}(I_{AC})^{-1} = T_B = I_{BD}(I_{ED})^{-1}I_{DA}$.

If $I_{BC}(I_{AC})^{-1} \neq T_B = I_{BD}(I_{ED})^{-1}I_{DA}$ then there are two conflicting constraints for the placement of B and therefore the graph has no layout equivalent. Such conflicts in the graph can be easily detected by performing the graph to layout transformation described in section 6.4.1.

Multiple Graph Representations for a Layout

Let C_G be a connectivity graph for layout L with cycles which contain no placement conflicts. From C_G several other graphs representing the same layout can be derived. Because cycles in the graph C_G contain redundant placement information, any graph derived from C_G by removing an edge in a cycle is also a connectivity graph for L. In fact any *spanning tree* derived by removing edges from C_G will also be a connectivity graph for L. For example, the graph of figure 6-8 (a) and the two spanning trees derived from it shown in figures 6-8 (b) and (c) all represent the same layout.

[6]Cycles in the underlying undirected graph.

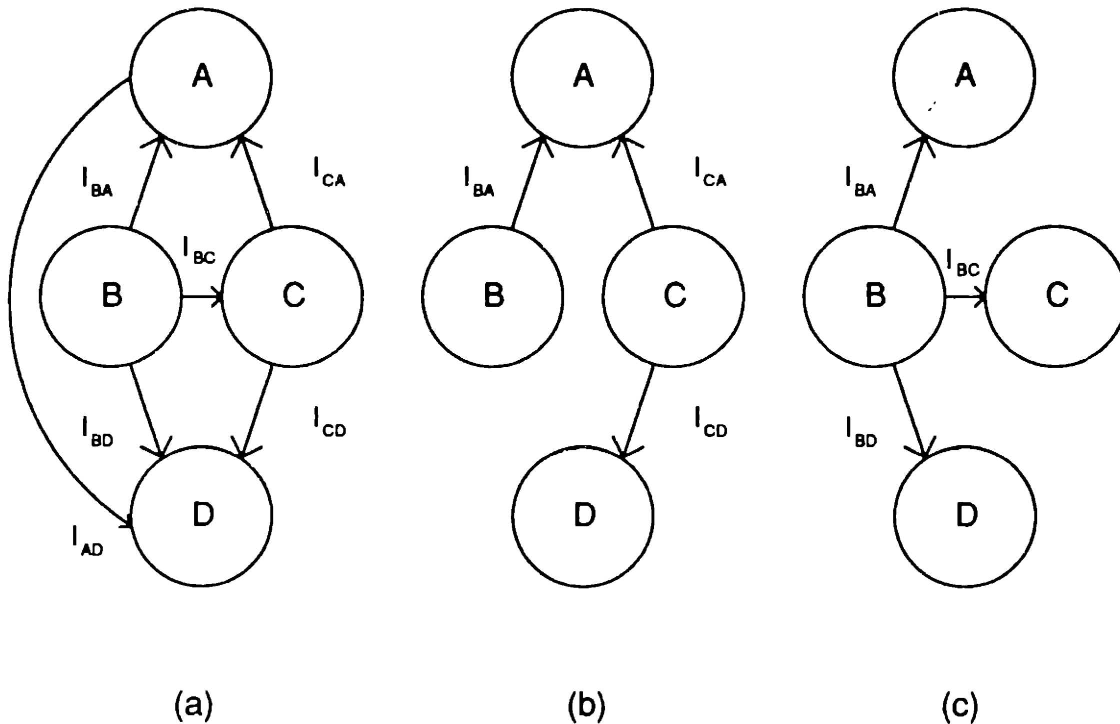


FIGURE 6-8: Equivalent Graphs

6.5.2 Normalizing the Graph Representation

In this section a *normal form* for connectivity graphs is described. For each correct layout L_i, there is at most one connectivity graph in *normal form* that is equivalent to it. The motivation for introducing a normal form is that during the verification process the layout is tiled with templates from a set of templates $\mathcal{T}$. This corresponds to *tiling* the graph of L_i with the graphs of the templates. This operation is greatly facilitated if both the graph for the layout and the graphs for the templates are in normal form. Furthermore, in chapter 8 a method for verifying the correspondence between a layout represented by its connectivity graph and a schematic is described. The methods employed by this latter technique also require that the connectivity graphs be in a normal form.

Definition 16 *A connectivity graph is in a normal form for a set of interfaces $\mathcal{I}$ if it is maximally connected. Maximally connected means that for any two instances in the graph, whose relative position corresponds to an interface I_i between the two corresponding cells, there is an interface edge labeled I_i between the two instances.*

In order to keep the number of edges in the graph as low as possible a minimal set of interfaces $\mathcal{I}$ is retained. The techniques described in chapter 8 for verifying that a schematic and a layout are compatible, require any two instances whose layouts interact

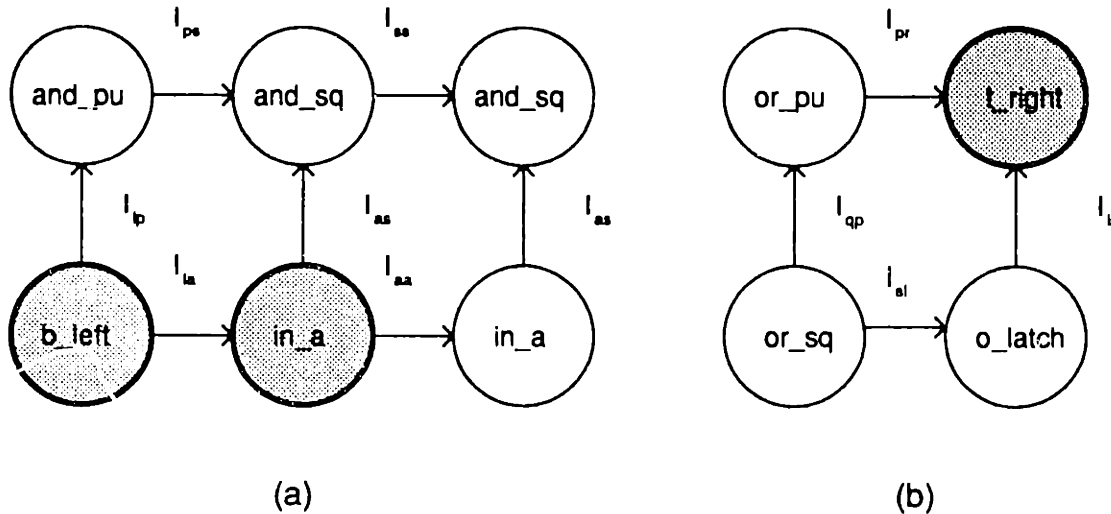


FIGURE 6-9: Graph Representation for Templates

electrically to have an interface between them. The layouts of two instances interact electrically if the underlying netlists of each of the instances share some electrical nets or if transistors are formed in the overlapping region of the two instances. The set of interfaces $\mathcal{I}$ must therefore include any interface that occurs in well-formed layouts (according to a set of templates L_i) for which the instances interact electrically. This boils down to making sure that in any template in T_i, for any two instances that interact electrically, the corresponding cells have a corresponding interface between them.

A graph with interface edges in $\mathcal{I}$ can be turned into normal form by performing a graph to layout transformation and for each instance I_i computing the set of instances its bounding box intersects with. For each instance I_j that intersects with I_i and whose relative position with I_i corresponds to an interface in $\mathcal{I}$, an interface edge is added if it does not already exist.

6.5.3 Template Occurrences in Connectivity Graphs

Normalized Graph Representation for Templates

Let T_i be a template and $C_i = (V_i, E_i, W_i)$, be the normalized connectivity graph representing the layout of T_i. $T_i^G = (V_i, V_i^0, E_i, W_i)$, where V_i^0 is the set of vertices of C_i corresponding to the instances in the interior of T_i, is a normalized graph representation of template T_i.

Figures 6-9 (a) and (b) show the graph representation of the templates in figures 6-3 (a) and (b). The vertices corresponding to the interior of the templates of figures 6-3 (a) and (b) (the vertices in V_i^0) are shaded.

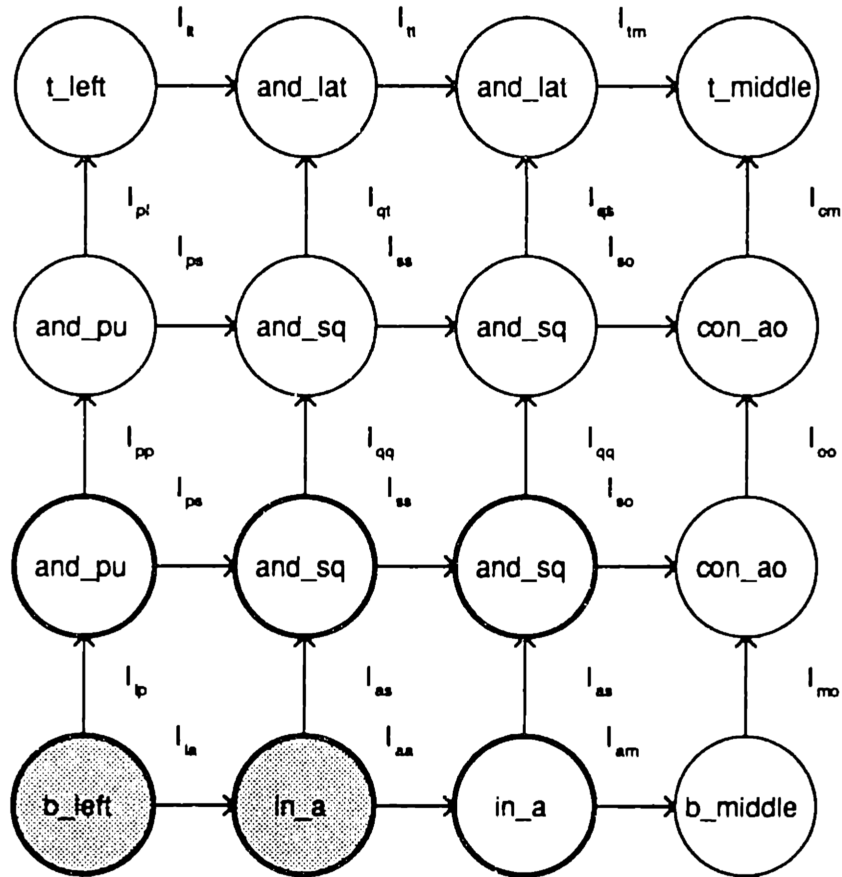


FIGURE 6-10: Template Occurrence

Template Occurrence

A condition on a normalized connectivity graph C, which is equivalent to the occurrence of a template T_j in the layout L_i of the graph, is now described. The graph C_i is assumed to have no placement conflicts due to cycles. Let $T_j^G = (V_j, V_j^0, E_j, W_j)$ be the normalized graph representation of T_j and $C_j = (V_j, E_j, W_j)$ the graph for the layout of T_j. Let C_{sub} be a subgraph of C isomorphic to C_j such that the bounding box of any vertex in C_{sub} corresponding to a vertex in V_j^0 intersects only with bounding boxes from vertices in C_{sub}. C_{sub} is said to be an occurrence of T_j^G in C. The vertices of C_{sub} corresponding to vertices in V_j^0 are said to be the interior vertices of the template occurrence.

Figure 6-10 shows an occurrence of the template of figure 6-9 (a) in a graph. The vertices corresponding to the template occurrence have thicker circles and the vertices in the interior of the occurrence are shaded.

Theorem 5 *A vertex v_j of a normalized connectivity graph C_i is in the interior of an occurrence of T_j^G if and only if its corresponding instance i_j of the layout L_i of C_i is in the interior of an occurrence of T_j.*

Proof 5 *If vertex v_j is in the interior of an occurrence of T_i^G then i_j is necessarily part of the layout of T_i at the position of an interior instance. Furthermore, since by definition of the interior of an occurrence of T_i^G, the bounding box of i_j intersects only with the bounding boxes from instances in T_i, i_j is in the interior of an occurrence of T_i.*

Conversely, if i_j is in the interior of an occurrence of T_i then v_i is in the normalized graph of the instances corresponding to the occurrence of T_i. This subgraph G_{sub} of C_i is necessarily isomorphic to the graph of T_i^G because both graphs are normalized. Since i_j intersects only with instances in the template occurrence, v_j is necessarily in the interior of an occurrence of T_i^G.

6.5.4 Criteria for Connectivity Graph Correctness

A criteria on a connectivity graph C_i equivalent to its corresponding layout L_i being in the range space of a set of templates $\mathcal{T}$ is now described.

Theorem 6 *Given a normalized connectivity graph C_i with no conflicting cycles, its corresponding layout L_i, a set of templates $\mathcal{T}$ and their corresponding normalized graph representations $\mathcal{T}^G$: L_i is in the range space of $\mathcal{T}$ if and only if each of the vertices of C_i is in the interior of an occurrence of a template in $\mathcal{T}^G$.*

Proof 6 *From theorem 5, a vertex v_i is in the interior of a graph template T^G if and only if its layout is in the interior of the corresponding template T. Since each vertex in C_i corresponds to an instance in L_i, each vertex C_i is in the interior of a template in $\mathcal{T}^G$ if and only if each instance in L_i is in the interior of a template in $\mathcal{T}$.*

6.5.5 Dealing with Encoded Cells

Often cells from a cell library can be grouped into classes of cells with similar layout characteristics. For example, up to eight essentially similar yet different full adder cells are used in the array multiplier of figure 5-13[7] as described in [10]. One technique often used to deal more effectively with such families of cells is to have one basic cell for the entire family and *personalize* it by superimposing the layout from one or several *encoding* cells.

In the PLA of figure 6-2 each **or_sq** cell spans two outputs and two product terms. Cell **or_sq** can be personalized by superimposing 0, 1, 2, 3 or 4 **o_t** cells. The presence

[7] This figure does not differentiate between the different celltypes.

of an o_t cell creates a transistor at one of the 4 transistor sites of or_sq. The PLA is encoded by adding o_t instances in accordance with the PLA's truth table.

Let T_i be a template in which a vertex v_j of type or_sq is an interior vertex. Since v_i is in the interior of T_i all the encoding vertices of celltype o_t must be part of the template (because they intersect with v_i). Since there are $2^4 = 16$ possible encoding configurations there must be at least sixteen templates with vertices of celltype or_sq in their interior. In general the sixteen possible encodings of or_sq will lead to a sixteen fold increase in the number of such templates.

In order to reduce the number of templates, encoded cells are handled differently. Conceptually the encoded vertex and its encoding vertices (for a legal encoding configuration) are first abstracted to a new vertex in a fashion similar to a GRASP network reduction. The new vertex can again be abstracted to another vertex with some additional encoding or can be used in a layout template once the abstraction process is complete. In this fashion the verification of the encoded vertex is performed in two independent parts, first the encoding configuration is verified then the *setting* of the encoded cell system in the rest of the layout is verified.

A similar effect can be achieved within the existing framework. Encoding vertices are treated as if their bounding box regions have *zero dimensions* and hence are incapable of intersecting with other vertices. Templates which contain encoded cells in their interior are then specified without the encoding.

If v_i is the vertex of an encoded cell which appears in the interior of an occurrence of T_j, then adding vertices that encode v_i to the layout will not cause v_i to cease being in the interior of the occurrence because the encoding vertices will be treated as if they do not intersect with v_i. Additional templates with the encoding cells in their interiors are needed to verify the encodings themselves.

Figure 6-11 (a) shows a template in which an instance of or_sq is in the interior. Occurrences of this template are not influenced by the encoding of the interior instance. figures 6-11 (b) (c) (d) and (e) show the templates used to verify the encoding instances. For the or_sq celltype the four encoding instances can be added independent of one another. Templates can of course be designed to select a few legal encoding configurations. Hypothetically if the top left encoding is legal only if the bottom two encodings are present then one might use a template like the one of figure 6-11 (f).

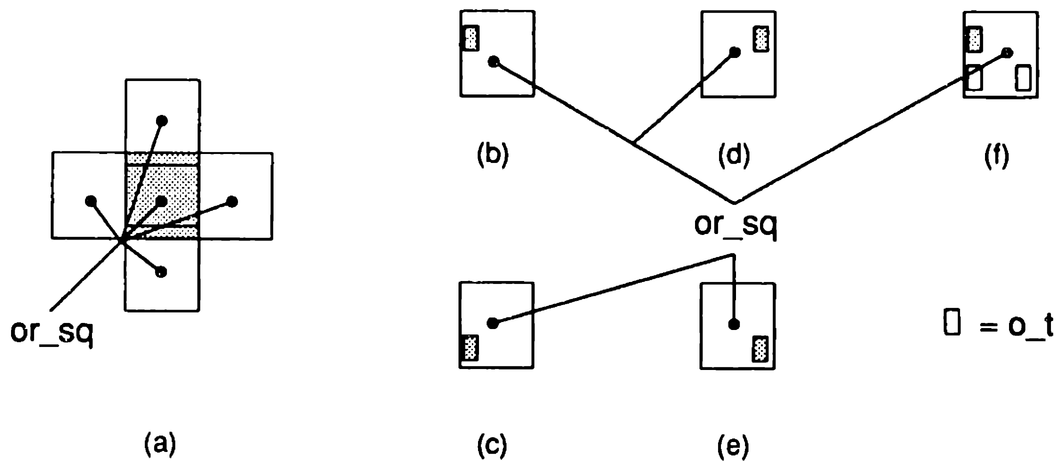


FIGURE 6-11: Encoded Cell Templates

7

Layout Verification Algorithm & Implementation

7.1 Verification Algorithm

7.1.1 Overview

The verification techniques described in chapter 6 have been implemented in a computer program called GLOVE (Graph-based Layout Verifier). The GLOVE verification algorithm is composed of five parts as shown in figure 7-1. In the first step, the connectivity graph of the layout is checked for illegal cycles. In the next step, for each vertex v_i in the graph the set of vertices whose bounding box intersects with that of v_i is computed. Next the graph is normalized by adding edges as described in section 6.5.2. Then, a template mapping algorithm finds occurrences of graph templates of $\mathcal{T}_i^G$ in C_i and labels each vertex that appears in the interior of at least one template occurrence. Finally, the list of vertices that do not appear in the interior of a template occurrence are reported as errors in the layout.

The template mapping algorithm is similar in essence to GRASP's event driven parsing algorithm. The main difference is that the connectivity graph does not get reduced i.e. the vertices of the graph remain the same. Also the layout verification algorithm is a *degenerate* event driven algorithm in the sense that during execution events get processed but no new events are created.

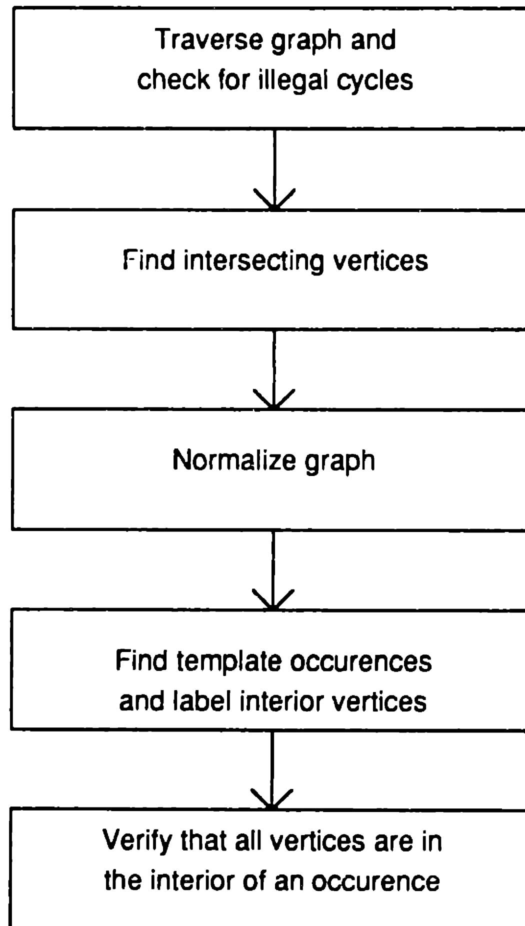


FIGURE 7-1: Components of the Algorithm

7.1.2 Preparing the Graph

Checking for Illegal Cycles

Cycles in the graph can easily be detected by performing a graph to layout transformation. During this transformation the graph is traversed in *depth first*[1] order and each newly visited vertex instance v_i is placed relative to one of its neighboring vertices which has already been visited. If the placement of each newly visited vertex v_i does not conflict with any of its other neighboring vertices that have already been placed, the graph is necessarily free from illegal cycles.

Finding Intersecting Vertices

Once the graph has been traversed and each of the vertices have been given a transform, the locations of their bounding boxes are defined. If n is the number of vertices in the graph, efficient algorithms such as [11], [33], [45] can be used for reporting the set of all intersecting bounding boxes in $O(n \log(n))$ time and $O(n)$ or less space. After this operation each vertex is provided with the list of the vertices that it intersects and the vertex can furnish this list upon request.

Normalizing the Graph

A traversed graph can be normalized by examining each pair of intersecting vertices. If the relative placement of the two vertices corresponds to one of the interfaces I_j between the two corresponding cells, then an interface edge labeled I_j is added between the two vertices if it does not already exist.

7.1.3 Finding Template Occurrences

Overview

The template mapping algorithm enumerates all the vertices in the graph and for each vertex v_i searches for all possible template occurrences in which v_i appears. Since the reader has been introduced to the terminology for event driven parsing described in section 5.1, the template mapping algorithm will be presented as a special case of GRASP's event driven algorithm. In this *degenerate* case of the event driven algorithm each event corresponds to a vertex in the graph. An event for each vertex in the graph is generated when the graph is read into the program and no new events are generated

[1]Or most other reasonable graph traversal schemes.

during verification. The event queue model is also especially useful for describing how the algorithm handles incremental modifications to the connectivity graph.

Servicing an Event

Events are serviced in a fashion similar to GRASP. Servicing an event with associated vertex v_i requires removing it from the event queue and finding **all** the subgraphs in G_i (in which v_i appears) which are isomorphic to the graph of a template T_j in $\mathcal{T}$. Each subgraph G_i is then examined to see if it can be the occurrence of template T_j. Finally the vertices in the interior of template occurrences are marked to indicate that they have appeared in the interior of some occurrence.

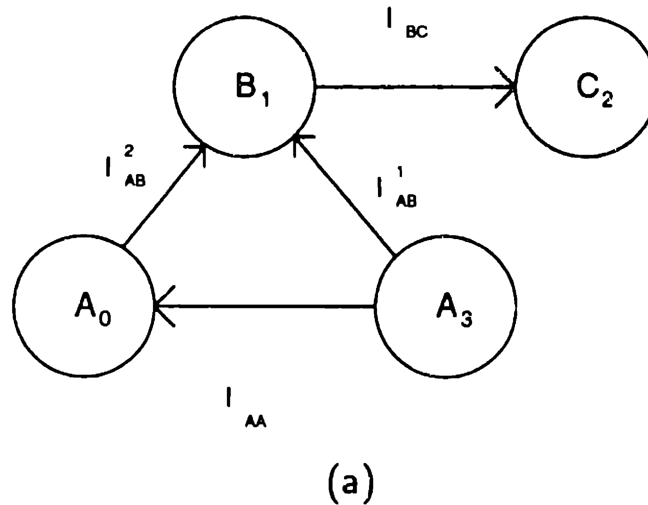
Finding a Match

In order to determine if the event vertex v_i of celltype A belongs to a subgraph isomorphic to the graph of a template the following operation is performed. For each position in which a cell of type A appears in the graph $G^0_{T_i}$ of a template T_i, the algorithm checks to see if v_i appears in a subgraph G_{T_i} isomorphic to $G^0_{T_i}$ at the same position (the concept of position in a graph is introduced in section 5.1.2). The verification algorithm maintains a list of positions in which cells of type A appear in template graphs (like in the GRASP parsing algorithm) and sequentially tries each position in the list for a match.

Determining if vertex v_i is in position P is performed using the search algorithm described in section 5.1.3 which implements an efficient depth first search of the graph surrounding the event vertex. The algorithm consists of two parts as described in section 5.1.3.

In the first part a template graph and a position in the graph are transformed into a set of instructions for determining if a vertex v_i in the graph appears in that position. The form of the instructions is $I = (I.interface_type, I.vertex_source_slot, vertex_destination_slot)$. Figure 7-2 (a) shows a template graph and figure 7-2 (b) the set of instructions for the position of vertex 0.

Given a vertex v_i and a sequence of instructions for position P (generated by part 1), the second part of the algorithm interprets the instructions to verify that v_i appears in position P in the graph. The procedure used to interpret the instructions is similar to the procedure described in figure 5-5 and is shown in figure 7-3. This procedure makes use of the fact that a vertex cannot have two edges with the same label and hence never backtracks. Therefore if any instruction at any *level of depth* in the search results in failure the whole process is terminated and the *top level* call to execute_vertex_instruction



(a)

Instructions for vertex 0

1	(Begin,0)	*Put the event vertex in slot 0*
2	$(I^2_{AB}, 0, 1)$	*Find the vertex connected to vertex in slot 0 via a forward edge labeled I^2_{AB}. Put the vertex in slot 1*
3	$(I_{BC}, 1, 2)$	*Find the vertex connected to vertex in slot 1 via a forward edge labeled I_{BC}. Put the vertex in slot 2*
4	$((I^1_{AB})^{-1}, 1, 3)$	*Find the vertex connected to vertex in slot 1 via a backward edge labeled I^1_{AB}. Put the vertex in slot 3*
5	(Done)	*All cells have been found and put in their proper slots. Check that the graph formed by the vertices in the slots constitutes an occurrence of the template and if so flag vertices in the interior of the occurrence.*

(b)

FIGURE 7-2: Graph and Corresponding Instructions

```
Procedure execute_vertex_instruction(instruction)
 if is_first_instruction(instruction)
      insert_vertex_in_slot(the_event_vertex,
                            instruction.vertex_destination_slot)
      execute_vertex_instructions(next_instruction(instruction))
 else if is_last_instruction(instruction)
      mark_interior_vertices_if_applicable()
      return(success)
 else  For each edge e of (vertex_slots[instruction.vertex_source_slot)
          If ((label_of(e) == instruction.interface_type) &&
              (direction_of(e) == direction_of(instruction.interface_type)
              insert_vertex_in_slot(other_vertex_of_edge(e),
                                    instruction.vertex_destination_slot)
              return(execute_vertex_instructions
                       (next_instruction(instruction)))
          return(failure)
 return(failure)
```

FIGURE 7-3: Procedure execute_vertex_instruction

results in failure.

Verifying that the Match is an Occurrence

When a subgraph G_{T_i} isomorphic to the graph of a template T_i has been found the vertices corresponding to the interior of T_i are identified. For each such vertex v_j the set A_{v_j} of its intersecting vertices is examined. If this set contains vertices not in G_{T_i} then G_{T_i} cannot be an occurrence of T_i because any vertex in the interior of a template occurrence can intersect only with other vertices in the occurrence. If for all vertices v_j in G_{T_i}, corresponding to the interior of T_i, A_{v_j} contains vertices only in G_{T_i} then G_{T_i} is an occurrence of T_i and all the vertices v_j in the interior of the occurrence are marked by a special flag to indicate that they have appeared in the interior of an occurrence.

Increasing Verification Speed

In order to increase verification speed each vertex maintains a list of positions (called the *occurring positions* list) in which it appears in a template occurrence. Besides increasing verification speed the list is also used during incremental updates to the graph. The elements of the list are vertices of template graphs. For example, vertex `b_left` in figure 6-10 *remembers* that it has appeared in a template occurrence of the template of

figure 6-9 in the position of vertex `b_left` by having that vertex in its *occurring positions* list. Every time a template occurrence is identified each vertex in the template graph is added to the list of *occurring positions* of the corresponding vertex in the occurrence.

During the servicing of an event with associated vertex v_i, before an element P in the positions list is processed it is first verified that v_i does not already appear in a template occurrence in that position. This is achieved by examining the *occurring positions* list of v_i and verifying that the vertex in the template graph for position P has not already appeared.

Because a vertex cannot have two edges labeled with the same interface v_i, for a given template graph G^O_{sub} there is at most one graph G_{sub} isomorphic to G^O_{sub} in which v_i appears in position P. Hence v_i can appear at most once in position P . Therefore if v_i already appears in position P no other occurrences in which v_i appears in that position can be found and processing that position in the position list can be omitted at a considerable saving in computation time.

For example, suppose that v_i, v_j, v_k, v_l, and v_m form the occurrence of a template T_α with positions p_i, p_j, p_k, p_l, and p_m respectively. Without loss of generality it is assumed that the event for v_i was the first to be processed. When the events for v_j, v_k, v_l, and v_m respectively are processed positions p_j, p_k, p_l, and p_m respectively will not be processed at a considerable saving in computation time.

7.1.4 Algorithm Complexity

The complexity of the verification algorithm is the sum of the complexities associated with its various components. The component which reports the set of intersecting bounding box pairs has the highest time complexity ($O(n\log(n))$). The number of bounding boxes that need to be examined however is much smaller than the number of polygons that need to be examined in a *regular* DRC program. The execution speed of the algorithm is therefore much faster than that of a *regular* DRC program.

Traversing the Graph and Finding Intersecting Boxes

Since the number of interface types any celltype can have is fixed and a vertex cannot have more than one edge labeled with a given interface type, the number of edges any vertex can have is bounded. Therefore if n_{vertex} is the number of vertices in the graph, the time required to traverse a vertex and check for loops is $O(1)$ and thus the time required to traverse the graph and check for loops is $O(n)$ Efficient algorithms such as [11], [33], [45] can be used for reporting the set of all intersecting bounding boxes in $O(n\log(n))$

time.

Finding Template Occurrences

Let T_{max} be the maximum number of vertices in a template graph. Since a vertex cannot have more than one edge labeled with a given interface type there is no backtracking in procedure execute_vertex_instruction and therefore the *top level* call to execute_vertex_instruction returns in time $O(T_{max})$. The time required to find a potential occurrence is therefore $O(T_{max})$.

Once a subgraph C_{sub} isomorphic to the template graph of T_i is found it remains to verify that the conditions for the internal vertices of the potential occurrence are met. For each such vertex v_i it must be verified that v_i intersects only with the vertices in C_{sub}. During this process at most T_{max} other vertices need to be examined because if more than T_{max} vertices intersect with v_i, then v_i necessarily intersects with a vertex not in C_{sub} and therefore C_{sub} is not an occurrence. Since there are at most T_{max} vertices in the interior of an occurrence this procedure is repeated for at most T_{max} vertices v_i. Therefore the time taken to verify that C_{sub} is an occurrence of T_i is $O(T_{max}^2)$.

If there are at most L_{max} positions in the positions list the time required to process an event is $O(L_{max} \cdot (T_{max} + T_{max}^2)) = O(L_{max} \cdot T_{max}^2)$. Therefore if n_{vertex} is the number of vertices in the graph, the time required to process all the events in the event queue is $O(n_{vertex} \cdot L_{max} \cdot T_{max}^2)$ which is linear in the size of the graph.

7.1.5 Incremental Update

The verification algorithm in GLOVE can be augmented to effectively deal with incremental additions and deletions to the connectivity graph. The graph in turn can be incrementally updated to reflect modifications in the layout it represents.

Adding a Vertex

If a vertex v_i is added to the connectivity graph of a layout, the transform of the new vertex can be computed in terms of the transform of its neighboring vertices. The newly augmented graph can be checked for illegal cycles by verifying that v_i has no placement conflicts with its neighbors. The set of intersecting vertices for v_i can then be easily computed and the graph normalized by adding edges between v_i and its intersecting vertices. Then by placing v_i on the event queue and enabling the processing of events, any occurrences in which v_i appears can be found.

Deleting a Vertex

If a vertex v_i is deleted from the connectivity graph all occurrences in which v_i appears must be *removed*. The vertex v_i maintains a list of *occurrence positions* in which it appears. For each position p_i in this list the vertices v_j, v_k, $\cdots v_z$ with positions p_j, p_k, $\cdots p_z$ respectively in the same occurrence can easily be found. For each of these vertices v_α the corresponding *occurrence position* p_α must be removed from its *occurrence position* list. Any vertices no longer in the interior of an occurrence are then flagged as errors.

7.1.6 Error Reporting

When the event queue becomes empty, all possible template occurrences have been found. Therefore any vertices that do not have their *interior flag* set cannot appear in the interior of a template. The regions corresponding to the bounding boxes of these vertices cannot be defined as well-formed in terms of the set of templates used to perform the verification. The regions of layout corresponding to these vertices are then reported back to the user as errors.

For each vertex v reported back to the user as an error and for each possible *occurrence position* p_i in which v would be in the interior of the occurrence, it is possible to generate an explanation of why v cannot be in a template occurrence in position p_i. Typically for each vertex v the set of such positions is small (usually less than three positions) and an explanation of why v cannot appear in each of these positions can be used effectively to understand why the error has occurred.

7.2 Implementation

Source Code

GLOVE is written in C and runs on an HP 9000 series model 350 workstation running UNIX and Xwindows. The source code is heavily borrowed from GRASP and consists of approximately 12000 lines of code out of which the core verification algorithm takes 2500 lines of code. As with GRASP, the code makes extensive use of data abstractions. GLOVE first reads in the templates, then the connectivity graph to be verified, proceeds with the verification and finally produces a list of vertices not in the interior of an occurrence. This version of the code uses a simple $O(n^2)$ algorithm to compute the set of intersecting bounding boxes and hence a speed penalty is incurred for large layouts.

```
Number_of_cells: 4
0 A /b 3 1 /f 1 2 ;
1 B /b 0 1 /b 3 1 /f 2 1 ;
2 C /b 1 1 ;
3 A /f 1 1 /f 0 1 ;
```

FIGURE 7-4: Textual Representation of a Connectivity Graph

Input Files

The templates are input graphically `by-example` [10]. Figure 6-3 resembles the input specification of a template. The layout for each template is provided to GLOVE as a ".DEF" format[2] file generated by the layout artwork editor HPEDIT [5]. From this layout GLOVE first extracts the interfaces present using RSG interface labeling conventions and then proceeds to generate the template graph.

The layout is input as a connectivity graph which is assumed to be already in normal form. Figure 7-4 shows a textual representation of the graph of figure 7-2 (a). A line such as "`0 A /b 3 1 /f 1 2 ;`" signifies that vertex 0 is of type A and is connected to vertex 3 via a backward edge labeled I^1_{AA} and to vertex 1 via a forward edge labeled I^2_{AB}.

Graphics Display

An Xwindow graphics interface built into the program can display either a template graph or a region of the graph to be verified centered around the event vertex. For every event vertex a display can be generated for each occurrence in which the event vertex appears. The vertices in the occurrence, the interior of the occurrence and those not in the occurrence are given different colors. The event vertex always appears at the top left corner of the screen. The graph is displayed in a two column format as shown in figure 7-5 which is a display of the template of figure 6-9 (a). The interior vertices, shown in a different color on the screen, are the shaded region. The edge directions are shown only between vertices of the same celltypes and the direction of the other edges is implied. Next to each edge the index of the interface between the two vertices is shown. For example, the edge between the two top cells is labeled "1" which signifies that this edge represents the first interface I^1_{ab} between cell types A and B.

[2] Very similar to the CIF format.

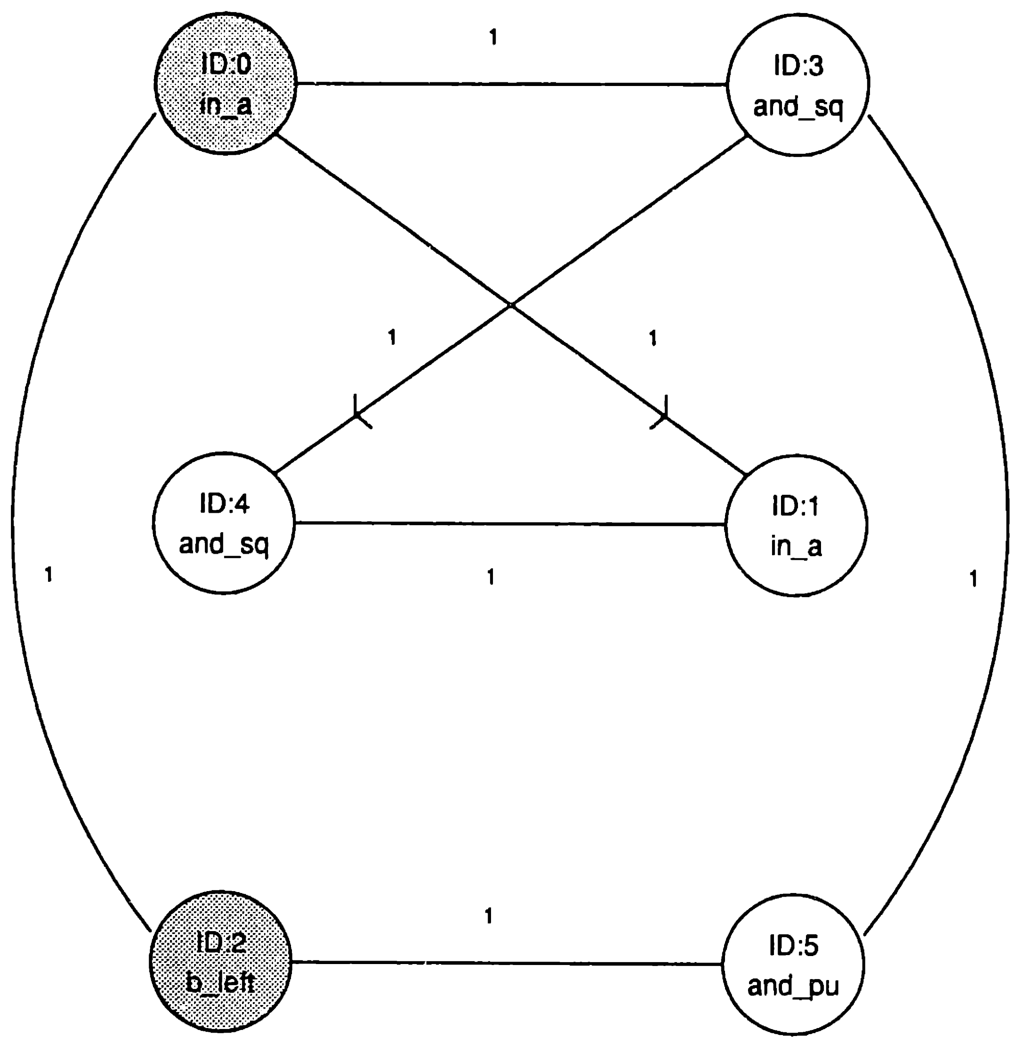


FIGURE 7-5: Xwindows Graph Display

PLA size	Number of instances	Verification time
10 × 20 × 20	537	3s
20 × 40 × 40	1993	12.5s
40 × 80 × 80	7668	51s
60 × 120 × 120	17142	117s

Table 7-1: Verification Times

7.3 Experiments

Templates for the PLA style of figure 6-1 have been designed and tested for various PLA sizes and encodings. The PLA templates make use of the special handling of *encoded cells* (described in section 6.5.5) provided by GLOVE. The template set consists of 25 templates (out of which 8 are for encoding the **and_sq** and **or_sq** cells) ranging in size from 2 to 10 vertices. The library cells for the PLA are taken from RSGs PLA library. The templates are derived by using RSG to build the layout for a small PLA and the layout editor HPEDIT [5] to *carve out* templates from that layout.

Tests have been run to evaluate the verification speed of GLOVE. For this purpose a PLA connectivity graph generator program has been written to test GLOVE for various sized and encoded PLAs. Since the regularity of the PLA does not enhance verification speed, the derived results are good indicators of how fast the algorithm will perform on less regular structures.

Table 7-1 summarizes the verification times. The indicated values are the time required to find template occurrences and do not include the time required to compute the set of intersecting vertices. As the PLA gets larger a greater fraction of the computation time is spent computing the bounding boxes. For the 60 × 120 × 120 case, the time spent computing intersections is larger than that spent finding template occurrences. The amount of time spent computing intersecting vertices can be greatly reduced by using an $O(n \log(n))$ instead of an $O(n^2)$ algorithm.

8

Schematic vs. Layout Comparison

8.1 Introduction

To face the challenge of carrying VLSI circuits of ever growing complexity from concept to silicon, design procedures that break the process into many distinct steps have been introduced. By factoring the design process into these steps the complete design is broken up into domains, each of which is a manageable task.

During the design process more and more of the design parameters are settled upon as the design proceeds from concept to mask. Intermediate representations between concept and mask for the design are introduced as an effective means to break up the task of deciding upon these parameters by providing a level of insulation between these tasks. As a result, some of the design steps involve taking a more abstract intermediate representation and producing a corresponding less abstract one.

In some cases the new representation can be mechanically generated from the more abstract one by a set of transformations which guarantee that by construction the new representation does indeed correspond to the more abstract one. Often however such a guarantee is not possible and it becomes necessary to provide a verification mechanism that can validate the new representation by comparing it to the more abstract one.

Schematics described in chapter 3 and layouts described in chapter 6 are two such representations referred to hereafter as domains. The schematic domain is usually considered the more abstract of the two domains. In one of the final steps of a design, a layout for a schematic of the design is generated. When the layout is not generated by a *correct by construction* procedure, it becomes necessary to determine whether the layout does indeed implement the logic circuitry or better still the functionality of the schematic. A solution to this problem is of great importance and as such several techniques have been

developed to cope with it. In this chapter a novel solution with significant advantages over previous methods is proposed.

Ideally a schematic vs. layout comparison system should be capable of reporting whether or not the functionality implemented by the schematic and the layout (given the context in which the circuits will be used) are one and the same. To this end the verification system should tolerate (within reason) differences in the implicit circuitries provided that they both implement the same function.

Simulation is often used as a tool to this end. Through the use of simulation the user can compare the result of computations by the two circuits for various inputs. Here, as in the case of schematic design style verification, simulation can be used to report discrepancies but not find them.

Simulators such as [14] are capable of generating symbolic descriptions for the outputs of a circuit in terms of its inputs and state. In theory such a representation can be generated for the circuit of the schematic and the circuit of the layout and the two representations checked for equivalence. Deciding whether two such representations are equivalent is an extremely difficult NP complete [24] problem making this technique impractical.

The most commonly used technique to solve the schematic vs. layout correspondence problem is to compare the circuit of the schematic with the circuit of the layout. For this purpose, the schematic is usually expanded into a transistor netlist and a node extractor [32] is applied to the layout to produce a netlist. The problem of comparing the two netlists is equivalent to the graph isomorphism problem. Although no known polynomial time algorithm exists for the graph isomorphism problem [16] [35] (it is not known whether this problem is NP complete) good heuristics exist for solving it in reasonable time for most practical circuits.

Among the more commonly used heuristics for solving the correspondence problem are *signature-calculation* [17], [41] (the most widely used method), *path tracing*, and *rule based pattern recognition* systems [39] which are described in the following sections.

8.1.1 Signature Calculation

Signature calculation algorithms attempt to partition a graph (or in this case a netlist) based on computed values for each vertex (respectively circuit element) called *vertex invariants*. A *vertex invariant* is a property of a circuit element that is preserved under netlist isomorphism. If two circuit elements in two isomorphic netlists correspond to each other then they necessarily have the same vertex invariant value. The value of the vertex

invariant of an element is also referred to as its signature.

Examples of simple vertex invariants include functions based on the element type (module type), its degree (the number of edges connected to the vertex), the length of the shortest (or longest) cycle it appears in etc. Vertex invariants partition the circuit elements into equivalence classes of equal signature. Two corresponding elements in two isomorphic netlists must necessarily be in the same equivalence class. The difficulty of checking the correspondence of two netlists is directly related to the number of elements in each class. Ideally each equivalence class is a singleton and a good vertex invariant is one which produces the fewest number of elements in each class. For this purpose each equivalence class can be further refined by recomputing a new signature for each element based on its current signature and the signature of its neighbors. The process may be repeated until no further subdivision is possible. If the two circuits end up with exactly the same equivalence classes and each equivalence class has the same number of elements then the graphs may be isomorphic otherwise they are not isomorphic.

The smallest possible number of elements in each equivalence class is the number achieved by the automorphism (isomorphism onto the graph itself) partition of the graph. By definition of the automorphism partition any element in an automorphism equivalence class A_i^1 of graph G_1 can be matched with any element of the corresponding class A_i^2 of G_2. In practice however, the equivalence classes do not result in an automorphism partitioning[1] and hence each element from an equivalence class E_j^1 of G_1 must be successively paired with an element of its corresponding class E_j^2 of G_2 till a match is found. In theory, the number of such matches that must be tried and undone is $\sum_i Card(E_i)!$. In practice however, the time complexity of good signature based systems is roughly linear in the size of the circuit. Signature based systems have recently been adapted to make use of hierarchy in the netlists [55] [50] (provided that the hierarchy is the same in the two netlists) resulting in greater verification speeds.

8.1.2 Path Tracing

Path tracing is a method used in special circumstances. In this technique a path from a known input or output is traced simultaneously in both circuits. For example, in a circuit with no feedback consisting of logic gates each with a single output that feeds only one other gate, a path can be traced simultaneously in both circuits from an input to an output. This method rapidly breaks down for more general circuits because for a given path in one circuit many potentially equivalent paths in the other might

[1] Even if by chance this did occur it cannot be easily detected.

have to be examined resulting in an exponential time complexity. The advantages of this technique over signature calculation are faster speed and potentially better error reporting in the special cases it is capable of handling. Occasionally path tracing is used in conjunction with signature calculation to discriminate between elements within large equivalence classes. In this scheme the algorithm alternates between path tracing and signature calculation yielding successive refinements of the equivalence sets.

8.1.3 Rule based Pattern Matching

The technique used for schematic vs. layout correspondence verification in the rule based expert system CV [39] resembles the parsing technique described in GRASP. The schematic is input using a hierarchical description. In such a description composite modules are defined in terms of interconnections of simpler ones. The entire schematic, using the same rationale, can be thought of as the definition of one very large module. From this hierarchy the system extracts a set of rules similar to GRASP's productions. For each composite module definition a corresponding rule is generated. The rule for the entire schematic corresponds to a production for the start symbol of a grammar.

Once all the rules have been derived from the schematic, an algorithm similar to grammatical parsing is used to check that the netlist of the layout corresponds to that of the schematic. During this process the netlist of the layout is *reduced* using the rules (productions) extracted from the schematic. If at the end of the process the module corresponding to the entire circuit (start symbol of the grammar) remains then the layout corresponds to the schematic. Since the rules (productions) extracted from the schematic do not necessarily constitute a *deterministic grammar*, rules known as *conflict resolution* rules are used to decide which *reductions* to perform.

8.2 Shortcomings of Existing Techniques

8.2.1 Error Reporting

In the three techniques described in sections 8.1.1, 8.1.2 and 8.1.3, the comparison is performed on netlists of the schematic and layout instead of on the schematic and layout representations themselves. When a mismatch is detected it is reported back to the user as a discrepancy between components or connections in the two netlists. The user must then identify the schematic and layout entities the netlist mismatch pertains to and decide how the mismatches relate to the internal structure of these entities or the connections between them.

In the signature based system it is usually difficult to identify the nature of the mismatch and localize it. One of the typical failure modes of a signature based system occurs when in netlist $\mathcal{N}_1$ there is an element ε with signature σ_ε and in netlist $\mathcal{N}_2$ no element with that signature exists. The mismatch is usually located around ε. However due to the mismatch, the elements in $\mathcal{N}_1$ in a *neighborhood* of ε may not have the same signatures as their corresponding elements in $\mathcal{N}_2$. It is therefore difficult to locate the elements corresponding to the *neighborhood* of ε in $\mathcal{N}_2$ making the mismatch hard to identify.

8.2.2 Incremental Comparison

A precondition for being able to perform the schematic vs. layout comparison incrementally is to have an incremental node extractor for the layout and netlist generator for the schematic. However, even when such extractors are available there are reasons intrinsic to the signature and rule based pattern matching comparison techniques that make incremental comparison difficult.

In effective signature based comparison techniques the signature of an element depends not only on its connected *neighborhood* but also on *far away* elements. The *successive refinement* signature calculation technique described in section 8.1.1 is a commonly used means of accomplishing this. This is one of the strengths of signature based techniques because elements in the two networks which do not correspond to each other but have similar types and neighborhoods will have different signatures and hence can be differentiated. However because of the non local nature of the signature calculation, if a region of a netlist is incomplete the signatures of many circuit elements even *far away* from that region can be affected making this scheme impractical to use on incomplete circuits.

In the rule based pattern matching technique described in section 8.1.3 **all** the rules are needed before reduction (verification) of the layout netlist can begin. This is because the system might have to resolve conflicts between several potentially applicable rules before choosing one to be fired as described in section 8.1.3 . To avoid firing the wrong rule, the system must be able to recognize when conflicts occur and therefore must be aware of all the rules and hence can start verification only when the complete schematic is provided.

8.2.3 Equivalence Flexibility

Unrestricted functional equivalence verification between the schematic and layout representations is impractical as mentioned in section 8.1. However requiring that the two netlists match exactly is too restrictive especially in the presence of circuit elements with interchangeable pins. A certain amount of tolerance is required in the matching process whereby small variations between the netlists, due to geometric optimizations in the layout which are best bound during layout generation, do not cause the matching process to fail. Existing systems try to provide whatever amount of flexibility can be achieved at reasonable cost within the framework of the basic isomorphism algorithms. They are however usually limited to permutations of pins or elements within a logic gate.

In commonly used circuit elements such as NAND and NOR gates or MOS transistors, some of the terminal pins are permutable (input pins for the NAND and NOR gates, drain and source pins for the MOS transistors). The functionality of the circuit is unchanged if the connections to these pins are permuted. Most comparison systems allow for such permutable pins. The user however must indicate which elements have permutable pins and the netlists of both the schematic and layout must be provided to the system in terms of these components with permutable pins.

For example, since NAND and NOR gates have permutable input pins both netlists must be expressed in terms of these gates and not their transistor equivalents. Since the netlist obtained by node extraction from the layout is necessarily expressed in terms of transistors and not logic gates, this netlist must first be *parsed* to reduce the gate netlists into gate modules before the comparison is performed.

Some systems [38] allow for additional freedom in the sequence in which series transistor blocks may be assembled within a gate and allow transistors in parallel (used for greater drive capability) to be treated as one element. Finally, rule-based expert systems [46] have been used to verify the functional equivalence between portions of the netlists that do not match using the signature based method.

8.3 Benefits of Template based Correspondence Verification

In chapter 9 a layout vs. schematic verification method with significant advantages over conventional techniques is described. The method operates directly on the schematic and layout entities and hence requires no node extraction. The schematic and layout representations input to the system for correspondence verification are those defined in

chapters 3 and 6. User defined *correspondence templates* which are a combination of the grammatical productions of chapter 3 and the layout templates of chapter 6 are used to define an equivalence relationship between schematic and layouts.

8.3.1 Benefits of Operating Directly on the Schematic and Layout Domains

Several benefits accrue from having the method operate directly on the schematic and layout entities. Node and netlist extraction are not needed. This saves computation time and eliminates the need for the intermediate netlist data format. Incremental verification which otherwise requires the use of an incremental node extractor is also facilitated.

During comparison using template-based techniques the number of objects accessed in both the schematic and layout domains is considerably less than in their corresponding netlists thus resulting in shorter verification times.

Errors are reported back to the user in terms of mismatches in the connectivity of the schematic modules or layout cells. Such reports are more meaningful and succinct than error reports of mismatches between the netlists.

Schematic vs. layout verification programs have trouble dealing with multiple correspondence possibilities. For example, attempting to match the netlists of two three-inverter ring oscillator circuits of figure 2-6 is difficult because an inverter in the first circuit can map to any of the three inverters in the second circuit. The same problem occurs when there are parallel computation paths in both circuits (in ALUs or bus like structures) because a computation path in one circuit can correspond to one of several such paths in the second circuit.

The user is frequently in a position to guide the correspondence verification system by specifying which inverters (in the ring inverter circuit) or parallel paths (in the ALU) correspond to each other. Describing the correspondence in terms of parings between elements in the two netlists is difficult especially when the two netlists are (functionally) equivalent but not isomorphic. Because feedback from the user is difficult methods which operate on the netlist representation must be robust enough to cope with *ambiguous* information without *any* user input. Operating directly on the schematic and layout representations facilitates user input and hence provides the user with a tighter control over the system.

8.3.2 Benefits of User Defined Equivalences

Since full functional equivalence verification is not practical, correspondence verification systems implement a few useful *equivalences* (such as pin permutations) which are by comparison computationally efficient to implement. These equivalences are usually insufficient to effectively deal with many of the equivalences (for optimization purposes) used by the layout designer. Knowledge about the equivalences are embedded into the system and as such adding new equivalences is difficult. Also many of the equivalences are easy to define on a *case by case* basis but are hard to generalize.

This thesis is primarily concerned with layouts built from library cells and as such only those specific equivalences used in the library cells are of particular interest. This not only restricts the types of equivalences considered but also restricts the locations in the netlist at which they can occur. The equivalences are allowed to occur only at the locations in which they occur in the library cells. The same sort of equivalence will be rejected if it appears elsewhere. This reduces the number of possible configurations that need to be examined.

Equivalences are usually local in nature since global optimizations to the circuit are best done at the schematic or more abstract levels of representation and not during schematic to layout conversion. They can therefore be effectively captured by the correspondence templates introduced in this section.

9

Template based Correspondence Verification

9.1 Overview

In this chapter correspondence templates are introduced. These templates consist of pairs of networks and connectivity graphs in *correspondence* with one another. The templates define an equivalence relation between schematics and layouts. First the formalisms of the templates are introduced and the underlying meaning of the formalisms is described in section 9.2. Conditions in terms of the templates which can guarantee that the netlists of the schematic and layout are isomorphic are then derived in section 9.3. Formal proofs of these conditions are also provided in section 9.3. Finally, techniques for matching schematics and layouts with non-isomorphic netlists are introduced in section 9.4.1.

Throughout this chapter it is assumed that all connectivity graphs are normalized as described in section 6.5.2 and that they satisfy the layout correctness criteria of section 6.5.4. This ensures that the layout is design rule correct and that any two vertices whose layouts interact electrically have an interface edge between them.

9.2 Correspondence Templates

Correspondence templates are used to define equivalence relations between schematics and layouts. The most useful of these equivalences are those in which the schematics and layouts have isomorphic or *very similar* netlists. The discussion is therefore biased toward showing how templates are used to define an equivalence relation between schematics

and layouts with isomorphic netlists. Later in section 9.4.1 useful equivalences between schematics and layouts with non-isomorphic netlists are described.

In chapters 3 and 6 modules and cell instances were introduced. Each module or cell instance has an associated netlist which can be thought of as a circuit[1] in which all modules are of a primitive module type (such as n-channel and p-channel MOS transistors) i.e. they cannot be expressed in terms of other circuits. For ease of explanation these primitive modules will be referred to as *transistor modules.*

9.2.1 Mappings

For a module type $\mathcal{T}_{mod}$, there is often a corresponding celltype $\mathcal{T}_{cell}$ which implements the *same* circuitry. This means that the netlists of any module A_{mod} of type $\mathcal{T}_{mod}$ and cell A_{cell} of type $\mathcal{T}_{cell}$ are isomorphic.

There may be several different isomorphisms possible between the netlists of A_{mod} and A_{cell} and it will be necessary to specify which one of them is being referred to. Let the transistor modules of the netlists of A_{mod} and A_{cell} be labeled $m_0, m_1 \cdots m_n$ and $c_0, c_1 \cdots c_n$ respectively. For a given pairing $P = ((m_i, c_j), (m_k, c_l) \cdots (m_u, c_v))$ between the modules of A_{mod} and A_{cell}, there is at most one network isomorphism (with corresponding module isomorphism $f_{\mathcal{M}}$) such that for any module m_α in the netlist of A_{mod}, $(m_\alpha, f_{\mathcal{M}}(m_\alpha)) \in P$. This pairing of modules, referred to as a mapping between the module and the instance, completely defines the isomorphism (if it exists) and can be used in lieu of a specification of the isomorphism. $M \alpha V$ is used to denote that module M and vertex V are in relation via mapping α.

Definition 17 *A* **mapping** *between a module and an instance is a pairing of transistors in the two corresponding netlists.*

Figures 9-1 (a) and (b) show the *netlist-wise* equivalent module and connectivity graph vertex instance of A_{mod} and A_{cell}. The pairing of transistors representing the isomorphism between the two netlists is shown by the arc labeled P.

Alternately, a schematic such as that of figure 9-2 (a) may have a netlist isomorphic to that of the layout (represented by its connectivity graph) of figure 9-2 (b). In order to characterize the isomorphism between the two netlists, mappings are used to describe the pairings of transistors in the two netlists. A mapping is provided for each module and instance vertex whose netlists have transistors in correspondence via the

[1] As defined in section 3.1.1.

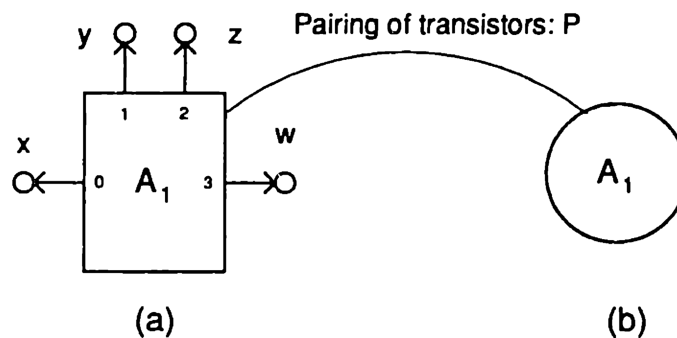


FIGURE 9-1: Mapped Module and Vertex

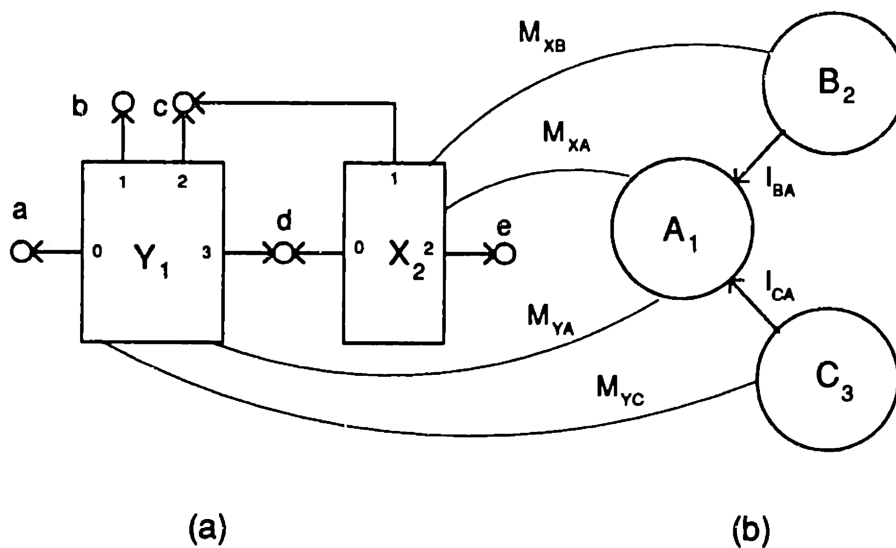


FIGURE 9-2: Mapped Schematic and Layout Regions

isomorphism. For example, if transistors x_3, x_5 and x_7 of module X correspond to transistors a_{24}, a_9 and a_3 of vertex A then the mapping M_{xa} with associated pairing $P_{xa} = ((x_3, a_{24}), (x_5, a_9), (x_7, a_3))$ is defined between module X and vertex A. The mappings between the modules and vertices are shown by labeled arcs in figures 9-2 (a) and (b).

9.2.2 Regions of Equivalence

Given a schematic S and a layout connectivity graph L which are netlist-wise *equivalent* and I the isomorphism between their netlists, it is often possible to identify networks and subgraphs S_i and L_i such that the netlists of each pair (S_j, L_j) are isomorphic via the isomorphism induced by the restriction of I to S_j and L_j. E_{SL}, the set of such pairs (S_i, L_i), is called the set of equivalence regions.

Definition 18 *Let* $(S_j, L_j) \in E$. *If* $\forall S_k \subset S_i, L_k \subset L_i$ *(with* $(S_i, L_i) \neq (S_k, L_k)$*), and* $(S_k, L_k) \notin E_{SL}$ *then* (S_j, L_j) *is called a* **minimal region of equivalence.**

For any $(S_i, L_i) \in E_{SL}$, all the mappings involving modules S_i are necessarily to vertices in L_i and vice versa. Therefore the minimal regions of equivalence are the smallest regions (S_α, L_α) such that all the mappings of modules of S_α are in L_α and vice versa.

9.2.3 Correspondence Templates Definition

Correspondence templates are used to specify how two or more minimal regions of equivalence can be combined to yield a new region of equivalence. Through the use of correspondence templates the isomorphisms between each of the regions of correspondence are extended to define isomorphisms between large portions of the schematic and connectivity graph and finally to the entire schematic and connectivity graph. A correspondence template consists of a schematic and layout (each containing one[2] or more minimal regions of equivalence) whose netlists are isomorphic and a set of mappings between modules and vertices which identifies the particular isomorphism involved.

Definition 19 *A* **correspondence template** *is a triplet* $T = (S, L, M)$ *where* S *is a schematic,* L *is a layout whose netlist is isomorphic to that of* S *via isomorphism* I *(with associated module bijection* $f_{\mathcal{M}}$*) and* M *is a set of mappings between modules*

[2] In order to be useful, templates should contain more than 1 minimal region of equivalence.

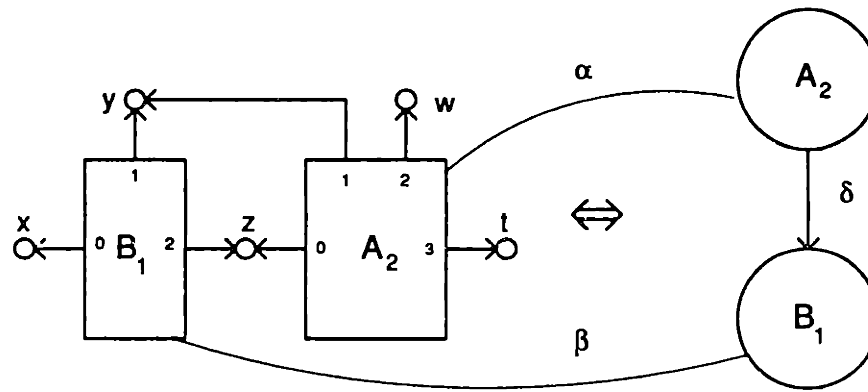


FIGURE 9-3: Simple Correspondence Template

of S and vertices of L such that: for transistors α and β in the netlists of S and L: $\beta = f_{\mathcal{M}}(\alpha) \Leftrightarrow \exists m_i \in M$ *with pairing of transistors* p_i *such that* $(\alpha, \beta) \in p_i$.

Figure 9-3 shows an example of a simple template consisting of two modules and two vertices. Vertex A (respectively B) is the layout for module A (respectively B) and the isomorphism between their netlists is identified by mapping α (respectively β) shown by the thin arc. This template asserts that if instance vertices A and B are relatively placed via interface δ then the electrical connections formed by the interactions of their layouts is equivalent to modules A and B connected as shown in figure 9-3.

Figure 9-4 shows a more complex template. The template consists of two minimal regions of equivalence formed by mappings α, β, δ (modules X, Y and vertices A, B) and ϵ, μ, ν (modules U, W and vertices C, D). The template asserts that if the vertices are placed as prescribed by the interfaces then the electrical connections between the layouts are equivalent to those shown in figure 9-4 between the modules A, B, C and D.

9.2.4 Correspondence Template Occurrence

Correspondence templates are used to *match* small regions of the schematic and layout to be compared. Verification of the correspondence between a schematic and a layout is achieved by *tiling*[3] the schematic and layout with *occurrences* of correspondence templates as prescribed in the next few sections. For ease of explanation it is assumed that mappings already exist between modules and vertices of the schematic and layout to be verified. The goal of the verification system is to verify whether, given this set of mappings, the schematic and layout netlists correspond to each other. Sections 10.1.3 and 10.1.4 describe how the verification algorithm computes these mappings while simultaneously finding *template occurrences.*

[3]The term *tiling* used here has nothing to do with *tiling by abutment.*

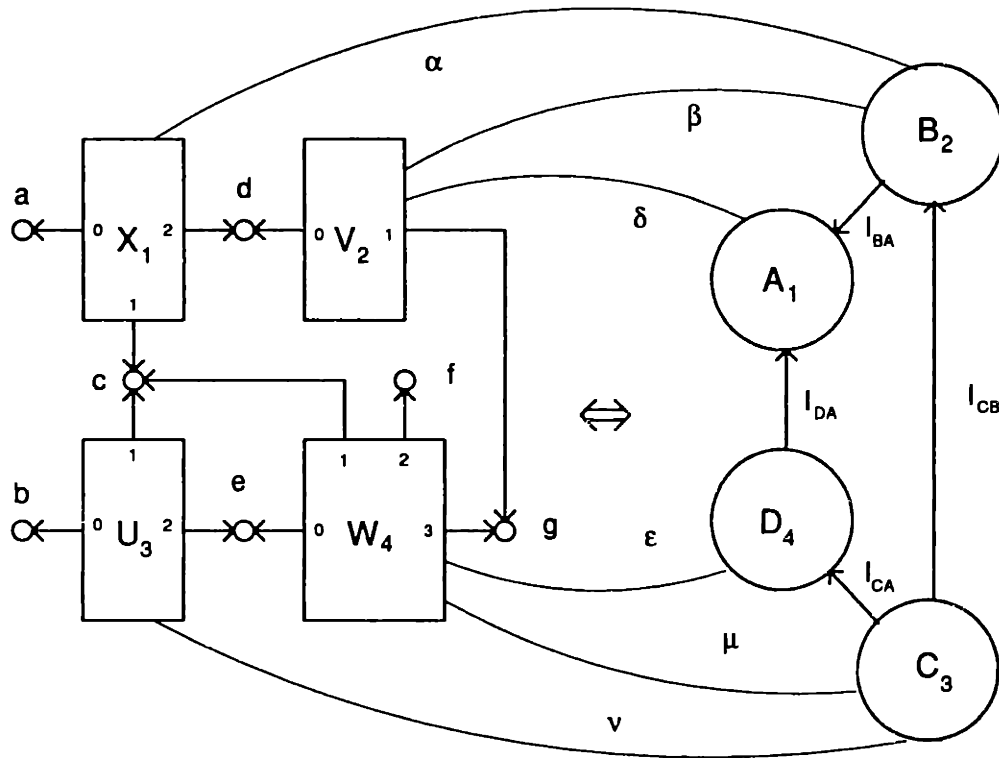


FIGURE 9-4: Correspondence Template

A new relation between networks in the schematic called *semi-isomorphism* is given in definition 20. This relation is used in definition 21 which defines template occurrences. The *semi-isomorphism* relation is quasi identical to the isomorphism relation between networks except that the function f_N which maps the nets of the two networks is surjective instead of a one to one mapping. Intuitively if N_1 is isomorphic to N_2 and if N_3 is obtained by *merging* some of the nets in N_2 together, N_1 is semi-isomorphic to N_3.

Definition 20 *Two networks* $N_1 = (\mathcal{M}_1, \mathcal{N}_1, \mathcal{C}_1)$ *and* $N_2 = (\mathcal{M}_2, \mathcal{N}_2, \mathcal{C}_2)$ *are* **semi-isomorphic** *if and only if there exist two one to one mappings* $f_M : \mathcal{M}_1 \rightarrow \mathcal{M}_2$, *and* $f_C : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ *and a surjection* $f_N : \mathcal{N}_1 \rightarrow \mathcal{N}_2$ *such that:* $\forall M \in \mathcal{M}_1$, $typeof(M) = typeof(f_M(M))$, *and* $\forall(p_1, \eta_1) \in \mathcal{C}_1$ *if* $(p_2, \eta_2) = f_C(p_1, \eta_1)$ *then* $\eta_2 = f_N(\eta_1)$, $module(p_2) = f_M(module(p_1))$ *and* $pinnumber(p_1) = pinnumber(p_2)$.

Definition 21 *Given a schematic* S, *a connectivity graph* L *and a set of mappings* M *between modules in* S *and vertices in* L. *Let* S_{sub} *be a network of* S *and* L_{sub} *a subgraph of* L. (S_{sub}, L_{sub}) *is said to be an* **occurrence** *of correspondence template* $T = (S_T, L_T, M_T)$ *if and only if:* S_T *is semi-isomorphic to* S_{sub} *(with module bijection* $f_{\mathcal{M}}$*),* L_T *is isomorphic to* L_{sub} *(with vertex bijection* $f_{\mathcal{V}}$*) and for any module* M *in* S_T, *vertex* V *in* L_T *and mapping* α *in* M_T, $M \alpha V \Leftrightarrow f_{\mathcal{M}}(M) \alpha f_{\mathcal{V}}(V)$.

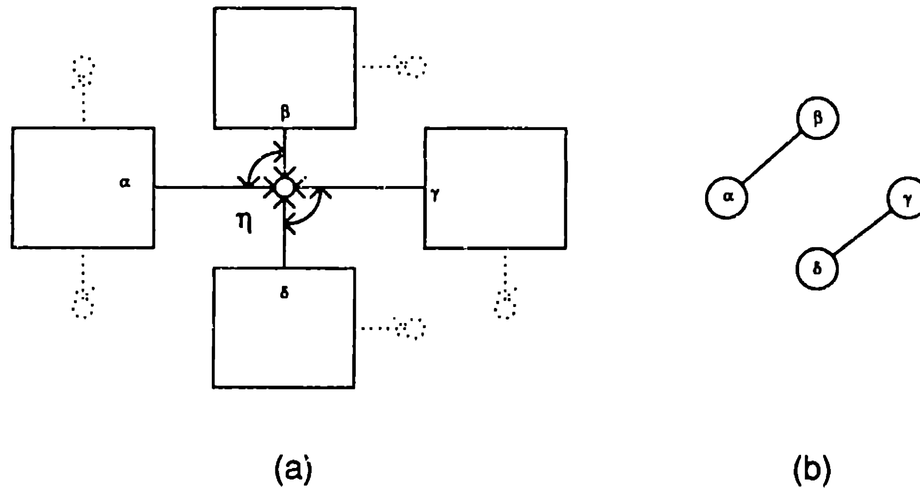


FIGURE 9-5: Net Connection Graph

9.2.5 Net Connection Graph

A useful data structure for keeping track of correspondence template occurrences is the *net connection graph.* For each net η in the schematic there is an associated net connection graph. The vertices of this graph are the pins connected to η. The edges in the graph represent connections between pins which are *known* to connect to the same net. These pins are identified through the use of template occurrences as described in definition 22. If the connection graph of η is (minimally) connected then all these pins must connect to the same net, namely η. It will be shown in section 9.3 that a necessary condition for a schematic and a layout to be in correspondence is that the connection graph for each net be connected.

Definition 22 *The net connection graph for net η is a graph $G_\eta = (P_\eta, O_\eta)$. The set of vertices P_η is the set of pins connected to η. The set of edges $O_\eta \subseteq P_\eta \times P_\eta$ is such that $(p_i, p_j) \in O_\eta \Leftrightarrow$ there is an occurrence of a template $\mathcal{T}$ in which p_i and p_j appear simultaneously and in which their corresponding pins in $\mathcal{T}$ $p_i^{\mathcal{T}}$ and $p_j^{\mathcal{T}}$ are connected to the same net.*

Figure 9-5 (b) shows the net connection graph for net η in figure 9-5 (a). Pins α and β appear simultaneously in a template occurrence of a template $\mathcal{T}$ in which the pins $\alpha_{\mathcal{T}}$ and $\beta_{\mathcal{T}}$ in $\mathcal{T}$ corresponding to α and β are connected to the same net. This is symbolized by an arc between the connections of those pins to net η in figure 9-5 (a). Because of this occurrence, the net connection graph of η in figure 9-5 (b) has an edge between vertices α and β. Similarly, because of an occurrence in which pins δ and γ appear simultaneously, there is an edge between vertices δ and γ in figure 9-5 (b).

9.3 Template based Criteria for Netlist Isomorphism

In this section a criterion based on net connection graphs of schematic S and interfaces of layout L which guarantees that the netlists of S and L are isomorphic is introduced. This criterion described in theorem 8 can be established by identifying the correspondence template occurrences in S and L.

Subsection 9.3.1 gives a definition for netlist and netlist isomorphism and provides the various proofs needed to prove theorem 8 which is finally done in section 9.3.2 proof 8. As before, in order to simplify the proofs it is assumed that the set of mappings between all schematics and layouts are provided. If the reader can believe theorem 8, section 9.3.1 and proof 8 can be safely skipped. In section 9.4.1 the criterion is extended to allow a match between schematics and layouts whose netlists are not necessarily isomorphic.

9.3.1 Preliminaries

Netlist Definition

In order to facilitate the proofs in this section, a definition for netlists different but equivalent to that of circuits in definition 5 is introduced. In this new definition nets do not appear explicitly but are replaced by sets of pins that are electrically connected together. This definition was not suitable for representing schematics introduced in chapter 3.1. In this section however, if the definition for schematics is used for netlists, *nets* in the netlists may have to be *merged* together due to short circuit connections between them. To avoid this the concept of a net is replaced by that of a connected pin set. The set of all connected pin sets is denoted by $\mathcal{E}$.

The reader is reminded that $\mathcal{M}$ is the set of all modules and given a set of modules $M \subseteq \mathcal{M}$, $\mathcal{P}_M$ is the set of all pins in M.

Definition 23 *A* **netlist** $\mathcal{N}$ *is a pair* $(\mathcal{M}_\mathcal{N}, \mathcal{C}_\mathcal{N})$ *where* $\mathcal{M}_\mathcal{N} \subseteq \mathcal{M}$ *and* $\mathcal{C}_\mathcal{N} \subseteq \mathcal{E}$ *such that* $\forall P \in \mathcal{C}_\mathcal{N}, P \subseteq \mathcal{P}_{\mathcal{M}_\mathcal{N}}$ *and* $\forall P_\alpha, P_\beta \in \mathcal{C}_\mathcal{N}, P_\alpha \cap P_\beta = \emptyset$ *and* $\bigcup_{P_i \in \mathcal{C}_\mathcal{N}} P_i = \mathcal{P}_{\mathcal{M}_\mathcal{N}}$.

$\mathcal{M}_\mathcal{N}$ *is called the set of modules of* $\mathcal{N}$ *and* $\mathcal{C}_\mathcal{N}$ *is called the set of connections of* $\mathcal{N}$.

The set of pins $\mathcal{P}_\mathcal{N}$ in a netlist $\mathcal{N} = (\mathcal{M}_\mathcal{N}, \mathcal{C}_\mathcal{N})$ can be thought of as vertices in a graph $G_\mathcal{N}$ such that: for any two pins p_α and p_β in $G_\mathcal{N}$ there is a path between p_α and p_β along edges in $G_\mathcal{N}$ if and only if $\exists P \in \mathcal{C}_\mathcal{N}$ such that $p_\alpha, p_\beta \in P$. $\mathcal{C}_\mathcal{N}$ is then the set of connected components of $G_\mathcal{N}$. $G_\mathcal{N}$ is called a connection graph associated with the connections $\mathcal{C}_\mathcal{N}$ and the netlist $\mathcal{N}$.

For explanatory clarity the modules in a netlist will be referred to as transistors whenever it becomes necessary to distinguish them from modules in the schematic.

Netlist Isomorphism

A definition for isomorphic netlists equivalent to that of isomorphic networks in definition 6 is given in definition 24. If f_M is a mapping between two sets of modules then f_P is used to denote the mapping between their pins and is defined by: $moduleof(f_P(P)) = f_M(moduleof(P))$ and $pinnumber(P) = pinnumber(f_P(P))$.

Definition 24 *Two netlists $\mathcal{N}_1 = (\mathcal{M}_1, \mathcal{C}_1)$ and $\mathcal{N}_2 = (\mathcal{M}_2, \mathcal{C}_2)$ are* **isomorphic** $\mathcal{N}_1 \simeq \mathcal{N}_2$ *if and only if there exists a one to one mapping $f_M : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ such that if f_P is the pin mapping associated with f_M then $\forall p_1, p_2 \in \mathcal{P}_{\mathcal{M}1}$, $\exists P_1 \in \mathcal{C}_1, p_\alpha, p_\beta \in P_1 \Leftrightarrow \exists P_2 \in \mathcal{C}_2, f_P(p_\alpha), f_P(p_\beta) \in P_2$.*

Union of Connections

An operation $\uplus$ similar to the set union operation is defined between elements of $\mathcal{E}$. It can be shown that the $\uplus$ operator is associative and commutative. Let $\mathcal{C}_1, \mathcal{C}_2 \in \mathcal{E}$ and $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ be connection graphs associated with $\mathcal{C}_1, \mathcal{C}_2$.

Definition 25 $\mathcal{C} = \mathcal{C}_1 \uplus \mathcal{C}_2$ *is the set of connected components of graph $G = G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$.*

Inclusion of Connections

A relation $\Subset$ similar to the set inclusion is defined between elements of $\mathcal{E}$. It can be shown that the $\Subset$ relation is transitive. Let $\mathcal{C}_1, \mathcal{C}_2 \in \mathcal{E}$ and $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ be connection graphs associated with $\mathcal{C}_1, \mathcal{C}_2$.

Definition 26 $\mathcal{C}_1 \Subset \mathcal{C}_2$ *if and only if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.*

This relation is used to define the subset relation for netlists in the obvious manner. If $\mathcal{N}_1 = (\mathcal{M}_1, \mathcal{C}_1)$ and $\mathcal{N}_2 = (\mathcal{M}_2, \mathcal{C}_2)$ then $\mathcal{N}_1 \subseteq \mathcal{N}_2 \Leftrightarrow \mathcal{M}_1 \subseteq \mathcal{M}_2$ and $\mathcal{C}_1 \Subset \mathcal{C}_2$.

Union of two Isomorphic Netlists

Let $N_1 = (M_1, C_1)$ be isomorphic to $N_1' = (M_1', C_1')$ with module mapping f_{M1} and $N_2 = (M_2, C_2)$ be isomorphic to $N_2' = (M_2', C_2')$ with module mapping f_{M2} such that $\forall M \in M_1 \cap M_2$, $f_{M1}(M) = f_{M2}(M)$. Let $f_M(M) = f_{M1}(M)$ for $M \in M_1$ and

$f_M(M) = f_{M2}(M)$ otherwise. Let G_1, G_1', G_2 and G_2' be connection graphs associated with N_1, N_1', N_2 and N_2'.

Theorem 7 *$N = (M_1 \cup M_2, C_1 \uplus C_2)$ is isomorphic to $N' = (M_1' \cup M_2', C_1' \uplus C_2')$ via module isomorphism f_M.*

Proof 7 *Clearly f_M is a one to one mapping from $M_1 \cup M_2$ to $M_1' \cup M_2'$. Let f_P be the associated pin mappings and G, G' be connection graphs associated with N and N'. If $\exists P_i \in C_1 \uplus C_2$ with $p_\alpha, p_\beta \in P_i$ then p_α and p_β are in the same connected component of G and therefore there is a path in G from p_α to p_β. This path can be decomposed into a path $(p_\alpha, p_0 \cdots p_n, p_\beta)$ of vertices in G_M such that each subpath (p_i, p_{i+1}), (p_α, p_0) and (p_n, p_β) is entirely in G_1 or G_2.*

For each such subpath (p_x, p_y), $(f_P(p_x), f_P(p_y))$ is a path either entirely in G_1' or in G_2'. Hence $(f_P(p_\alpha), f_P(p_0) \cdots f_P(p_n), f_P(p_\beta))$ is a path in G' and therefore $f_P(p_\alpha)$ and $f_P(p_\beta)$ are in the same connected component of G' and $\exists P_i' \in C_1' \uplus C_2', f_P(p_\alpha), f_P(p_\beta) \in P_i'$. This proves $\exists P_i \in C_1 \uplus C_2,\ p_\alpha, p_\beta \in P_i \Rightarrow \exists P_i' \in C_1' \uplus C_2', f_P(p_\alpha), f_P(p_\beta) \in P_i'$.

Similarly, since f_P is a one to one mapping it can be shown that $\exists P_j' \in C_1' \uplus C_2', f_P(p_\alpha), f_P(p_\beta) \in P_j' \Rightarrow \exists P_j \in C_1 \uplus C_2, p_\alpha, p_\beta \in P_j$. Hence N and N' are isomorphic.

Netlist of a Schematic or Layout

Let $\mathcal{S} = (M_S, N_S, C_S)$ be a schematic where each module $M_i \in M_S$ ($\bigcup_{i \in I} M_i = M_S$) has associated netlist $N_i = (\mathcal{M}_i, \mathcal{C}_i)$. The netlist $\mathcal{N}_S$ of $\mathcal{S}$ consists of the netlists of each of the modules M_i plus the connections (between the pins of transistors in the netlist) generated by the connections of modules in M_S to the nets in N_S. Each net η in N_S electrically connects the pins of some of the transistors (in netlists) of the modules in the schematic. The set of connections in $\mathcal{N}_S$ created at net η is denoted by $\mathcal{C}_\eta$. The netlist $\mathcal{N}_S$ then consists of the transistors in each of the modules in the schematic and the connections of transistors within each of the module netlists plus the connections generated by the connection of modules in M_S to nets in N_S: $\mathcal{N}_S = (\bigcup_{i \in I} \mathcal{M}_i, (\biguplus_{i \in I} \mathcal{C}_i) \uplus (\biguplus_{\eta \in N_S} \mathcal{C}_\eta))$.

Similarly, let $\mathcal{L} = (V_\mathcal{L}, E_\mathcal{L}, W_\mathcal{L})$ be a connectivity graph where each vertex $V_j \in V_\mathcal{L}$ ($\bigcup_{j \in J} V_j = V_\mathcal{L}$) has associated netlist $N_j = (\mathcal{M}_j, \mathcal{C}_j)$. The netlist $\mathcal{N}_\mathcal{L}$ of $\mathcal{L}$ consists of the netlists of each of the vertices V_j plus the connections generated by the interfaces in $E_\mathcal{L}$. The set of connections in the netlist created by interface τ is denoted by $\mathcal{C}_\tau$. The netlist $\mathcal{N}_\mathcal{L}$ then consists of the transistors in each of the vertices in the layout and

the connections of transistors within each of these vertex netlists plus the connection generated by the interfaces in $E_{\mathcal{L}}$: $\mathcal{N}_{\mathcal{L}} = (\bigcup_{j\in J} \mathcal{M}_j, (\biguplus_{j\in J} \mathcal{C}_j) \uplus (\biguplus_{\tau\in E_{\mathcal{L}}} \mathcal{C}_\tau))$.

Netlists of an Occurrence

Let $\mathcal{T}_i$ be an occurrence of $\mathcal{T}$. Let S_i and L_i be the schematic and layout portions of $\mathcal{T}_i$ respectively. The netlists $\mathcal{N}_{S_i} = (\mathcal{M}_{S_i}, \mathcal{C}_{S_i})$ and $\mathcal{N}_{L_i} = (\mathcal{M}_{L_i}, \mathcal{C}_{L_i})$ associated with S_i and L_i respectively consist of the netlists of each of the components in the occurrence plus the connections established in the occurrence which are the connections between the components necessary for $\mathcal{T}_i$ to be recognized as an occurrence of $\mathcal{T}$. Since the layout and schematic netlists of $\mathcal{T}$ are isomorphic it can be shown that $\mathcal{N}_{S_i}$ and $\mathcal{N}_{L_i}$ are necessarily isomorphic.

9.3.2 Netlist Isomorphism Criteria

Intuitively, if the net connection graph of each net η in the schematic is connected, all the pins connected to η are accounted for. Similarly, if each interface ι appears in a template occurrence, the electrical connections caused by ι are accounted for. Since the mappings between the schematic and the layout are given and each component is assumed to have a complete set of mappings, there is already a one to one relation between the transistors in both netlists. Since all the connections in both netlists are *accounted for* the two netlists must be isomorphic. A formal definition of this criterion and a formal proof are given in theorem and proof 8 respectively.

Theorem 8 *Let $\mathcal{S} = (M_S, N_S^0, C_S)$ be a schematic where each module $M_i \in M_S$ ($\bigcup_{i\in I} M_i = M_S$) has associated netlist $N_i = (\mathcal{M}_i, \mathcal{C}_i)$. Let $\mathcal{L} = (V_{\mathcal{L}}, E_{\mathcal{L}}, W_{\mathcal{L}})$ be a connectivity graph where each vertex $V_j \in V_{\mathcal{L}}$ ($\bigcup_{j\in J} V_j = V_{\mathcal{L}}$) has associated netlist $N_j = (\mathcal{M}_j, \mathcal{C}_j)$.*

If all the net connection graphs in $\mathcal{S}$ are connected and all the interfaces in $\mathcal{L}$ appear in a template occurrence. The netlists $\mathcal{N}_S$ and $\mathcal{N}_{\mathcal{L}}$ of $\mathcal{S}$ and $\mathcal{L}$ are then isomorphic to each other.

Proof 8 *Let $N_{S_\kappa} = (M_{S_\kappa}, C_{S_\kappa})$ and $N_{L_\kappa} = (M_{L_\kappa}, C_{L_\kappa})$ be the netlists associated with the schematic and layout components of occurrence O_κ and let $\mathcal{K}$ be the set of indices such that O_κ is an occurrence. The schematic and layout netlists associated with all the occurrences are then $N_S = (M_S, C_S) = (\bigcup_{\kappa\in\mathcal{K}} M_{S_\kappa}, \biguplus_{\kappa\in\mathcal{K}} C_{S_\kappa})$ and $N_L = (M_L, C_L) = (\bigcup_{\kappa\in\mathcal{K}} M_{L_\kappa}, \biguplus_{\kappa\in\mathcal{K}} C_{L_\kappa})$. Since $\forall\kappa \in \mathcal{K}$, $N_{S_\kappa} \simeq N_{L_\kappa}$ from theorem 7 it is necessarily true that $N_S \simeq N_L$.*

Clearly the netlists N_{S_κ} and N_{L_κ} of each template occurrence O_κ are subsets of the netlists $\mathcal{N}_S$ and $\mathcal{N}_\mathcal{L}$ of the schematic and layout. Hence $N_S \subseteq \mathcal{N}_S$ and $N_L \subseteq \mathcal{N}_\mathcal{L}$.

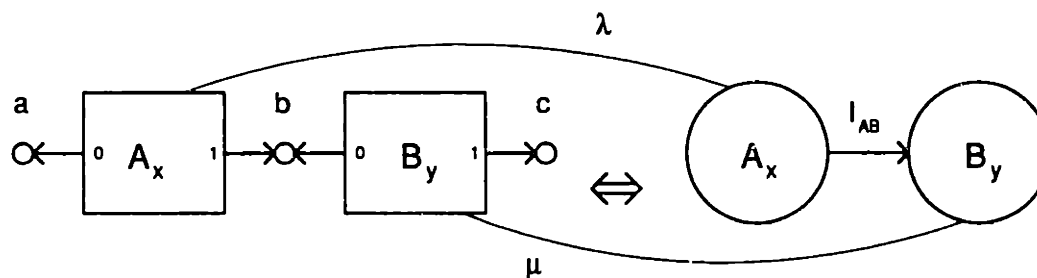
If each interface appears in a template occurrence then for any interface τ in the layout the connections associated with τ is $\mathcal{C}_\tau \underline{\Subset} \biguplus_{\kappa \in \mathcal{K}} C_{L_\kappa} = C_L$. Hence $(\biguplus_{\tau \in E_\mathcal{L}} \mathcal{C}_\tau) \underline{\Subset} C_L$. If all interfaces appear in an occurrence then all vertices in the layout must also appear in an occurrence and hence the transistors and connections in the netlist of those vertices must appear in the layout netlist of an occurrence. Hence the transistors in the layout netlist $\bigcup_{j \in J} \mathcal{M}_j$ and the connections of those transistors in each of the netlists $\biguplus_{j \in J} \mathcal{C}_j$ must satisfy $(\bigcup_{j \in J} \mathcal{M}_j) \subseteq M_L$ and $(\biguplus_{j \in J} \mathcal{C}_j) \underline{\Subset} C_L$. Hence $(\biguplus_{j \in J} \mathcal{C}_j) \uplus (\biguplus_{\tau \in E_\mathcal{L}} \mathcal{C}_\tau) \underline{\Subset} C_L$. Therefore $\mathcal{N}_\mathcal{L} \subseteq N_L$ and since $N_L \subseteq \mathcal{N}_\mathcal{L}$ it must be that $\mathcal{N}_\mathcal{L} = N_L$.

If the net connection graph for a net η is connected it can be shown by induction on the number of pins connected to η that $\mathcal{C}_\eta \underline{\Subset} \biguplus_{\kappa \in \mathcal{K}} C_{S_i} = C_S$. This is because in the graph of C_S there is a path between all the pins associated with net η and hence all such pins appear in one connected component. The sole element of $\mathcal{C}_\eta$ (the connection graph of $\mathcal{C}_\eta$ has one connected component) is necessarily a subset of this connected component. Therefore if N_S^0 is the set of nets in the schematic, $\biguplus_{\eta \in N_S^0} \mathcal{C}_\eta \underline{\Subset} C_S$. With this result by the same reasoning as in the previous paragraph applied to the modules and connections in $\mathcal{N}_S$, it can be shown that $\mathcal{N}_S \subseteq N_S$ and since $N_S \subseteq \mathcal{N}_S$ it must be that $\mathcal{N}_S = N_S$.

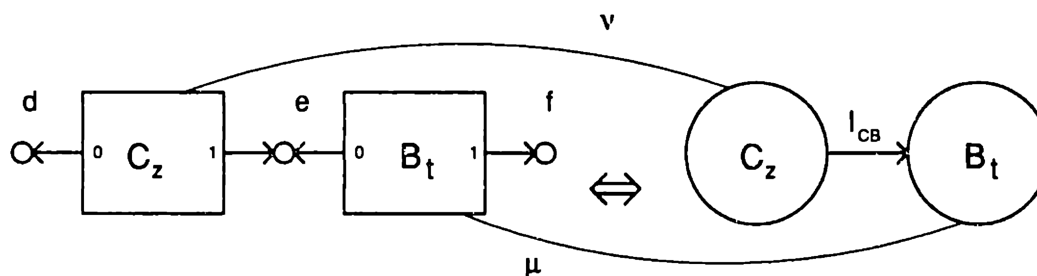
Therefore $\mathcal{N}_\mathcal{L} = N_L \simeq N_S = \mathcal{N}_S$ and so $\mathcal{N}_S$ and N_L are isomorphic.

Example

Figures 9-6 (a) and (b) show two correspondence templates T_1 and T_2. In this example, theorem 8 and templates T_1 and T_2 are used to prove the equivalence of the layout and schematic of figure 9-7 (a). The pins connected to net η in figure 9-7 (a) are given unique identifiers α, β and γ which are indicated on the connection from the pin to the net. The schematic and layout are in equivalence because all the interfaces in the layout appear in a template occurrence and all the net connection graphs are connected. The net connection graph of net η is shown in figure 9-7 (b). The graph is connected and the origin of its edge is indicated. The other nets (g, h and i) connect to only one pin and appear in at least one template occurrence each and hence are considered as *trivially* connected.



(a) Template T_1



(b) Template T_2

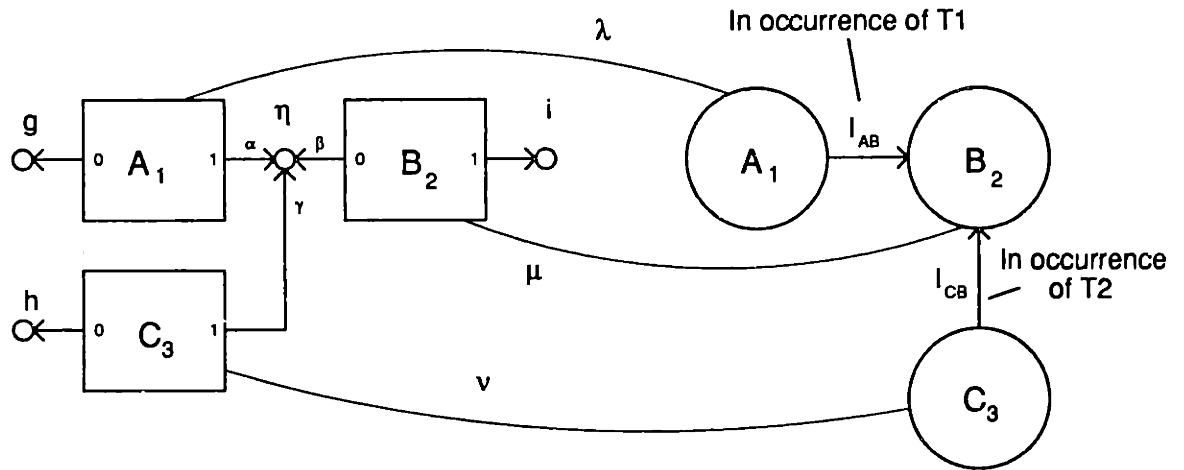
FIGURE 9-6: Equivalence Templates

9.4 Extensions

9.4.1 Equivalence Flexibility

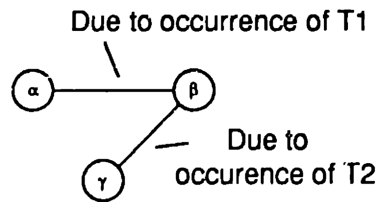
The techniques described in section 9.3 can be extended to allow the matching of schematics and layouts whose netlists are *equivalent* but not necessarily isomorphic. In section 9.3 each layout cell and schematic module type is assumed to have an associated netlist. Each correspondence template asserts that the netlists of the schematic and layout sections of the template are isomorphic, and the associated netlists of these components are implicitly assumed. The verification process attempts to match the netlists of the schematic and the layout given these assumed netlists.

Since the verification system does not actually manipulate the netlists of the schematic or layout, the assumed netlists of the various components and the actual netlists implemented by the components may differ. The results provided by the verification system are those obtained using the assumed netlists. A schematic and layout with non isomorphic netlists will be reported as a match by the verification system if the netlists obtained using the assumed component netlists are isomorphic. By exploiting the potential differences between the actual and assumed netlists, matching of schematics and layouts



(a)

Net Connection Graph of η



(b)

FIGURE 9-7: Equivalence Criterion

whose netlists are not necessarily isomorphic can be achieved.

Each template assumes a given association between a component and its netlist. These associations are indirectly captured by the mappings between the components in each domain. Different templates may assume different associations. For explanatory purposes it will be presumed that the assumed netlist of the schematic component is fixed and is always equal to the actual netlist. If a layout celltype L_c has a single mapping to moduletype M_a in template $\mathcal{T}_a$ and to moduletype M_b in template $\mathcal{T}_b$ then if M_a and M_b have different netlists the assumed netlist of L_c is effectively different in each of the templates. The verification algorithm finds the mappings between components as well as the template occurrences and will hence implicitly choose the appropriate assumed netlists for each (layout) component that make a match possible.

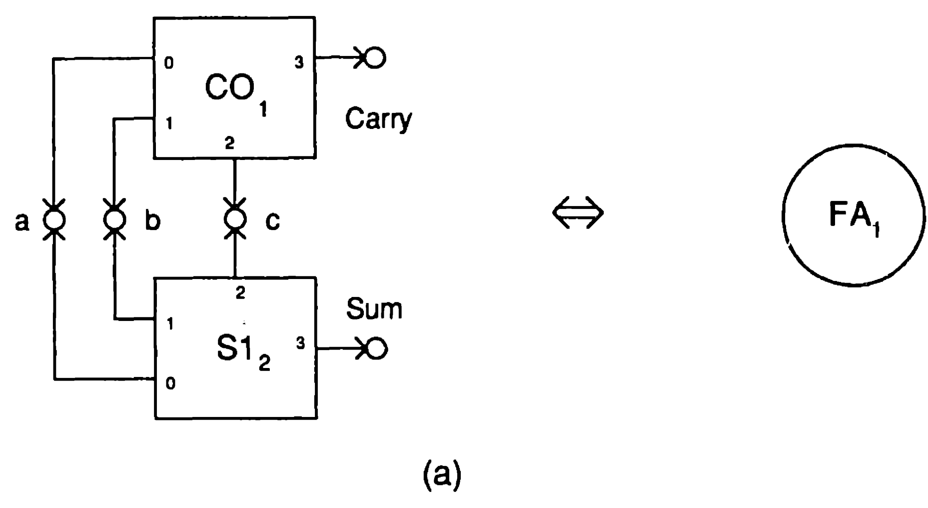
Through the use of mappings and template occurrences, equivalences are implicitly defined between the actual and assumed component netlists. For the templates to be meaningful in each minimal region of equivalence, the assumed and implemented netlists of that region of layout should be functionally equivalent (given the context in which they are used).

For example, in figures 9-8 (a) and (b) if FA is a full adder celltype, CO is a moduletype which computes $Carry = ab \vee ac \vee bc$, $S1$ is a moduletype which computes $Sum = abc \vee a\bar{b}\bar{c} \vee \bar{a}b\bar{c} \vee \bar{a}\bar{b}c$ from the values a, b and c and $S2$ is a moduletype which computes $Sum = (a \vee b \vee c) \cdot \overline{Carry} \vee abc$ from the values a, b, c and $Carry$. The two templates of figures 9-8 (a) and (b) then both represent *reasonable*[4] templates (power, ground and the mappings between modules and vertices are not shown). The assumed netlist of the layout cell FA is different in the two templates.

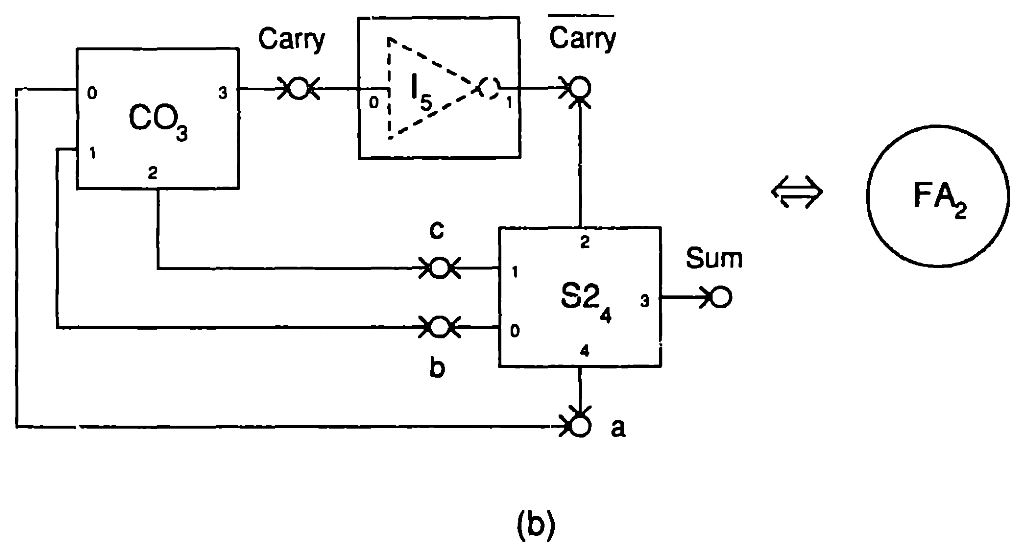
Requirements on Correspondence Templates

The assumed netlist C_1 of the layout component of a minimal region of equivalence R_{min} is typically designed to be connected to the rest of the circuit via certain specific *boundary nets*. An actual netlist C_2 can be used in lieu of C_1 if it has a set of *boundary nets equivalent* to that of C_1. The functionality of the circuit is unchanged if C_2, connected via its boundary nets, is substituted for C_1. However, the functions computed on nets of C_1 not on the boundary may be different from those of C_2. In fact C_1 and C_2 may have a different number of such nets. For example, the netlists of the schematics of figures 9-8 (a) and (b) although *equivalent* probably have a different number of nets and

[4]These templates are for explanatory purposes only. They have only one region of equivalence and hence are not of any practice value.



(a)



(b)

FIGURE 9-8: Full-adder Cell Implementations

hence at least one of them has a different number of nets than that of the implemented netlists of FA.

Care must be taken to ensure that the assumed nets in C_2 as well as the actual nets in C_1 that are not in the boundaries are not connected to the rest of the circuit. Hence nets that appear in the schematic but are not in the boundary of C_1 cannot connect to modules which are not in this region of equivalence. Similarly, the layout vertices of this region of equivalence should not have any interfaces to vertices outside R_{min} that result in connections to nets which are not in the boundary of C_2.

By carefully designing the correspondence templates these requirements can be easily verified. For the set of mappings between the schematic and layout in R_{min}, templates that allow connections of modules not in R_{min} to non-boundary nets in C_1 and templates that allow interfaces to vertices not in R_{min} to access nets which are not in the boundary of C_2 are not permitted. If this requirement is met, *illegal* connections to non-boundary nets in C_1 will cause the net connection graph of those nets to be unconnected because the *illegal* connections to those nets will not be accounted for in any template occurrence. Similarly, *illegal* connections to non-boundary nets in C_2 will result in the interfaces causing those connections not to appear in any template occurrences. Both these situations will cause the schematic vs. layout comparison to fail.

Using GRASP

The network reduction capabilities in GRASP can be used to further enhance the effectiveness of matching schematics and layouts with non-isomorphic netlists. Using a special grammar, the GRASP parser can reduce networks consisting of small modules such as transistors and inverters into larger functional blocks such as a full adder cell or XOR gate. The reduced schematic consisting of these larger functional blocks is then compared with the layout using SCHEMILAR. For example, the networks of figures 9-8 (a) and (b) can both be reduced into the same full adder celltype. The schematic with these networks replaced by full adder cells is then compared with the layout.

There are several advantages of reducing the schematic before comparison with the layout.

1. Design alternatives for a same functionality are better captured by the circuit grammars used in GRASP rather than by having one correspondence template for each possible schematic and layout alternative.

2. After reduction, since the schematic contains fewer modules with more distinctive module types, it is easier for the user to identify and rectify the cause of a

correspondence failure.

3. The design of the correspondence templates requires knowledge of the layout cell library whereas the circuit grammar which is capable of reducing the schematic into larger functional blocks does not. By first reducing the schematic using GRASP, for a given cell library the number of correspondence templates that need to be designed is reduced.

4. Correspondence verification time (in the worst case) is exponential in the number of modules[5] in a correspondence template. By first reducing the number of modules in the schematic, the number of modules in the schematic portion of the correspondence templates is reduced and correspondence verification speed is increased.

9.4.2 Dealing with Bus Instances

In this section the formalisms of section 9.3 are extended to deal with layout *bus* cells which contain no active circuitry and are used to electrically connect different portions of the layout. Because such cells contain no active circuitry, they can have no mappings to schematic modules and hence useful templates in which they may occur cannot be defined. First an explanation of why this problem occurs is provided. Then through the use of *blank* modules the techniques of section 9.3 are extended to deal with layout *bus* cells.

Cause of the Problem

Mappings are the mechanism used in this thesis for defining the correspondence between the schematic and layout components and hence implicitly defining the correspondence between the elements of their netlists. The mappings describe correspondences between transistors in each netlist, correspondences between the nets and the connections between pins and nets are implied. Through the use of mappings, the entire isomorphism between the layout and schematic netlists is defined.

This scheme requires that in both netlists each net is connected to at least one pin. The correspondences between nets not connected to any modules cannot be inferred by the defined correspondences between the modules. For this reason isomorphisms between networks with unconnected nets cannot be defined. The netlists corresponding to layout

[5] Typically there are many modules for each layout vertex, hence the number of modules in a correspondence template can become quite large.

bus vertices consist solely of nets. Hence mappings cannot be used to identify such nets in the netlist.

Although *bus* cells have no active circuitry and hence have no corresponding components in the schematic, they do participate in the electrical properties of the layout. Removing them from the layout changes the electrical characteristics of the layout. Consider the case of figures 9-9 (a), (b) and (c). Figure 9-9 (a) is the schematic representation consisting of two inverters. Figure 9-9 (b) is the layout of figure 9-9 (a) in which the two inverter instances are connected together via m instances of the *bus* cell. Finally figure 9-9 (c) is the connectivity graph representation of the layout of figure 9-9 (b).

In order for the system to recognize that the two layout inverter cells are electrically connected as specified in the schematic, the system must know the nets corresponding to layout bus instances B_1 and B_n. Since the verification system must be able to handle this situation for any value of n, a template whose layout component consists of two bus cells connected together must exist. The problem of defining a schematic corresponding to such a layout and thus enabling templates to be defined is the the topic of the next subsection.

A Solution based on Blank Modules

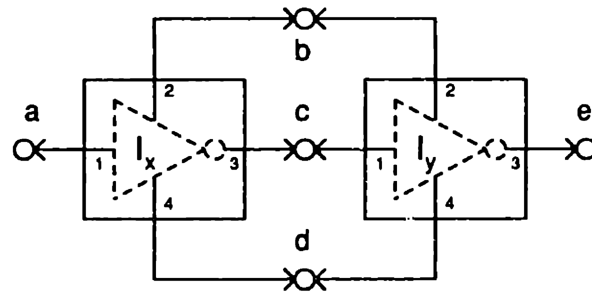
In order to be able to associate a layout *bus* cell instance with its corresponding nets in the schematic a moduletype *blank* with special properties is introduced. *Blank* modules have one pin and their purpose is to identify the net connected to them. There is at most one blank module at each net. Blank modules do not contribute to the functionality of the circuit and therefore if a layout matches a schematic with blank modules it matches the same schematic with the blank modules deleted. Hence blank modules can be inserted into the schematic *at will* to facilitate the verification process. The verification process can then be defined by the sequence of steps described below. Steps 1 and 3 have been introduced to deal with bus instances.

1. Add a blank module to each net.

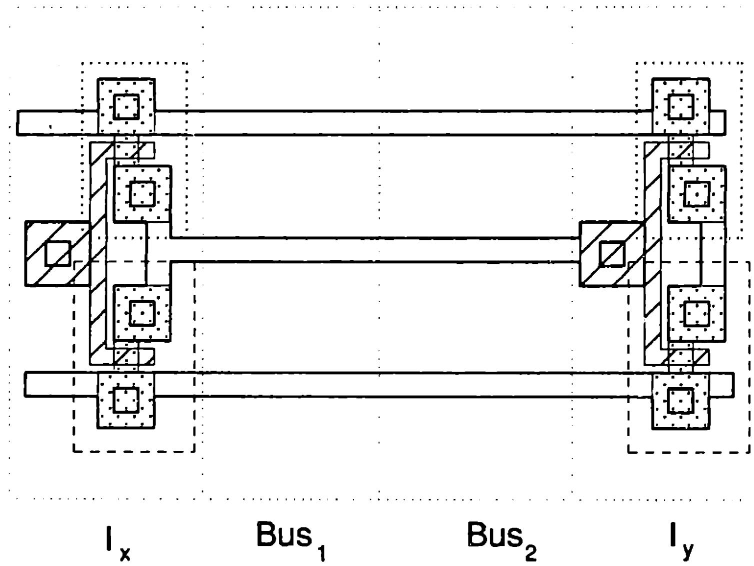
2. Find the template occurrences.

3. Remove all blank modules that are not mapped.

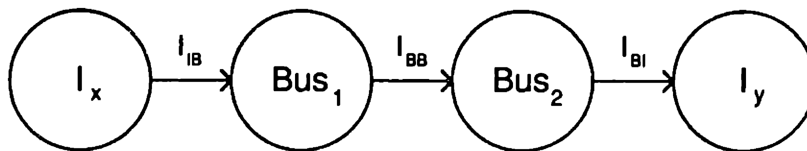
4. Verify that all net graphs are connected and all interfaces appear in a template occurrence.



(a)



(b)



(c)

FIGURE 9-9: Bus Cells

Example

Figures 9-10 (a), (b) and (c) show the templates needed for dealing with the case described in figure 9-9. The nets corresponding to the bus cells can now be identified by the blank module they connect to. For instance, the bus vertex i in figure 9-10 (b) has three corresponding nets t, u and v (corresponding to the bus wires of the bus cells shown in figure 9-9 (b)) which can be identified by their associated blank modules d, e and f.

Special Properties of Blank Modules

Unlike regular modules whose mappings are limited to the vertices in a template occurrence, the number of mappings a blank module can have is unbounded. For example, in figure 9-9 the blank modules corresponding to the nets in the bus are mapped to all m of the bus cells.

Unlike regular templates, the schematic portion of templates which deal with blank modules may not be connected as is demonstrated in figure 9-10 (b). Because of this it is sometimes impractical to design templates which capture all of the blank modules mapped to a bus vertex. This is especially true for bus cells which have busses running in both the horizontal and vertical directions (such as PLA crosspoint cells). In this case it is advantageous to deal with horizontal and vertical connections of bus vertices separately and hence deal with only some of the nets (those nets corresponding to either the horizontal or vertical busses) of the bus cell at a time. Because of the special *functionless* properties of blank modules, it is meaningful to define templates which capture only some of the blank modules associated with a bus instance.

Some layout instances are combinations of busses and active circuitry and these two aspects of the cell can then be dealt with separately.

9.4.3 Dealing with Encoded Layout Cells

Encoded cells described in section 6.5.5 is a technique used to deal more efficiently with families of layout cells having similar layout characteristics. In this method the functionality of one basic cell, called the encoded cell, is altered by superimposing the layout of encoding cells. Encoded cells are frequently used (in fact the experiments described in section 10.3 make heavy use of them) and as such it is important that a correspondence verification system of the type described in this thesis be capable of dealing with them.

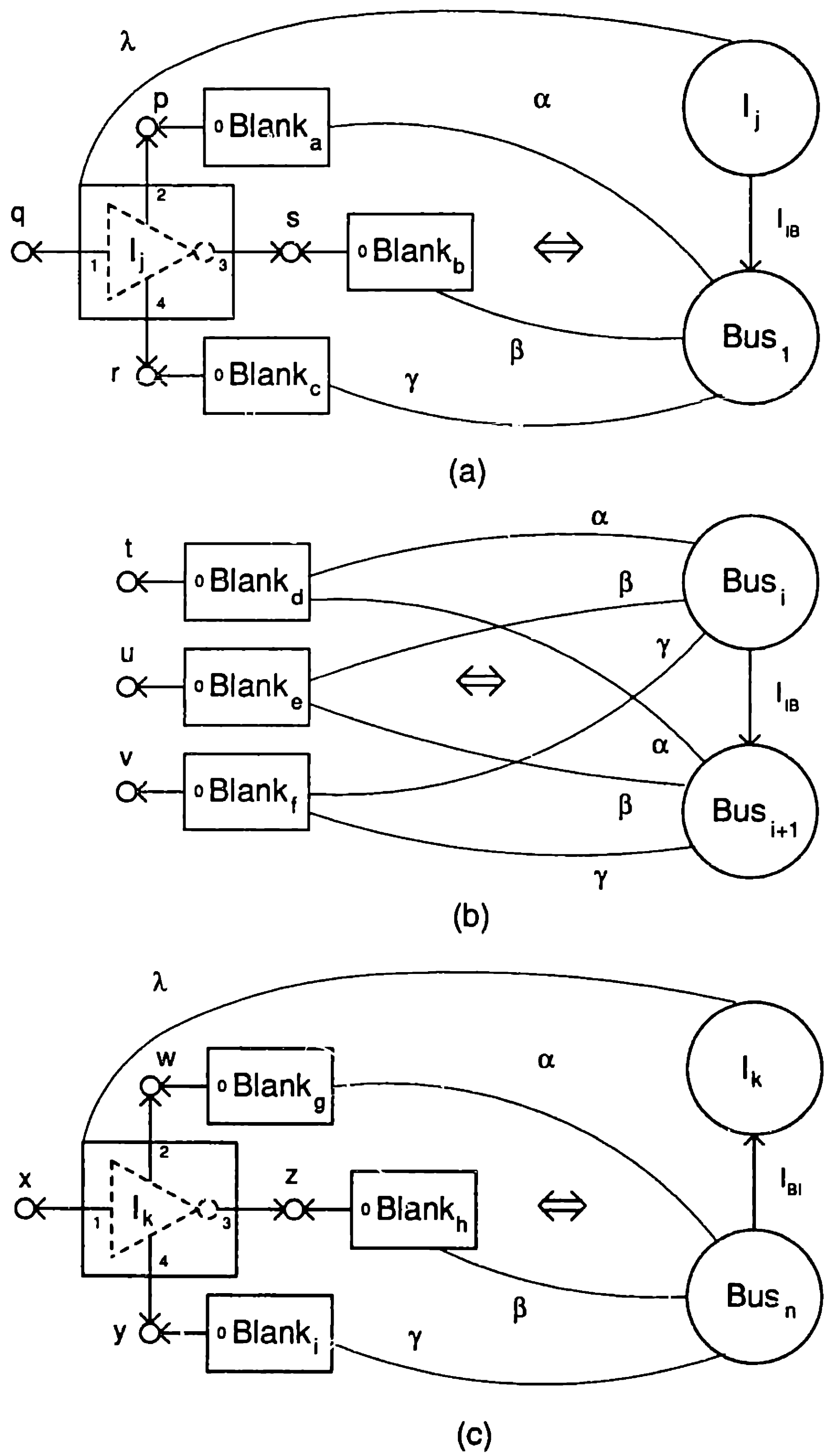


FIGURE 9-10: Correspondence Templates for Bus Cells

For the purposes of correspondence verification, encoded cells are no different from any other cells. Due to the interaction between the layout of the base cell and its encoding cells, the netlists of the base cell and the encoding cells are connected together just as between two *ordinary* cells.

However, certain minimal conditions on the base cell and its encodings are required. Transistors that are not present in either the base cell or its encoding cells cannot be created when the two instances are put together. The encoding cells **a_t** and **o_t** used to encode the AND and OR planes of the PLA in figure 6-2 consist of a single rectangle of diffusion. When this rectangle of diffusion is added to the base cell **and_sq** or **or_sq** a pulldown transistor not present in either the base cell or the encoding cell is formed. By adding to the encoding cells **a_t** and **o_t** the layout necessary to form this transistor, the final layout of the PLA is unchanged and it is now possible to verify the netlist of the schematic using the template based approach.

In practice the layout cells **a_t** and **o_t** need not be modified. By designing every correspondence template in which these cells appear, assuming that the transistor is already formed, the correspondence verification system will perform correctly.

9.4.4 Schematic vs. Schematic Correspondence Verification

The techniques described in this chapter can be adapted to perform schematic vs. schematic correspondence verification. The formalisms developed in this chapter carry through for correspondence templates which have two schematic components instead of a schematic and a layout component. In this case each mapping relates a module in one schematic to a module in the other schematic. The concept of net connection graph applies to both schematics and the criteria for a correspondence match is that the net connection graphs in both schematics are connected.

The same benefits accrue as those associated with the template based schematic vs. layout correspondence method described in sections 8.3.1 and 8.3.2 . First by operating directly on the schematic objects, netlist extraction is not needed. Secondly, user input and incremental verification is facilitated. Finally, user defined equivalences permit much greater flexibility in the correspondence definition process allowing matching between schematics with non-isomorphic netlists as described in section 9.4.1.

10

Correspondence Verification Algorithm & Implementation

10.1 Verification Algorithm

10.1.1 Overview

The template based correspondence verification technique described in section 8.3 has been implemented in a computer program called SCHEMILAR (**Sche**matic vs. **la**yout comparator). SCHEMILAR's algorithm is a combination of parts of GRASP's and GLOVE's algorithms. GRASP's network finding algorithm is coupled with GLOVE's sub-graph finding algorithm to yield an event driven algorithm capable of identifying correspondence template occurrences.

Each event corresponds to a vertex or a module which has recently been given a mapping. If the event corresponds to a vertex (respectively module), the algorithm first identifies the layout (respectively schematic) component of the occurrence. The existing mappings of the vertices (respectively modules) are then used to identify one or more modules (respectively vertices) in the occurrence. Then the schematic (respectively layout) component of the occurrence is found using that module (respectively vertex) as a starting point. Modules and vertices which were previously unmapped are given a mapping and a new event is created for each of these components. After all the events have been processed, if all the net connection subgraphs are connected and all interfaces appear in an occurrence then the schematic and layout are reported to match.

10.1.2 Preliminaries

Events

SCHEMILAR uses an event queue to manage events. Each event is associated with either a module in the schematic or a vertex in the layout. The modules or vertices for which there is an event on the queue are components whose mappings have been defined and for which all the template occurrences in which they appear may not yet have been found. When the event for a component is processed all the occurrences in which that component appears are identified. During this process mappings for other components may be generated. An event for each of these components is then created and placed on the event queue.

It is possible to simplify the algorithm and retain only one kind of event. In this scheme all events are associated with a module in the schematic or all events are associated with a vertex in the layout. In order to facilitate the discussion it will be assumed that all events correspond to vertices in the layout. The penalty associated with this simplification is a reduced flexibility during incremental updates to the schematic. However, with very minor modifications the scheme can be made to accommodate *module events* as well as *vertex events.*

Module and Vertex Position

The concept of a module position in a production RHS defined in section 5.1.2 (respectively vertex position in a layout template defined in section 7.1.3) is used to define the concept of module (respectively vertex) position in a correspondence template in the obvious manner. For every template, instructions, akin to those introduced in sections 5.1.3 and 7.1.3, are generated for each module and vertex position. Given a module in the schematic (respectively vertex in the layout), the procedure of section 5.1.3 (respectively section 7.1.3) is used to determine if the module (respectively vertex) appears at a certain position in an occurrence of a given correspondence template.

Because the schematic portion of a correspondence template $\mathcal{T}$ and any occurrence $\mathcal{T}_i$ of $\mathcal{T}$ need only be semi-isomorphic, the procedure of section 5.1.3 is modified to identify networks that are semi-isomorphic instead of *strictly* isomorphic to the network of the correspondence template. This is achieved by allowing nets to appear in one or several net slots (described in section 5.1.3 and shown in figure 5-3) instead of requiring that the nets in the slots be distinct.

10.1.3 Servicing an Event

Finding Potential Occurrences

When an event is serviced it is removed from the event queue. For each correspondence template $\mathcal{T}$ and position $\mathcal{P}$ in which a vertex of the type T of the event vertex V_e appears, it is checked whether V_e, appears in a subgraph isomorphic to the layout portion of the correspondence template at position $\mathcal{P}$. This is accomplished using the occurrence positions list (described in section 7.1.3) for vertices of type T and the procedure of figure 7-3. If such a subgraph G_l exists, the schematic position of the occurrence must then be found.

At least one of the modules in the schematic portion of the occurrence can be identified using the mappings of the vertices because at least one of the modules in G_l, the event module V_e, has mappings. Let V_c be a vertex in G_l with mappings and V_t be the vertex in the template $\mathcal{T}$ to which it corresponds. Let M_t be the module in $\mathcal{T}$ mapped to V_t via mapping σ_t. $\mathcal{P}_t$ is the position in which M_t appears in $\mathcal{T}$. Vertex V_c must be mapped to a module M_c in the schematic via mapping σ_t otherwise since it is already mapped it cannot have mapping σ_t. In this case the mapping of some of the vertices in G_l are inconsistent with those of $\mathcal{T}$ and therefore V_e cannot be in an occurrence of $\mathcal{T}$ at position $\mathcal{P}$.

For there to be an occurrence of correspondence template $\mathcal{T}$, module M_c in the schematic must now correspond to module M_t in $\mathcal{T}$. It is therefore necessary that M_c appear in a network isomorphic to the schematic portion of $\mathcal{T}$ at the position of module M_t. This is accomplished using the procedure of figure 5-5. If such a network N_s exists it remains to be verified that the mappings of the modules in N_s and the vertices in G_l are consistent with those of the corresponding modules and vertices in $\mathcal{T}$.

Mapping Validation

Let N_s and G_l be a network and sub-graph isomorphic to the schematic and layout portions of $\mathcal{T}$ respectively found using the procedure described above. Some of the modules and vertices in N_s and G_l may have mappings and for some of them the mappings may not yet have been defined. In order for N_s and G_l to form an occurrence of $\mathcal{T}$ the mappings of the modules in N_s and vertices in G_l (which have mappings) must satisfy certain conditions.

For each module M_m in N_s which has mappings, if M_s corresponds to module M_t in $\mathcal{T}$ and M_t is mapped to vertex V_t via mapping σ_t then M_m must be mapped via σ_t to a vertex V_m in G_l which corresponds to V_t in $\mathcal{T}$. The same must be true for every vertex

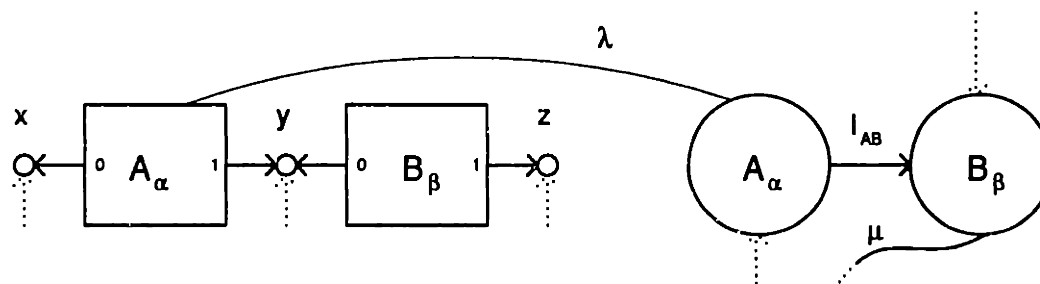


FIGURE 10-1: Mapping Validation

in G_l.

For instance, a necessary condition for the network and sub-graph of figure 10-1 to be an instance of correspondence template T_1 shown in figure 9-6 (a) is that, if mapped, vertex β on the RHS of figure 10-1 be mapped to module β on the LHS of figure 10-1 via mapping μ. Since vertex β is already mapped to a module different from module β, the condition mentioned in the previous paragraph cannot be satisfied.

If N_s and G_l form an occurrence of $\mathcal{T}$ then the modules and vertices in N_s and G_l have must mappings corresponding to the mappings of the associated modules and vertices in $\mathcal{T}$. For each unmapped module M_u in N_s which corresponds to M_t in $\mathcal{T}$ and for each mapping σ_i from M_t to a vertex V_i, a mapping σ_i is added between M_u and the vertex V_u in G_l which corresponds to V_i.

Rescheduling Newly Mapped Vertices

For each vertex that is given a new mapping an event is created and placed on the event queue. Hence soon after the mapping of a component becomes known all potential template occurrences in which it might appear are examined. If the layout portion G_l of a potential occurrence is found at least one of the vertices in G_l (the event vertex) is guaranteed to be mapped and hence a starting point for finding the schematic portion of the occurrence can always by found. Hence by guaranteeing that each event vertex is mapped, computation effort is expended locating occurrences only when it is guaranteed that a starting point for the schematic portion of the occurrence can be found.

Mapping Conflicts

Before creating new mappings for the unmapped modules and vertices in N_s and G_l it must first be verified that no other occurrences which would cause these modules and vertices to be mapped differently can be found. If a mapping conflict occurs then at this point in the correspondence verification process it is not known if N_s and G_l constitute

an occurrence of $\mathcal{T}$.

In this case the system assumes that N_s and G_l are not an occurrence of $\mathcal{T}$. Often during the processing of some other event the mapping ambiguity is resolved and some of the unmapped vertices in G_l (and modules in N_s) are given mappings. Events for these newly mapped vertices are created and put on the event queue.

When an event corresponding to one of these vertices gets serviced the network N_s and sub-graph G_l necessarily get examined again to see if they form an occurrence of $\mathcal{T}$. This time since more modules in N_s and vertices in G_l have mappings there is a better chance of resolving mapping conflicts.

Templates needed to resolve Mapping Conflicts

Without loss of generality it can be assumed that each template contains two minimal regions of equivalence. Let $\mathcal{I}$ be an interface between two vertices in G_l where the vertices are in two different regions of equivalence. If $\mathcal{I}$ is to appear in a template occurrence then the layout portion of the occurrence must entirely contain G_l. The mappings of the vertices in G_l must therefore necessarily be those corresponding to a template whose layout portion is isomorphic to G_l [1].

This means that in order to verify that there are no other template occurrences that would cause the unmapped modules and vertices in N_s and G_l to have mappings different from those imposed by the occurrence of $\mathcal{T}$, it suffices to consider only those other occurrences whose layout portion is isomorphic to G_l. These occurrences must contain the event vertex V_e and hence it is possible to locate them during the servicing of the event for V_e.

Connecting Net Connection Graphs and marking Interfaces

For each net in the schematic, a net connection graph is maintained. For every net η which appears in the occurrence of $\mathcal{T}$, if ν_η and μ_η are two vertices in the net connection graph of η which correspond to pins of modules in the occurrence of $\mathcal{T}$ then an edge is added between ν_η and μ_η. Similarly, each interface which appears in the occurrence is marked with a special label indicating that it has appeared in a template occurrence.

[1] If no such template exists then the interface $\mathcal{I}$ cannot appear in a template occurrence and hence the schematic cannot match the layout.

10.1.4 Complete Algorithm

The correspondence verification algorithm can be summarized by the three steps described below.

1. A few user provided mappings between modules and vertices are required to start the algorithm. Events for the mapped vertices are created and put on the event queue. The greater the number of mappings provided by the user, the fewer the chances of running into mapping ambiguities that cannot be resolved by the system and hence the better the chances of a successful match.

2. Each event is serviced as described in section 10.1.3. New events get generated when unmapped vertices are given mappings. When occurrences are found, some of the vertices in net connection graphs are connected together via edges and some of the interfaces are marked.

3. Verify that each net connection graph is connected and every interface is marked. Report any unconnected net connection graphs and unmarked interfaces as errors.

Example

Figure 10-2 shows a correspondence template for two inverters. This template is used to verify the correspondence of the schematic and layout in figure 10-3 (a). In figure 10-3 (a) the schematic and layout have just been read into SCHEMILAR and a user specified mapping between module $M1$ and vertex $V1$ is created. An event for vertex $V1$ is placed on the event queue and the processing of events is enabled.

Figure 10-3 (b) shows the state of the verification after the event for $V1$ is serviced. An occurrence of $\mathcal{T}$ consisting of vertices $V1$ and $V2$ and modules $M1$ and $M2$ is found. Module $M2$ and vertex $V2$, previously unmapped, are mapped[2] to each other in accordance with $\mathcal{T}$ and an event for vertex $V2$ is created and placed on the event queue. The two pins in the net connection graph of net η now become connected. This is indicated by the ∪ shaped arrow between the connections to η. The interface between $V1$ and $V2$ is marked by a $\surd$ to indicate that it has appeared in a correspondence template occurrence.

Figure 10-3 (c) shows the state of the verification after the event for $V2$ is serviced. Vertices $V1$ and $V2$ and modules $M1$ and $M2$ are found to appear in an occurrence of $\mathcal{T}$. For this reason a mapping is created between module $M3$ and vertex $V3$ and an event for vertex $V3$ is created and placed on the event queue. The two pins in the net

[2]After verifying that the mapping is not ambiguous.

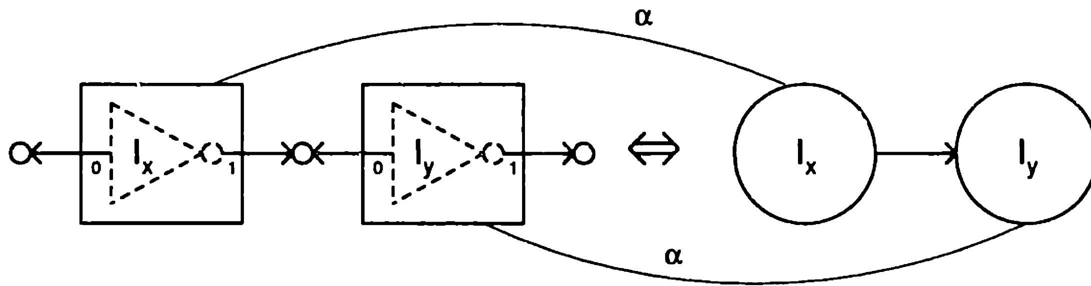


FIGURE 10-2: Inverter Template

connection graph of net μ become connected as shown and the interface between vertices $V2$ and $V3$ are marked with a $\surd$.

No additional template occurrences of $\mathcal{T}$ are found during the servicing of the event of $V3$. Since all net connection graphs are connected and all interfaces are marked the schematic and layout match.

Increasing Verification Speed

The technique for increasing layout verification speed introduced in section 7.1.3 is used in SCHEMILAR to reduce correspondence verification time. Each vertex V_i in the layout maintains a list of positions (called the *occurring positions* list) in which it appears in the layout portion of a template occurrence. The elements of the list are the vertices in the templates which correspond to V_i in each occurrence. The list is also used during incremental updates to the layout.

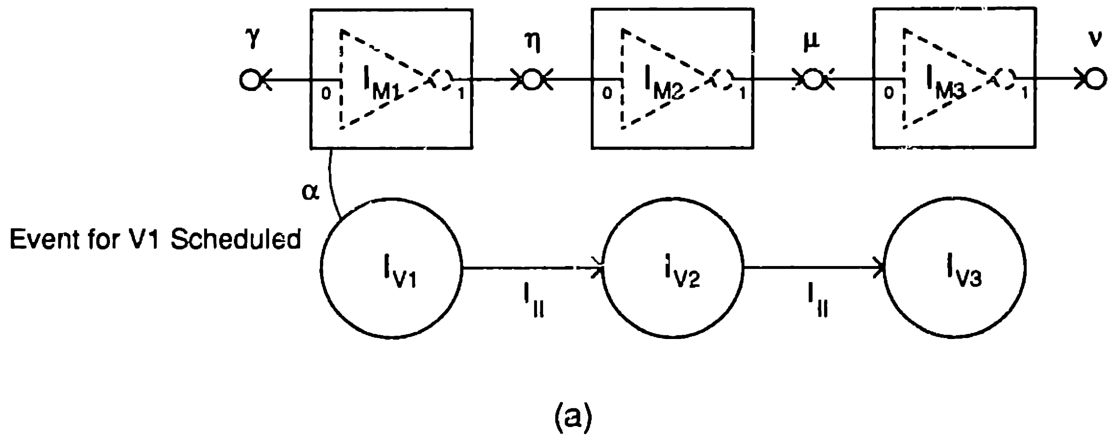
From section 7.1.3 it is known that a given vertex V_e in the layout can appear in only one sub-graph G_l isomorphic to the layout portion of a template $\mathcal{T}$ in position $\mathcal{P}$. Given a set of mappings for the elements of G_l, there can be at most one schematic network N_s in correspondence with G_l. Hence there can be only once occurrence of $\mathcal{T}$ in which V_e appears in position $\mathcal{P}$. Therefore during the processing of the event for vertex V_e the system checks if V_e appears in an occurrence of $\mathcal{T}$ at position $\mathcal{P}$ only if that position does not already occur in the *occurring positions list* of V_e.

Additional verification speed can be achieved by carefully designing the schematic instructions as described in section 5.1.4 so that nets with the least number of pins connected to them are searched.

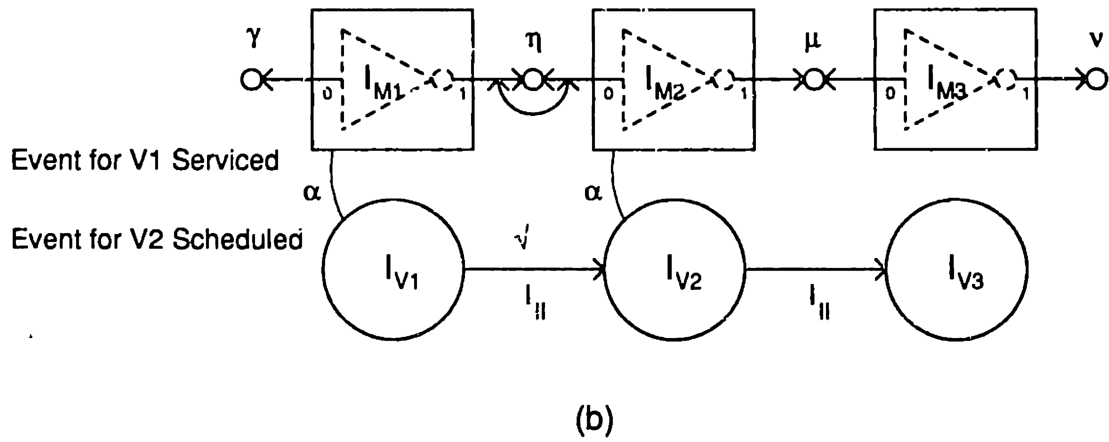
10.1.5 Algorithm Complexity

Definitions

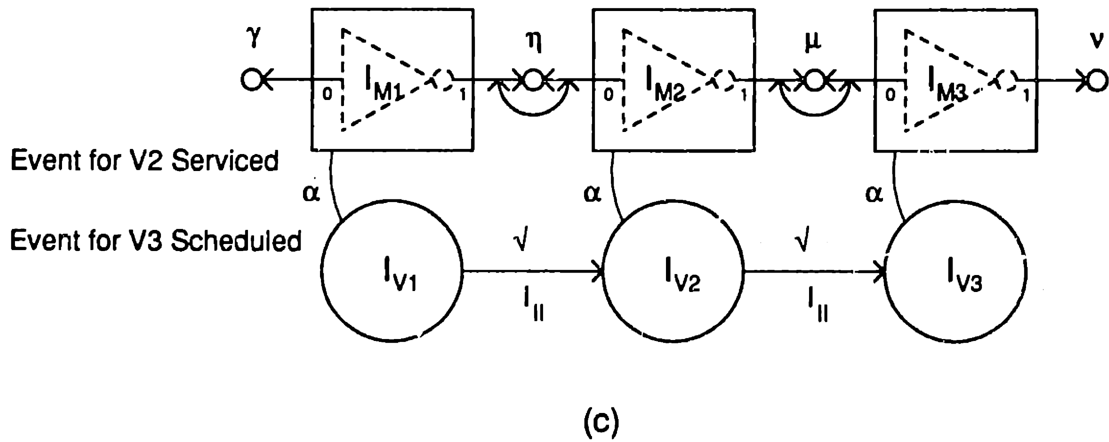
- β_{max} is the maximum number of mappings any vertex in any template can have.



(a)



(b)



(c)

FIGURE 10-3: Verification Example

- n_{vertex} is the number of vertices in the layout.
- n_{mod} is the number of modules in the schematic.
- T_{max} is the maximum number of vertices in a correspondence template.
- R_{max} is the maximum number of modules in a correspondence template.
- L_{max} is the maximum length of the list of positions (introduced in section 7.1.3) for any cell.
- ρ_η is the number of pins connected to net η.
- α_{pins} is the total number of pins in the schematic.
- σ_{max} is the maximum number of pins any module can have.
- ϵ_{max} is the maximum number of interfaces (interface types) a vertextype can have.

Number of Events

Vertices which do not contain bus structures are mapped (at most) once. Vertices that contain bus structures may be mapped several times as described in section 9.4.2. The maximum number of times a bus vertex can be mapped is certainly less than β_{max}. Typically β_{max} is less than three. Since each event corresponds to a vertex being mapped, the number of events is less than $n_{vertex} \cdot \beta_{max}$.

Finding Potential Occurrences

From section 7.1.4 it is known that the time required for identifying the layout portion of an occurrence is less than $O(T_{max})$. From section 5.1.4 it is known that the time required for identifying the schematic portion of the occurrence is typically less than $O(2^{R_{max}})$, (actually $O(n_{mod}^{R_{max}})$ in the worst case). The time required to compute all the possible occurrences that the event module V_e can appear in is typically less than $O(L_{max} \cdot T_{max} \cdot 2^{R_{max}})$. To simplify the equations, since T_{max} and R_{max} are independent of circuit size $O(T_{max}) = O(2^{R_{max}}) = O(1)$ will be assumed.

Mapping Validation

Having computed the potential occurrences, it remains to identify the ones which have incompatible mappings for the unmapped modules and vertices. This can be accomplished with $O(L_{max}^2)$ comparisons by examining each pair of occurrences.

Connecting Net Connection Graphs and marking Vertices

When an occurrence of a template $\mathcal{T}$ is found, edges are added to the connection graphs of nets in the occurrence and interfaces in the occurrence are marked. The number of vertices in the net connection graph that need to be connected as well as the number of interfaces that need to be marked depends on the template and is independent of schematic and layout size. The values will be assumed to be $O(1)$.

Verifying that the Net Connection Graphs are connected and that the Vertices are marked

Because the connection graph of η has ρ_η vertices, the time taken to verify that the net connection graph of η is connected is $O(\rho_\eta)$. Since the total number of pins in the schematic is $\alpha_{pins} = \sum_\eta \rho_\eta$ the time required to verify that all the connection graphs in the schematic are connected is $O(\alpha_{pins}) \leq O(\sigma_{max} \cdot n_{mod}) = O(n_{mod})$. The time required to verify that all the interfaces are marked is less than $O(n_{vertex} \cdot \epsilon_{max}) = O(n_{vertex})$.

Total Time

The total verification time required is equal to the total number of events multiplied by the time required to process an event plus the time required to verify that all connection graphs are connected and all interfaces are marked. Taking $O(\beta_{max}) = O(L_{max}) = O(1)$, this time is $O(n_{vertex} \cdot \beta_{max}) \cdot (O(L_{max}) + O(L^2_{max})) + O(n_{mod}) + O(n_{vertex}) = O(n_{mod}) + O(n_{vertex})$ which is linear in the number of modules and vertices.

10.1.6 Incremental Update to the Layout or Schematic

The correspondence verification algorithm in SCHEMILAR can be modified to deal with incremental additions and deletions to the layout and incremental deletions to the schematic. Adding an element to either the schematic or the layout requires that an event for that element be created and placed on the event queue. Incremental additions to the schematic can be dealt with by allowing events for both schematic modules and layout vertices.

Adding a Vertex

When a vertex is added to the layout, an event for it is created whether it is mapped or not. When the event is processed, all occurrences which contain that vertex will be found and the ultimate result of the comparison will be the same as if the vertex had been

added to the layout before the verification process was started. If a module (mapped or unmapped) is added to the schematic an event for that module is created and placed on the event queue provided the algorithm knows how to service schematic events.

Deleting a Module or Vertex

When a schematic module or layout vertex is deleted, all the mappings between modules and vertices in the corresponding minimal region of equivalence must be deleted. Let $\mathcal{O}$ be the set of occurrences in which the deleted element appears. The effect of all occurrences in $\mathcal{O}$ must then be undone. Hence any edges in any net connection graph which correspond to an occurrence in $\mathcal{O}$ must be deleted. Each interface must maintain a list of occurrences in which it appears. The interface is considered to be marked if this list is not empty. For any interface $\mathcal{I}_i$ which appears in an occurrence in $\mathcal{O}$, all occurrences in $\mathcal{O}$ must be removed from the occurrence list of $\mathcal{I}_i$.

The validity of the mappings of modules and vertices which were originally mapped during an occurrence in $\mathcal{O}$ may be compromised. These components may have to be unmapped. Unmapping a vertex V_i causes every component in the minimal region of equivalence of V_i to become unmapped. For each vertex V_i, let $\mathcal{O}_i$ be the set of occurrences it appears in. Because the original mappings of V_i may no longer be valid these occurrences must be deleted. This in turn may cause the mappings of modules and vertices which were originally mapped during an occurrence in $\mathcal{O}_i$ to be compromised which requires more components to be unmapped etc. The process continues until no more vertices need to be unmapped. Enabling the processing of events will then cause the match to be in the same state as if the deleted element was not in the circuit at the time the verification was started. It is computationally advantageous to manually check to see if the mappings that have been compromised by the deletion cannot be retained.

10.1.7 Error Reporting

When the event queue becomes empty all the possible correspondence template occurrences that can be inferred from the initial mappings have been found. If the match fails valuable information can be provided to the user to help him locate the source of the mismatch.

The existing mappings can be used to identify the layout and schematic components that are known to be in correspondence with each other. Errors are usually located on the boundaries of mapped and unmapped regions of the schematic and layout.

The unconnected net connection graphs contain valuable information on electrical

connections between modules that cannot be accounted for by the correspondence verification system. The connected components of the graph represent pins that are known to connect together. Electrical connections between pins in different connected components of a net connection graph cannot be verified by the system and are hence suspect. Similarly, unmarked interfaces correspond to electrical connections in the layout netlist that the system cannot account for and are therefore suspect.

When the system fails to match the schematic and the layout, the user must decide (with the help of the information provided to him by the system) whether the failure is caused because the schematic and layout are incompatible or because the system was not able to resolve mapping ambiguities.

If the schematic and layout are incompatible, the user can make incremental changes to the schematic or layout to correct them using the procedure described in section 10.1.6.

If the failure was caused by a mapping ambiguity, the user (armed with the knowledge of the existing mappings) is usually in a position to resolve the ambiguity. By creating mappings for some of the unmapped vertices (and placing these vertices on the event queue) the user can help the system resolve mapping ambiguities.

10.2 Implementation

Source Code

SCHEMILAR is written in C and runs on an HP 9000 series model 350 workstation running UNIX and Xwindows. Much of the source code is borrowed from GRASP and GLOVE. The source code consists of approximately 17000 lines out of which the core algorithm takes 4000 lines. SCHEMILAR first reads in a specification of the correspondence templates then reads in a schematic netlist and a layout connectivity graph and finally proceeds by comparing the netlist with the graph.

If SCHEMILAR is unable to match the schematic and the layout then the set of unconnected net connection graphs and the set of interfaces not accounted for is reported. For each unconnected connection graph the state of the graph and the modules and pins in each of the connected components of the graph is reported. The user can at any time during execution examine the mappings between the modules and the vertices as well as the state of a net connection graph or interface

```
(sl_template  2-inverters (2 3 2)
                 % Template has: 2 modules, 3 nets and 2 vertices
               (moduletypes inverterm inverterm)
                 % Moduletypes of the schematic modules.
                 % The first module is module 0, the second module 1 etc..
                (vertextypes  inverterv inverterv)
                 % Celltypes of the layout vertices
                 % The first vertex is vertex 0, the second vertex 1 etc..
              (module_conn (0 (0 1))    % Connections of module 0
                           (1 (1 2)))   % Connections of module 1
              (vertex_conn (0 1 1))     % Vertex 1 and 4 have are connected
                                        % via interface 1
              (mappings (0 0 inverterm2inverterv) (1 1 inverterm2inverterv))
                 % module 0 is mapped to vertex 0 via mapping
                 % 'inverterm2inverterv'.
                 % module 1 is mapped to vertex 1 via mapping
                 % 'inverterm2inverterv.'
)
```

FIGURE 10-4: Textual Representation of a Correspondence Template

Input Files

Since the actual geometrical values of the interfaces are not needed during correspondence verification, the templates are specified in Lisp-like format resembling that of the GRASP production file. The syntax of the correspondence template file is shown in figure 10-4. The correspondence template used in figure 10-4 is the one shown in figure 10-2 which is described in the example of section 10.1.4.

The format of the schematic file is almost identical to the one used by GRASP shown in figure 5-11. The only difference is that the user can specify mappings to some of the layout vertices. These mappings are needed to start the event driven algorithm. The format of the layout file is identical to that used by GLOVE shown in figure 7-4. Figure 10-5 shows the file required to describe the schematic of figure 10-3 (a) and figure 10-6 shows the file for the corresponding layout also shown in figure 10-3 (a). The term `'(0 inverterm2inverterv)'` on the third line of the schematic file of figure 10-5 indicates that the module on that line is mapped via mapping `'inverterm2inverterv'` to vertex 0 which appears on the third line of the layout file of figure 10-6.

```
Number_of_nets: 6
%Schematic array of 5 inverters
inverterm 0 1 (0 inverterm2inverterv)
inverterm 1 2
inverterm 2 3
```

FIGURE 10-5: Schematic Input Netlist

```
Number_of_nets: 6
%Layout array of 5 inverters
0 inverterv /f 1 1 ;
1 inverterv /b 0 1 /f 2 1 ;
2 inverterv /b 1 1 ;
```

FIGURE 10-6: Layout Input Graph

Algorithm

Events in SCHEMILAR correspond only to vertices (and not to modules) as described in section 10.1.2. Incremental update is not supported. When an event for vertex V_e is being serviced and a potential occurrence O_i is found, all the potential occurrences in which V_e appears are recomputed and checked to see if their mapping requirements conflict with that of O_i. A substantial increase in speed is possible by modifying the code to store the list of potential occurrences and compute each one only once.

Each net has an associated blank module which is automatically created when the schematic file is read in. The criteria for correctness is adapted to reflect the presence of these blank modules. The net connection graph for each net η must be connected except perhaps for the vertex associated with the blank module connected to η.

Net Connection Graph Implementation

To conserve memory and keep the data structures simple, the connection graph for each net is implemented using an array of numbers. Each entry in the array corresponds to a vertex in the connection graph. The connected components of the graph are given consecutive numbers starting with 0. The value of an element in the array corresponding to vertex ν is the index number of the connected component in which ν appears. Since initially the graph has no edges, each element in the array initially has a different number.

When vertices ν and μ are connected together, the two connected components in

which each of them appear are merged together. If the array slot of ν contains number α and the slot of μ contains β with $\alpha \leq \beta$ then all the array slots which have value β have α copied into them. This corresponds to adding all the vertices in component β to component α and then deleting component β. If all the elements in the array contain a 0 then the net connection graph is connected. This method corresponds to a linear time implementation of the `union-find` algorithm [2][3]. This technique is efficient enough for most nets but is slow for the **vdd**, **gnd** and clock nets which have very large numbers of pins connected to them.

Graphics Display

An Xwindows graphics interface built into SCHEMILAR is capable of simultaneously displaying the schematic and the layout. For each template occurrence in which the event vertex appears, a connected neighborhood of the minimal region of equivalence of the vertex in both the schematic and layout is displayed. The layout is displayed in the two column format used in GLOVE shown in figure 7-5. The event vertex appears on the top left corner of the screen. The schematic is displayed in the format used in GRASP shown in figure 5-12. Modules are displayed on the left side of the window and nets on the right side. The module used as a starting point for finding the schematic portion of the occurrence appears at the top left corner of the screen. The mappings of any module or vertex on the screen can be examined. The state of any net connection graph or any interface can be printed out as well.

10.3 Experiments

Correspondence templates for the multiplier of figure 5-13 and the PLA of figure 6-1 have been designed and tested for various multiplier and PLA sizes. Both the multiplier and PLA make heavy use of cell encoding which is handled effectively within the existing framework. Regular structures such as the multiplier and PLAs were chosen to facilitate the generation of the test cases. Programs that can simultaneously generate the schematic and layout for any sized multiplier and any sized and encoded PLA have been designed to generate these test cases. Debugging of the code was also facilitated by the regularity of these structures. SCHEMILAR does not exploit this regularity and as such the correspondence verification speed results of tables 10-1 and 10-2 are similar to those that would be obtained for less regular structures.

[3] A logarithmic time implementation of this algorithm exists.

Multiplier size	Number of instances	Number of Modules	Verification time
5×5	102	107	3s
10×10	402	412	13s
20×20	1602	1622	62s
30×30	3602	3632	174s

Table 10-1: Multiplier Correspondence Verification Times

10.3.1 Bit Systolic Multiplier

The schematic and layout of the bit systolic multiplier described briefly in section 5.3 and in more detail in [8] has been compared using SCHEMILAR for various multiplier sizes. The multiplier layout makes heavy use of encoded cells. Each full adder cell in the multiplier layout consists of a basic cell which is personalized by three encoding cells according to its location in the array as prescribed in [8]. Twenty-two correspondence templates were used to describe the relation between the multiplier layout cells and their encodings and multiplier schematic modules. A single mapping between a schematic module and an encoding vertex in one of the full adder cells in the multiplier was provided by the user. All other mappings were computed by SCHEMILAR. Table 10-1 summarizes the correspondence verification times[4] for various multiplier sizes. The schematic of the multiplier contains composite modules and hence the number of transistors in a netlist of the multiplier is considerably larger than the number of modules in the schematic.

10.3.2 PLA

The PLA of figure 6-2 represents one of the more difficult verification cases for SCHEMILAR. First since most PLAs are sparse, the number of transistors is small compared to the number of layout vertices. Also, the PLA has many similar electric paths in parallel. SCHEMILAR cannot by itself identify which input buffer in the schematic corresponds to a given in_a vertex in the layout. The same is true for the out_o and the and_pu cells. For each of these cells a mapping between the vertex and at least one of its corresponding modules in the schematic is needed for the match to be successful.

Figure 10-7 shows the circuit for one path in the PLA. The modules shown in figure 10-7 are those used in the schematic of the PLA. The layout cells to which these modules correspond is outlined by the dotted line. In reality each cell is more complex than shown in figure 10-7. Each in_a cell feeds the input and its complement to the AND plane and each out_o cell shares two outputs. Finally the and_sq and or_sq cells each

[4]The time required to read in the templates as well as the schematic and layout is not included.

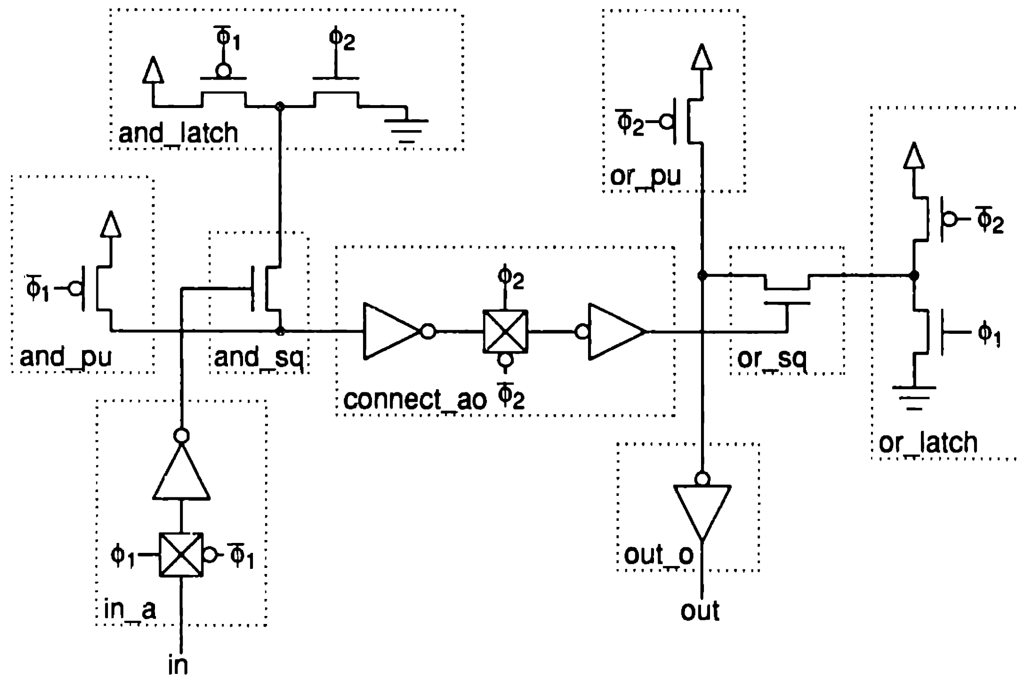


FIGURE 10-7: PLA Path

PLA size	Number of instances	Number of Modules	Verification time
5 × 10 × 20	236	295	4.5s
10 × 10 × 20	326	370	8s
20 × 20 × 40	1164	958	46s
40 × 40 × 80	4267	2601	303s

Table 10-2: PLA Correspondence Verification Times

share two bit-lines and word-lines.

Table 10-2 summarizes the correspondence verification times for various sized and encoded PLAs. The PLA size is given as inputs×outputs×terms. Because the or_sq and and_sq cells are shared between several bit-lines and word-lines, the number of vertices is comparable to the number of modules even though the PLAs are sparse.

11

Conclusions

11.1 Summary

A novel set of representations and formalisms for both schematics and layouts which cleanly captures structural design constraints has been presented. These representations and formalisms allow the structural correctness of a schematic or layout as well as schematic vs. layout correspondence to be verified. Fast non-heuristic verification techniques with one basic structural recognition method for all three verification areas are introduced. The proposed techniques are applicable over a continuum of module and cell sizes allowing the verification to proceed at a fine grained or coarse grained level. The proposed techniques have been implemented in three computer programs GRASP, GLOVE and SCHEMILAR (corresponding to ERC, DRC and CV respectively) which share the same basic event driven structure finding algorithm. Experiments with these programs show that they are both fast and that practical designs can be effectively verified with them.

For schematic design style verification, context free circuit grammars are used to define design style correctness and grammatical parsing is used to verify that a given schematic obeys the design style. The use of circuit grammars is made possible through the introduction of a technique called net-bundling in which individual nets are encapsulated into packets which are dealt with as one physical object.

Layouts are modeled as connectivity graphs and structural constraints are captured by user defined graph templates comparable to grammatical productions used to verify schematics. DRC verification is performed by *covering* the layout graph with these templates.

Finally, schematic vs. layout correspondence verification is accomplished by simultaneously covering the schematic and layout to be compared with correspondence tem-

plates. These are similar to layout templates but span both schematics and layouts. This verification method requires no netlist extraction for either the schematic or the layout and allows matching between schematics and layouts whose netlists are not necessarily isomorphic.

Figure 11-1 shows the different possible inputs and outputs of GRASP, GLOVE and SCHEMILAR. By using a design style grammar, the schematic parser GRASP can be used to build a parse tree which can prove that the input schematic obeys the design style as shown in figure 11-1 (a).

Using a special grammar, GRASP can also be used to reduce a schematic into large functional blocks (as shown in figure 11-1 (b)) to facilitate matching of schematics and layouts with non-isomorphic netlists. The schematic vs. layout correspondence verification program SCHEMILAR takes a schematic, a layout connectivity graph (which can be extracted from a mask layout) and a set of correspondence templates and reports whether the schematic and the layout match. The schematic input to SCHEMILAR can be either the schematic itself or the reduced schematic as mentioned above. The layout input to SCHEMILAR requires that first it be verified by the layout verifier GLOVE as shown in figure 11-1 (b).

Finally, GLOVE can be used solely to verify the DRC correctness of a layout as shown in figure 11-1 (c).

11.2 Future Work

11.2.1 Extensions

Attribute Circuit Grammars

Since modules are not allowed to have attributes (such as delay, transistor or capacitor size etc.), circuit constraints such as logic gate maximum fanout, W/L ratioing, maximum delay, charge sharing etc. cannot easily be captured by the grammatical formalisms of chapter 3. These constraints require that module parameters not specified by the module type be known.

It is possible to extend the set of design constraints that can be verified by associating a set of attribute types $\alpha_i^{\mathcal{T}}$ with each module type $\mathcal{T}$ and requiring that any module M of type $\mathcal{T}$ have associated attributes of types $\alpha_i^{\mathcal{T}}$. In this augmented verification strategy, the grammatical productions specify constraints on the attributes of the modules in the network to be reduced and attributes for the new composite modules are computed in terms of the attributes of the above mentioned modules.

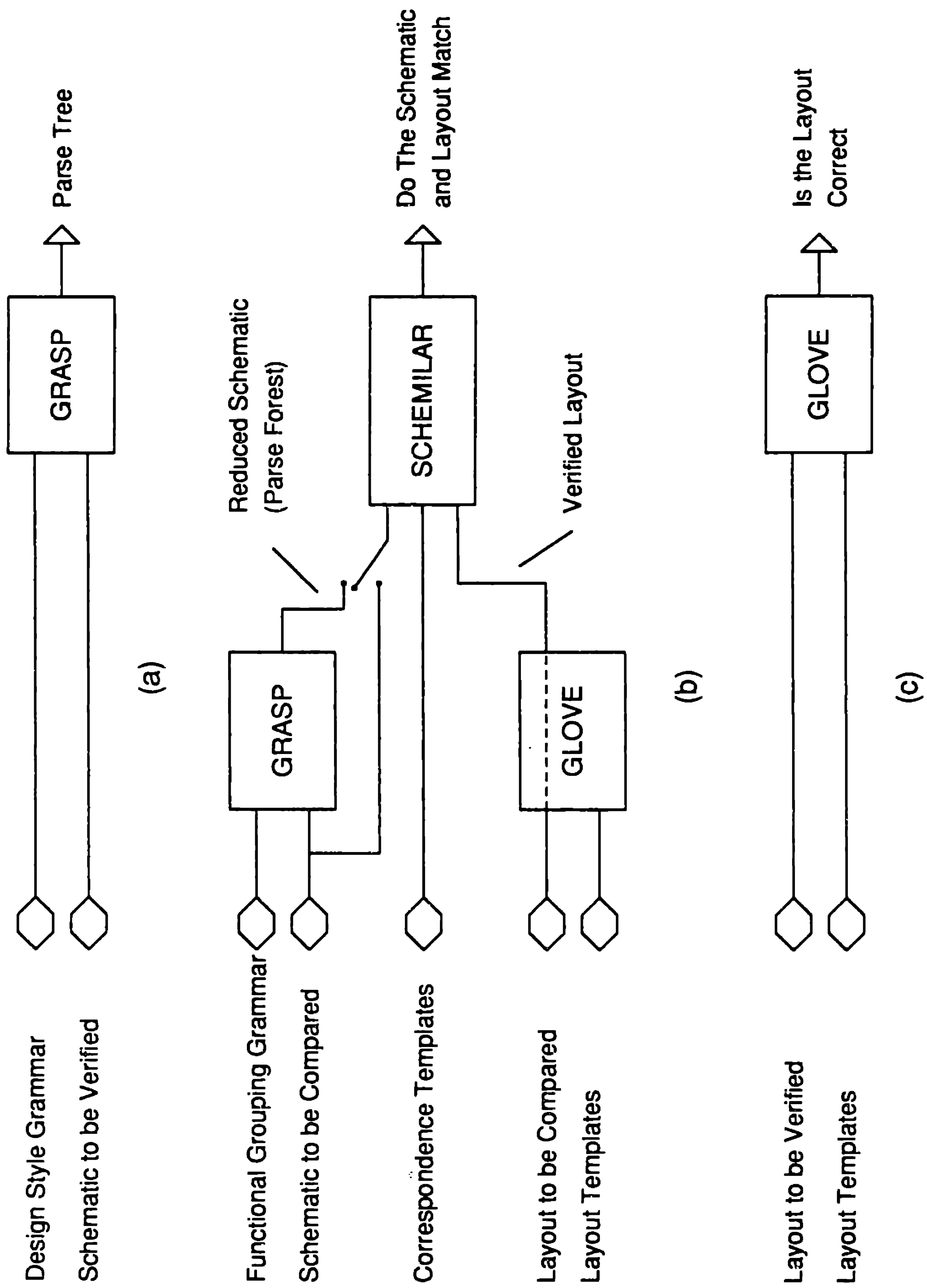


FIGURE 11-1: Program Inputs and Outputs

Automatic Condition Module Generation

In order for GRASP to be able to perform its parsing functions deterministically, conditi^~ modules for each production in the circuit grammar must be specified. The design of a circuit grammar can be greatly facilitated if the condition modules (or some other equivalent conditions) are automatically extracted from the set of circuit grammar productions.

Grammar Range Space Characterization

The ability to relate an independent (non-grammatical) characterization of the design methodology with the range space of its assumed grammar can be of considerable help during grammar design. This can be used to prove that any circuit in the assumed grammar's range space satisfies all the structural constraints of the design methodology and that the grammar is complete (its range space covers all such circuits). The technique could also be used to grammatically characterize and evaluate circuit methodologies for new technologies.

11.2.2 New Directions

The grammatical and template based techniques presented in this thesis are geared toward structural verification. The possibility of using similar techniques in synthesis needs to be researched. During synthesis, circuit grammars and correspondence templates are used to explore structurally different implementation possibilities.

In one possible method, *decompositional* grammars are defined for each large functional block. The range space of these grammars is the set of all schematics that implement the functionality of the associated block. For schematics composed of these large functional blocks, schematic alternatives are explored by examining the different possible grammatical expansions of the blocks using the productions in the *decomposition* grammar.

Correspondence templates are then used to generate possible layout alternatives for that schematic. These alternatives are such, that using the correspondence templates, the schematic and layout can be shown to be in correspondence with one another. Each possible alternative may have to be verified for structural correctness. The schematic block expansion, schematic to layout *translation* and structural verification can all be performed incrementally and in parallel.

These above mentioned techniques provide a structured approach for examining layout alternatives for a given functionality. Exhaustive enumeration and evaluation of all

possible layout alternatives is not practical. For these techniques to be successful, a necessary pre-requisite is an effective strategy for steering the exploration process toward a desirable solution.

Bibliography

[1] F. Van Aelten. *Efficient Verification of* VLSI *Circuits Based in Syntax and Denotational Semantics.* Master's thesis, MIT, 1989.

[2] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*, chapter 4, pages 124–145. Addison Wesley, 1974.

[3] A. Aho and J. Ullman. *Principles of Compiler Design.* Addison Wesley, Reading Massachusetts, 1979.

[4] A. Aho and J. Ullman. *Principles of Compiler Design*, chapter 6, pages 229–233, 241. Addison Wesley, 1979.

[5] R. Armstrong. *HPEDIT Reference Manual.* MIT Research Laboratory of Electronics, 1982.

[6] R. Armstrong. *User's Guide to XDRC.* MIT Research Laboratory of Electronics, 1982.

[7] M. Arnold and J. Ousterhout. Lyra: a new approach to geometric layout rule checking. In *ACM IEEE* 19th *Design Automation Conference*, pages 530–536, 1982.

[8] C. Bamji. *A Design-by-Example Regular Structure Generator.* Master's thesis, MIT, 1985.

[9] C. Bamji and J. Allen. GRASP: a **Gra**mmar-based **S**chematic **P**arser. In *ACM IEEE* 26th *Design Automation Conference*, pages 448–453, 1989.

[10] C. Bamji, C. Hauck, and J. Allen. A design-by-example regular structure generator. In *ACM IEEE* 22nd *Design Automation Conference*, 1985.

[11] J. Bentley and T. Ottmann. Algorithms for reporting andf counting geometric intersections. *IEEE Transactions on Computers*, C-2(9), September 1979.

[12] I. Bolsens, W. De Rammelaere, C. Van Overloop, L. Claesen, and H. De Man. A formal approach towards electrical verification of synchronous MOS circuits. In *Proceedings of the ISCAS conference*, 1988.

[13] R. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C33:160–177, 1984.

[14] R. Bryant. Symbolic verification of MOS circuits. In *Chappel Hill Conference on Very Large Scale Integration*, 1985.

[15] E. Carlson and R. Rutenbar. Mask verification on the connection machine. In *ACM IEEE* 25th *Design Automation Conference*, pages 134–140, 1988.

[16] D. Corneil and D. Kirkpatrick. A theoretical alalysis of various heuristics for the graph isomorphism problem. *SIAM Journal of Computing*, 9(2):281–297, 1980.

[17] C. Ebeling and O. Zajicek. Validating VLSI circuit layout by wirelist comparison. In *IEEE International Conference on Computer-aided Design*, 1983.

[18] N. Gonclaves and H. De Man. NORA: A racefree dynamic CMOS technique for piplined logic structures. *IEEE Journal of Solid-State Circuits*, S-18(3), June 1983.

[19] G. Goos and J. Hartmanis. *Graph-Grammars and their Application to Computer Science*. Springer-Verlag, New-York, 2nd international workshop edition, 1983.

[20] G. Goos and J. Hartmanis. *Graph-Grammars and their Application to Computer Science and Biology*. Springer-Verlag, New-York, international workshop edition, 1979.

[21] C. Hauck, C. Bamji, and J. Allen. The systematic exploration of pipelined array multiplier performance. In *Proceedings Internation Conference on Accoustic, Speech and Signal Processing*, 1985.

[22] J. Hopcroft and J. Ullman. *Automata Theory Languages and Computation*, chapter 6, page 125. Addison Wesley, Reading Massachusetts, 1979.

[23] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 14, pages 389–391,393. Addison Wesley, Reading Massachusetts, 1979.

[24] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading Massachusetts, 1979.

[25] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 9, pages 223–228. Addison Wesley, Reading Massachusetts, 1979.

[26] K. Karplus. Exclusion constraints: a new application of graph algorithms to VLSI design. In *Proceedings of the Fourth MIT Conference on Advanced Research in VLSI*, 1986.

[27] A. Kolodny, R. Friedman, and T. Ben-Tzur. Rule-based static debugger and simulation compiler for VLSI schematics. In *IEEE International Conference on Computer-aided Design*, 1985.

[28] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Proceedings of the Third Caltec conference on VLSI*, 1983.

[29] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[30] C. Lob, R. Spickelmier, and A. Newton. Circuit verification using rule-based expert systems. In *IEEE Symposium on Circuits and Systems*, 1985.

[31] H. De Man, I. Bolsens, E. vanden Meersch, and J. van Cleynebreugel. Dialog: an expert debugging system for MOS VLSI design. *IEEE Transactions on CAD of Integrated Circuits and Systems*, CAD-14(3):303, July 1985.

[32] S. McCormick. EXCL: a circuit extractor for IC designs. In *ACM IEEE 21st Design Automation Conference*, pages 616–623, 1984.

[33] J. Nievergelt and F. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the* ACM, 25(10), October 1982.

[34] R. Owens and R. Chen. Applications of graph theory. In *IEEE International Symposium on Circuits and Systems*, 1986.

[35] R. Read and C. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1978.

[36] D. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon Inc., Boston, Massachusetts, 1986.

[37] L. Seiler. *A Hardware Assisted Methodology for* VLSI *Design Rule Checking.* PhD thesis, MIT, 1985.

[38] Y. Shiran. YNCC: a new algorithm for device-level comparison between two functionally isomorphic VLSI circuits. In *IEEE International Conference on Computer-aided Design*, pages 298–301, 1986.

[39] R. Spickelmier and A. Newton. Connectivity verification using a rule-based approach. In *IEEE International Conference on Computer-aided Design*, 1985.

[40] R. Spickelmier and A. Newton. Critic: a knowledge-based program for critiquing circuit designs. In *International Conference on Computer Design*, 1988.

[41] R. Spickelmier and A. Newton. WOMBAT: a new netlist comparison program. In *IEEE International Conference on Computer-aided Design*, 1983.

[42] G. Steele. *Common LISP.* Digital Press, Bedford, MA, 1984.

[43] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programmming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[44] B. Stroustrup. *The C^{++} Programming Language.* Addison Wesley, Reading Massachusetts, 1986.

[45] T. Szymanski and C Van Wyk. Space efficient algorithms for VLSI artwork analyis. In *ACM IEEE* 20th *Design Automation Conference*, 1983.

[46] M. Takashima, A. Ikeuchi S. Kojjima, T. Tanaka, T. Saitou, and J. Sakata. A circuit comparison system with rule-based functional isomorphism checking. In *ACM IEEE* 25th *Design Automation Conference*, pages 512–516, 1988.

[47] G. Taylor and J. Ousterhout. Magic's incremental design-rule checker. In *ACM IEEE* 21st *Design Automation Conference*, pages 160–165, 1983.

[48] C. Terman. *User's Guide to NET, PRESIM, and RNL.* MIT, 1982.

[49] C. Thompson. *A Complexity Theory for* VLSI. PhD thesis, Carnegie-Mellon University, 1980.

[50] J. Tygar and R. Ellickson. Efficient netlist comparison using hierarchy and randomization. In *ACM IEEE* 22nd *Design Automation Conference*, pages 702–708, 1985.

[51] A. Vladimirescu and S. Liu. *The Simulation of* MOS *Integrated circuits using SPICE2.* ERL Memo M80-7, Berkeley, February 1980.

[52] D. Weise. *Formal Multilevel Hierarchical Verification of Synchronous* MOS VLSI *circuits.* PhD thesis, MIT, 1986.

[53] N. Weste and K. Eshraghian. *Principles of* CMOS VLSI *Design.* Addison Wesley. 1985.

[54] T. Witney. *A Hierarchical Design-Rule Checker.* Master's thesis, California Institute of Technology, 1981.

[55] Y. Wong. Hierarchical circuit verification. In *ACM IEEE* 22nd *Design Automation Conference*, pages 695–701, 1985.