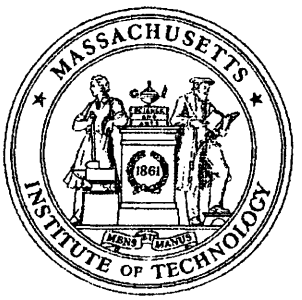

Efficient Verification of VLSI Circuits Based on Syntax and Denotational Semantics

RLE Technical Report No. 546

July 1989

Filip Van Aelten

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

Efficient Verification of VLSI Circuits Based on Syntax and Denotational Semantics

by

Filip Van Aelten

Submitted to the
Department of Electrical Engineering and Computer Science
on May 12, 1988 in partial fulfillment of the requirements
for the degree of Master of Science.

Abstract

A fully automatic circuit verification system takes in a circuit description and a specification of its behavior, and checks whether the circuit behaves as specified. Existing verification systems follow one of two approaches: a deductive approach, based on a formal logic, or a rewrite rule approach, which starts from knowledge about how transistors work, and goes straight but slowly from premises to conclusions. I present a new, more efficient approach, which incorporates large scale knowledge about VLSI circuits in a coherent fashion. The approach is based on the denotational method for defining the semantics of a programming language. A circuit is parsed according to a circuit grammar, and the resulting parse tree is mapped into a behavioral description which is matched with the user supplied specification. The new strategy is implemented in the program Semanticist.

Thesis Supervisor: Jonathan Allen
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to give special thanks to Jonathan Allen, my thesis supervisor, for his support, his encouragement, the work style that he exemplified, and his insistence on a clear exposition of my ideas. Thanks also to Cyrus Bamji, for providing the concepts and the practical support that got me started on this project, and to Ivo Bolsens, for introducing me to circuit verification.

Thanks furthermore to all the people of the eighth floor VLSI group: Bob Armstrong, Don Baltus, Srinivas Devadas, Abe Elfadel, Kevin Lam, Jennifer Lloyd, Andy Lumsdaine, Steve McCormick, Ig McQuirk, Keith Nabors, Mark Reichelt, Miguel Silveira, Dave Standley, Ricardo Telichevesky, Barry Thompson, Chris Umminger, Jacob White and John Wyatt. It was a pleasure to work in your midst.

I would also like to thank the Belgian American Educational Foundation for supporting me with a fellowship during my first year as a graduate student.

Finally, I would like to thank my parents for their support.

This report describes research done at the Research Laboratory of Electronics of the Massachusetts Institute of Technology. Support for this research was provided by Analog Devices, under a letter agreement dated 11/24/86.

Contents

Chapter 1

Introduction

1.1 Computer Aided Design and Formal Verification

Building VLSI chips with hundreds of thousands of transistors on a die of about $1cm^2$, is quite different from most other activities people are engaged in. The chips, though gigantic constructions, are shaped at once, and are expected to work the first time. VLSI designers can't continuously interact with the physical world. They can't start with a partial physical implementation, see how it works, modify it, see again how it works, go on with other parts, and so on. There is no game of action and counteraction with the physical world, as in most of our activities. Chip builders have to "sit and think" and perform one physical action that is immediately right.

Fortunately, designers don't literally have to sit and think until a correct design has evolved in their head. They can make symbolic representations of the design, concentrate on one part at a time, and store all the representations they made before without having to remember them. Better still, they can use computers that manipulate symbolic representations in interesting ways. One interesting computer configuration is a simulator, which takes in symbolic representations of input signals and behaves just the same way the physical device would operate on the physical input signals. A VLSI designer can then interact with his simulator as if he interacted with the physical world.

Unless he simulates all the situations that can possibly occur on the circuit, which is out of the question for very large circuits, he still won't be sure that his design is correct. A second kind of Computer Aided Design tool is a verification program that takes a description of the design and answers whether, to its knowledge, the design is correct or not. Still other tools are able to actually produce designs from a given specification.

All these tools manipulate symbolic representations according to some formal laws. One who wants to design the tools, embarks on a quest for formal mechanisms that are correct and efficient. Formalisms that are compact, and that have clarity and theoretical precision, are a great help. A set of ad hoc rules is just as formal as any other prescription, but building a correct and efficient program from it is less likely to succeed than building one from well structured and orderly organized laws.

The subject of this thesis is formal verification. Given a circuit description and a behavioral description, how do you determine whether the circuit behaves as specified in the behavioral description? How do you build a mechanical inference system, that takes in symbolic descriptions of a circuit and its intended behavior, and infers whether the circuit is correct or not?

Inference systems are at the heart of traditional Artificial Intelligence research. A central theme in that research is the tradeoff between how complete the knowledge of an inference system is, and how efficient it is (see for instance [LB 85]). Pure logic systems, for instance, are fairly expressive but intractable. Tractability is bought by resorting to simpler world views (e.g. a "closed world" view in which all you know about the world at a given time is all there is).

A number of researchers are now trying to find a way out from this impasse by giving up on the notion of having symbolic representations and inference systems that operate on them (see for instance [Agre 88]). Instead of doing symbol crunching inside the "head", they try to establish a continuous interaction between the agent and the physical world. The problem becomes too hard, so they argue, if you abstract stimuli from the world to symbolic representations, and try to solve the entire problem within the "head". By performing small actions and see how the world reacts, by engaging in a play of action and counteraction with the environ-

ment, agents get along without having to compute beforehand a complete plan of action.

Design and verification of integrated circuits, however, does have to evolve "in the head". It necessarily involves symbol manipulation and the engineering tradeoffs that go with it. One tradeoff, as mentioned before, is between completeness (or competence) and efficiency. Not that in every application completeness and efficiency can't be combined. There are some domains in which AI researchers succeeded in crafting competent and efficient systems. Symbolic algebra is one such domain. The price for those successes was hard work. Only by spending enough time analysing the application domain and building a program that was carefully geared to it, did AI workers end up with complete and efficient systems.

1.2 Formalisms for Circuit Verification

How do we set off and build a verification system, that takes a circuit and a description of the intended behavior, and decides whether the circuit is correct or not? The following questions have to be addressed:

- What is circuit structure, and how do you represent it?
- What is circuit behavior, and how do you represent it?
- What is the relation between structure and behavior, and how do you represent that?
- How do you verify a design with those formalisms? How efficient and correct is each approach?

An analogy that I have found very productive is one between the relation computer program - computational process and the relation circuit structure - circuit behavior. A program is a syntactic entity, a sequence of strings that is ordered in some way. It's a symbolic description of a computational process, which takes in certain inputs, performs a number of operations and produces an output. The

process evolves when the program is interpreted on a computer. A circuit structure is like a program: it's a syntactic configuration of symbols, a two dimensional one this time. A circuit behavior is a computational process that evolves when the physical circuit is 'interpreted' by nature, or when the symbolic version is handed to a simulator.

Computer Science has developed compact and precise formalisms for expressing the relation between a program text and the computational process that it stands for. It has formal grammars to express the rules according to which correct programs are configured. It has formalisms to describe processes. And it has semantic formalisms to relate parsed programs with formal process descriptions.

One method for describing the meaning of a program is the *operational semantics* method. An operational definition of a programming language L consists of two parts: a specification of an abstract machine M, and a function which transforms programs in L to programs for M. The meaning of a program is taken to be the result of interpreting the transformed program on the abstract machine M. There is no direct representation of the meaning of a program as such. You just have formal rules to determine the result of executing a program on some input.

A second method is *denotational semantics.* A denotational definition of a programming language L maps programs in L to mathematical functions representing a computational process. More specifically, it maps a parse tree (the result of parsing the program text) to a process description by recursively mapping the subtrees of the top node, and combining the results in a way that is appropriate for how the top expression is composed of the subexpressions. Denotational semantics gives a direct representation of the meaning of a program, and it has a functional formalism to derive such a representation from a program text.

A third method is *axiomatic semantics.* An axiomatic semantics of a programming language consists of a language to express properties of processes, and inference rules to derive statements in that language from portions of program text. Again, there is no direct representation of the meaning of a program. Instead, properties of the process are stated, like: "After execution of this program text, the values of x, y and n are such that $x = y^n$."

Axiomatic semantics allows you to prove that a program meets some specification (e.g. that $x = y^n$ after the program has been executed). Denotational semantics only tells you what processes are performed, not what the outcome is. From the point of view of program verification, the axiomatic formalism is clearly the more expressive one. On the other hand, a logic inference system performing the actual verification, is less efficient than the mappings in denotational semantics, which go straight from premises to conclusions.

All three formalisms can be applied to circuit verification. Operational semantics, which uses a new, better understood interpreter, translates to circuit simulation, which uses mathematical machinery as a reference point for circuit behavior. The other two approaches correspond to formal verification techniques. Axiomatic semantics has already been tried out as a vehicle for circuit verification (see for instance [CGM 87] and [HD 86]). Applying denotational semantics to verification is the central idea of my work.

1.3 Previous Work on MOS VLSI Verification

This thesis concerns the verification of digital MOS circuits. The primitive building blocks in these circuits are transistors, not boolean gates. Transistors have a W/L ratio and their nodes have a capacitance. MOS designers often rely on these features to establish certain computations. W/L effects and charge sharing between nodes can also corrupt the behavior that the designer intended to establish. Other features include capacitances between nodes of a transistor and between wires, and resistances of wires, but these features are usually less critical to correct operation.

For the purpose of efficient analysis of digital MOS circuits, taking just the most crucial effects into account, Randy Bryant proposed a switch level model, which considers transistors as switches with a certain conductance and certain capacitances at their nodes. This model is a starting point for serious work on verification of MOS VLSI circuits. Simpler models fail to give an account of crucial effects on the circuit. Researchers who started from this model to design verification systems for arbitrary circuits are Ivo Bolsens (who built DIALOG), Daniel Weise (who built

Silica Pithecus), Rick Spickelmier (who built Critic), Randy Bryant (who built MOSSYM), and a group of workers who built logic inference systems for circuit verification. Below is a brief review of the work of these researchers. A more in depth discussion follows in chapter 3, where I develop a framework for looking at different verification systems.

Ivo Bolsens' DIALOG [Bolsens 88] is based on a formal theory about what it means for a circuit to be correct. The theory starts with careful definitions of structural and behavioral concepts, goes on with axioms that define the various aspects of correct behavior, and derives theorems stating sufficient conditions for correct behavior. DIALOG checks whether these conditions hold in a given design. DIALOG's basic algorithms are generic over all circuit styles. They know how a transistor works, and figure out how a circuit behaves from there. To speed up the program, heuristics are included to recognize portions in a familiar design style (e.g. static complementary logic), so that further analysis of these portions is unnecessary. DIALOG doesn't verify whether the circuit conforms with a behavioral specification. It only checks for electrical bugs (e.g. charge sharing, W/L bugs, sneak paths, races). It does derive an internal boolean level representation which can be used for logic simulation.

DIALOG's weak spot is the brute force nature of its basic algorithm. It partitions the circuits into certain subcircuits (DCN's, discussed in chapter 2), and for each subcircuit, it sees what happens for all the combinations of input signals, which takes an exponential amount of time. What accounts for DIALOG's weak spot, also accounts for its strength. Thanks to the generic nature of the algorithm, it handles all types of circuits correctly.

Daniel Weise's Silica Pithecus [Weise 86] is based on the notion of abstraction from the circuit level to behavioral levels of representation. It abstracts the circuit description to a logic level representation, makes sure that the abstraction is valid, and matches the logic representation with a behavioral specification supplied by the designer. Although different, at first inspection, from Ivo Bolsens' approach, it has the same exponential algorithm buried in it. Silica Pithecus' position with regard to the tradeoff between competence and performance is thus similar to the one of

DIALOG.

It does gain efficiency, however, by taking advantage of hierarchy. Subcircuits are matched with subbehaviors, and subbehaviors are composed the same way the subcircuits are composed. To make this approach work, Daniel Weise relied on the notion of constraints. For a subcircuit to have the behavior of the corresponding logic description, certain constraints have to be satisfied. At the next level in the hierarchy, along with composing the subbehaviors, Silica Pithecus checks whether the constraints are satisfied. If it can't verify them, it passes them on to the next level. Eventually, a number of contraints pops out from the top level to the user.

Hierarchy helps to alleviate the exponential blowup problem. If a DCN (a subcircuit for which all combinations of inputs are considered) is broken up in the hierarchy, the run time is not exponential in the total number of inputs any more. There is no guarantee, however, that all large DCN's are broken up like that.

Rick Spickelmier's Critic [Spickelmier 88] is quite different from the other verification systems. It doesn't have a unified notion of design correcness. It has a collection of error configurations that it knows of, and it can check whether one of them occurs in a circuit, but it can't tell whether a design is correct or not. It doesn't know all error configurations that can possibly occur.

Randy Bryant's MOSSYM is not quite the same as a verification system either. It's a symbolic simulator: one that can take symbols as inputs and produce symbolic expressions as outputs. In principle, the system could be used as a verification system by giving symbols for all inputs, but the delay model is to weak to handle static feedback loops and circuits with races, and the system is as least as inefficient as the other systems.

A number of workers have built logic inference systems for circuit verification (see for instance [CGM 87] and [HD 86]). They use a higher order logic to describe the structure of a particular circuit, the behavior that it should have, and the relation between structure and behavior in general. From these premises, their systems try to prove that the design is correct. The programs suffer from the lack of performance that characterizes all logic inference systems, but they give precise answers to the question whether a design is correct or not.

1.4 My work

My work consists of two parts.

1. An attempt at getting fundamental insight in circuit verification, which characterizes existing verification systems, and points to a new, efficient verification method, based on syntax and denotational semantics.

   I developed a framework for understanding different approaches to verification, borrowing semantic formalisms from Computer Science, and drawing from experiences of other researchers in switch level simulation and verification. The framework rests on a clear uderstanding of what circuit structure and circuit behavior are, how they are represented, and how the relation between them can be expressed. The framework reveals the advantages and disadvantages of existing verification strategies, and points to a new method which is more efficient but also more conservative than other methods.

2. An implementation of the new method as a program, called Semanticist, and an analysis of its performance, competence and practicality as a CAD tool.

   Semanticist is a Lisp program that verifies combinatorial CMOS circuits. It verifies whether a given circuit displays a given boolean functionality.

   Semanticist is based on the denotational semantics approach to verification. It sits on top of Cyrus Bamji's GRASP program [Bamji 89], which takes a circuit grammar and produces a corresponding parser. Semanticist reads a circuit description, calls GRASP to parse the circuit, gets back a parse tree from GRASP, maps the parse tree to an internal behavioral description, and matches it with the behavioral description that the user supplied. The user supplied behavioral description consists of Lisp functions that can be executed on the Lisp interpreter.

   Cyrus Bamji used GRASP to verify whether a design adheres to a certain design style. He implemented a grammar for NORA as an example. I wrote a grammar which is intended to be as tolerant as possible. My grammar doesn't

have to incorporate all correctness constraints. Some constraints can be handled by the denotational semantics functions, others can be dealt with once the internal behavioral description has been built up. Cyrus Bamji verified circuit designs that rely on structural well-formedness rules for their correctness. I wanted to verify arbitrary circuits whose behavior can be understood with a simple switch-level model.

Semanticist has efficient algorithms, but it rejects some circuits that are correct. The difficult part in the design of the program was to come up with a circuit grammar that covers as many correct circuits as possible. The current grammar covers most of the circuits in [Weste 85]. Further extension of the range space can be achieved with more flexible grammar formalisms. GRASP's grammar formalism is based on context free graph grammars. Further work is needed to arrive at more flexible formalisms while maintaining the efficiency properties of GRASP.

Semanticist can be used incrementally. Subcircuits at the bottom of the circuit hierarchy can be verified first, higher level circuits next, and so on. It provides useful guidance in case of design errors. The behavioral representation that it matches with, extends all the way to the register transfer level and the system level.

1.5 Overview of Subsequent Chapters

The first part of this thesis, comprising chapters 2 and 3, establishes a framework to understand existing verification systems for MOS VLSI. Chapter 2 discusses Bryant's switch level model and the notion of design correctness based on that model. It recapitulates work that I did earlier in the field of circuit verification [VAVO 87]. From a precise formulation of what it means for a design to be correct, sufficient conditions for correctness are derived. These conditions formed the theoretical foundation for DIALOG, and they do so for my own verification system as well.

Chapter 3 presents a spectrum of verification approaches, situated in a framework that draws from the results of the previous chapter and from theoretical Computer Science. Existing verification systems which prove design correctness are shown to be at the inefficient side with regard to the efficiency - completeness tradeoff, and a new method at the efficient side is contrasted with them.

The second part of the thesis elaborates on the implementation of the denotational semantics approach. Chapter 4 presents the circuit grammar that I wrote. It starts out with an informal enumeration of what combinatorial CMOS circuits can look like, and discusses an attempt to implement it in Cyrus Bamji's grammar formalism.

Chapter 5 explains how the denotational semantics approach is implemented. It presents the overall architecture of Semanticist, elaborates on the major parts of it, and shows an example verification run.

Chapter 6 presents the results of my research in terms of software. It discusses Semanticist's performance, competence and practicality as a CAD tool.

Chapter 7 concludes with further work that has to be done.

Chapter 2

Design Correctness

In the beginning of the 1980's, Randy Bryant proposed a switch level model for analyzing the digital behavior of MOS circuits [Bryant 84]. The model was intended as a basis for switch level simulators which approach the speed of gate level simulators while maintaining much of the accuracy of simulators based on a detailed electrical model. It captures the phenomena that are most relevant to the logical behavior of MOS circuits. The model is now widely in use as a basis for CAD tools. It also forms the basis for formal verification systems for MOS circuits.

This chapter presents the switch level model that my verification system is based on. It differs only in minor aspects from Bryant's original model. I also discuss the notion of design correctness based on the model, and present sufficient conditions for correctness. The purpose of this chapter is twofold. First, it presents the "raw material" that has to go in a verification system. It introduces the knowledge that, in one form or another, has to be present. Different representation schemes can later be tested for how accurate they express the knowledge, and how efficient the resulting verification strategy is. Second, it qualifies my verification system and other systems that are based on similar models. If the logical behavior of a circuit is corrupted by effects that are not accounted for by the model, the verification system returns a wrong answer.

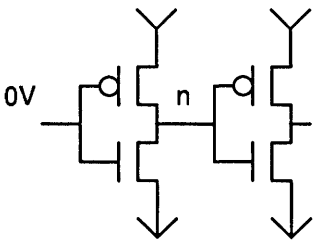

Figure 2.1: Node n has a 1 value.

2.1 A Switch Level Model

A switch level model consists of an enumeration of states that nodes and transistors can be in, and statements about causal relations between states of nodes and states of transistors. My model makes statements about states of transistor paths instead of transistors for reasons that will become clear later.

States of nodes and transistor paths

In my model, as in other models, there are three possible states for a node: it can have a 1 value, a 0 value, or an X value. A node has a 1 value if it is logically interpreted as a 1, and if its voltage is high enough to block currents that would be blocked by a perfect *Vdd* voltage. Analogously, A node has a 0 value if it is logically interpreted as a 0, and if its voltage is low enough to block currents that would be blocked by a perfect *Gnd* voltage. A node has an X value if it has neither a 1 nor a 0 value.

Some examples clarify these definitions. Node n in figure 2.1 has a 1 value. It is interpreted as a 1 by the next stage, and it blocks the current in the pMOS transistor it controls. Node o in figure 2.2 doesn't have a 1 value. Because of the voltage drop over the nMOS transistor, it doesn't block the current in the pMOS transistor it controls. Node p in figure 2.3 on the other hand, does have 1 value. In spite of the voltage drop it suffered, it still blocks the current in the pMOS transistor it controls, because source s also suffered a threshold drop.

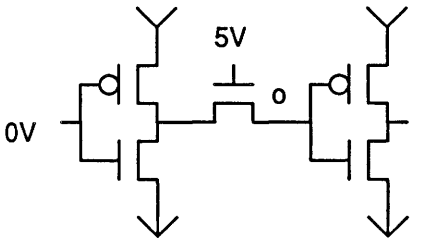

Figure 2.2: Node o doesn't have a 1 value.

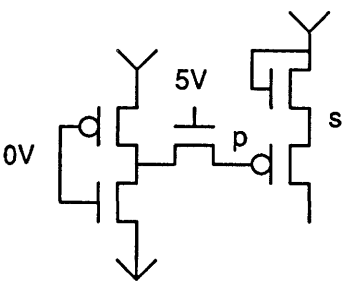

Figure 2.3: Node p has a 1 value.

Requiring 1 and 0 levels to be extreme enough so as to block currents has two benefits. First, it avoids static currents that increase the power dissipation. Second, when the node controls a pass transistor, it may be crucial to the correct operation of the circuit that the pass transistor is not conducting.

Transistor paths are in a "closed", "open", or "unknown" state. A "closed" state is a conducting state, being "open" is being non conducting, and being in an "unknown" state means that it is not known whether the path is conducting or not.

Causal relations

The state of a transistor path depends on the state of its input nodes in my model. Physically, it depends both on the source voltage and gate voltage whether a transistor is conducting, not on the gate voltage alone. By making statements about transistor paths between certain nodes (rather than about transistors in general), we will try to get around this problem.

A path is said to be "open" if it has at least one nMOS transistor with a 0 value at its gate, or if it has at least one pMOS transistor with a 1 value at its gate. This statement poses no problems. Physically, there is indeed no current through the path, unless you take subthreshold currents into consideration, or unless you take account of the possibility that the source of a pMOS transistor can get a voltage higher than Vdd by capacitive feedthrough, and that the source of an nMOS transistor can get a voltage lower than Gnd by the same effect.

A path between Gnd and a node n is said to be "closed" when all nMOS transistors in it have a 1 value and all pMOS transistors a 0 value. The path will physically conduct until n has been set to $0Volt$ (when the path has only nMOS transistors) or $0Volt + V_T$ when there are pMOS transistors in it. Analogously, a path beween Vdd and a node n is "closed" when all nMOS transistors in it have a 1 value and all pMOS transistors a 0 value.

The state of a node depends on the states of the paths that connect it to Vdd and Gnd, and on the previous state if the node is isolated. Whether a closed path between a node n and Vdd or Gnd can impose a 1 or 0 value on n depends on the types of the transistors in the path and on whether there is compensation for a

threshold drop in the path that n controls. For example, a closed path between Vdd and n which contains an nMOS transistor, can only impose a 1 value if the next stage is compensated for a threshold drop, like in figure 2.3 . In the following paragraphs we ignore this complication for the sake of easy readability.. A closed path between Vdd and n is supposed to contain only pMOS transistors. The statements we make remain valid for the case of paths with nMOS transistors in it, if there is compensation for threshold drops in the next stage.

A node n gets a 0 value in either of the following cases.

1. When all of the following conditions hold:

   - There is a closed path between n and Gnd.
   - All paths between n and Vdd are open.

   Figure 2.4 gives an example of this situation.

2. When all of the following conditions hold:

   - There is a closed path between n and Gnd.
   - All paths between n and Vdd that are not open, have, all together, a total W/L value smaller than some fraction f of the total W/L value of the closed path between n and Gnd. The value of f has to be determined for the particular fabrication process that is used.

   Figure 2.5 gives an example of such a case. Node n is initially at $5V$ and the output at $0V$. When the input changes to $5V$, n gets a 0 value if the W/L of $T1$ is small enough with respect to the W/L's of $T2$ and $T3$.

3. When all of the following conditions hold:

   - All paths to Vdd and Gnd are open.
   - n held a 0 value before.
   - The capacitance of n is big enough to overcome charge sharing with nodes from which it isn't isolated.

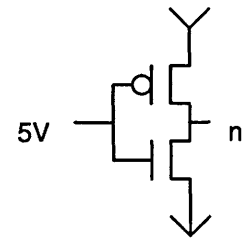

Figure 2.4: Node n gets a 0 value.

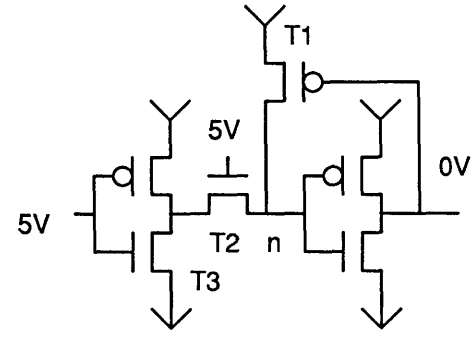

Figure 2.5: Node n gets a 0 value if the W/L of $T1$ is small enough with respect to the ones of $T2$ and $T3$.

An example of this situation can be seen in figure 2.6 . Node n gets a 0 value if its capacitance is high enough to overcome charge sharing with n'.

A symmetrical statement applies for getting a 1 value.

Comparison with other switch level models

This model differs only slightly from ones that other researchers have used. The model underlying Bryant's switch level simulator MOSSIM [Bryant 84], and later MOS analysis systems of his [Bryant 85,Bryant 87], has the following differences with respect to the one presented above. First, it has discrete W/L values and capacitances. This is essential to the algorithms Bryant uses: he iterates over all possible values that W/L's and capacitances can take. Second, Bryant's model

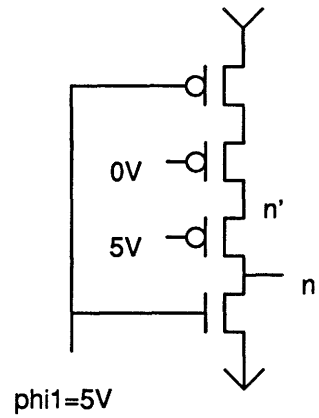

Figure 2.6: Node n gets a 0 value if its capacitance is high enough to overcome charge sharing with n'.

doesn't account for threshold drops over transistors. And third, the state of a transistor depends only on the state of its gate in Bryant's model. The statements above about the state of transistor paths are more realistic, but the net result in interpreting practical circuits is the same.

Bolsens' model [Bolsens 88] is exactly the same as the one presented here. Weise's model [Weise 86], on the other hand, is somewhat more complex. He models a transistor as a resistor in series with a threshold device (i.e. a voltage controlled switch which opens if the absolute voltage at the drain passes some threshold). The resistance depends on the W/L and the type of the transistor, and both the resistance and the threshold voltage depend on the gate voltage. The net result in interpreting practical circuits, is not different, to my knowledge, from interpretations according to the model of Bolsens of me.

2.2 Design Correctness

Both Weise and Bolsen start from a formal statement of what it means for a design to be correct [Weise 86,Bolsens 88]. Bolsens' correctness definition comes from work that I did together with Cris Van Overloop [VAVO 87]. It states what it means for

a circuit to be electrically correct (i.e. to have no circuit level bugs). Weise's definition has a different scope. It defines correctness for a design consisting of a circuit description and a description of the intended behavior. It subsumes Bolsens' definition.

Weise defines a multilevel design to be correct if, given some constraints on how the circuit is used, its abstracted behavior matches with the user supplied behavioral description. The key notion in this definition is abstraction. A circuit is correct if it can be abstracted to a logic level representation which matches with the one supplied by the user. The circuit is abstracted to a logic level representation within a certain context which is expressed as constraints that are guaranteed by the user to be satisfied (e.g. that exclusively one of a set of input signals is logically one, or that an input signal has suffered no voltage drop). Only in a certain context does a circuit show a particular behavior.

The question: "When is a circuit correct?" is thus reduced to: "What is the abstraction function from circuit to behavior?". The answer to that question is spread out over a number of chapters in Weise's Ph.D. thesis. Bolsens' work deals with the first component of Weise's notion of correctness. He gives a compact answer to the question: "When can a circuit be abstracted to a logic level representation?". He is not concerned with whether the abstracted behavior corresponds with the intended one.

Bolsens' correctness definition assumes a level-sensitive design discipline. A circuit is level-sensitive when its functionality depends only on voltage levels, never on whether one signal has a longer delay than another. In order to ensure level-sensitive behavior, the circuit should be synchronous. In synchronous systems, feedback signals are held up by registers, and once all inputs to a combinatorial block have arrived, they are all passed to the combinatorial block at the same time. If feedback signals were passed to the combinatorial block as soon as they arrive, the ultimate output of the circuit would depend on the order in which feedback signals arrive, and the system would not be level-sensitive.

In the theory in [VAVO 87], synchronous circuits are partitioned into combinatorial blocks which are connected through "memory nodes" and "control nodes".

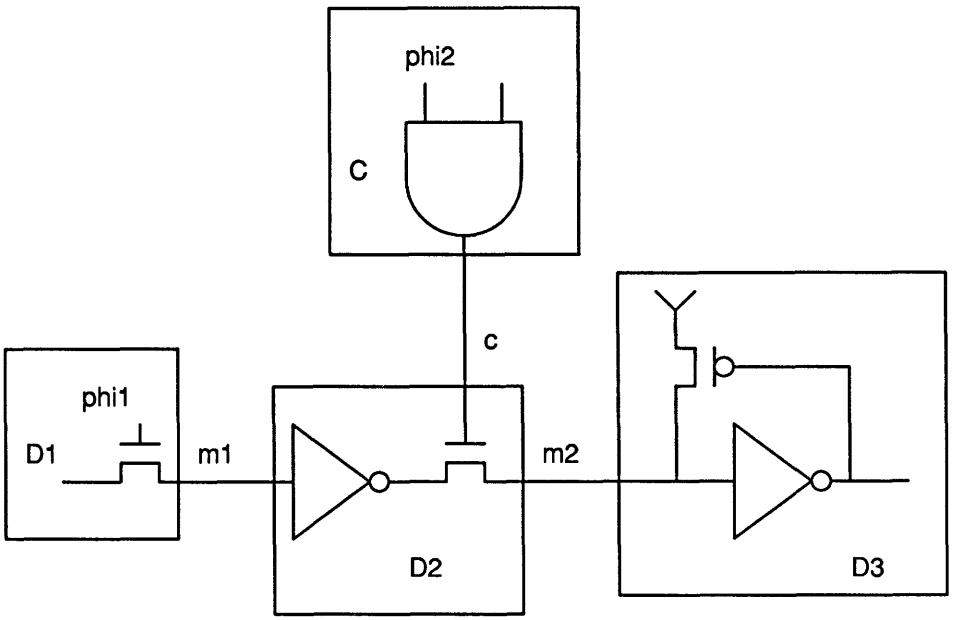

Figure 2.7: $m1$ and $m2$ are memory nodes; c is a control node; $D1$, $D2$, $D3$ and C are combinatorial blocks; $D1$, $D2$ and $D3$ are data blocks and C is a control block.

Memory nodes are particular nodes in the registers between combinatorial blocks, and control nodes are nodes that control when the registers release their information to the next combinatorial block.[1] Figure 2.7 gives a simple example. Nodes $m1$ and $m2$ are memory nodes, node c is a control node, and $D1$, $D2$, $D3$ and C are combinatorial blocks. Combinatorial blocks that end in control nodes are called control blocks (e.g. C in figure 2.7); ones that end in memory nodes are called data blocks (e.g. $D1$, $D2$ and $D3$ in figure 2.7).

Correct level-sensitive behavior means that correct digital levels are produced at the outputs of the combinatorial blocks, and that the time behavior of those blocks is such that delays don't influence the overall digital behavior. Formally, a circuit is said to have a *correct level-sensitive behavior* if its combinatorial blocks have a "correct steady state behavior" and a "correct transient behavior", and if its

[1] The presentation of definitions and theorems in this chapter is kept informal. Precise formulations can be found in [VAVO 87] or [Bolsens 88]

memory nodes have a "correct memory behavior". An informal definition of those concepts follows below.

Steady state behavior is the fictitious behavior of a combinatorial block when its inputs are kept constant for an infinite time. It is not concerned with whether the inputs are indeed kept constant for a long enough time to allow the outputs to reach a steady state. Neither is it concerned with how fast the outputs reach a steady state. Those concerns have to do with transient behavior.

A combinatorial block is said to have a *correct steady state behavior* if, for any combination of inputs to the block, the outputs evolve to steady states that represent a valid logical 1 or 0, and that are only functions of the inputs (not of a state of the circuit). In other words, correct steady state behavior means that the combinatorial block can be abstracted to a combinatorial boolean function.

Memory nodes alternate between phases in which their value is memorized and phases in which they get a new value. We call those phases memorization phases and computation phases respectively. A memory node is said to have a *correct memorization behavior* if it keeps its value during the memorization phase.

What it means for a combinatorial block to have a *Correct transient behavior* depends on whether it is a data block or a control block. For a datablock it means that the outputs get the appropriate steady state values before the end of their computation phases. What happens during the transition is not important. A control block has a correct transient behavior if the outputs have glitch-free transitions between two clock phases, and if their values remain constant during any one clock phase. Unlike data block outputs, control block outputs have to satisfy well-behavedness constraints at all times.

2.3 Sufficient Conditions for Design Correctness

In the previous section a design was defined to be correct if the circuit representation can be abstracted to a logic level representation, and if the abstracted logic level representation corresponds with the one supplied by the designer. The first component in the definition was further developed assuming a level-sensitive design

discipline. It was reduced to the requirements of correct steady state behavior, correct transient behavior and correct memorization behavior. In this section the notion of correct steady state behavior is further developed. Transient behavior and memorization behavior are beyond the scope of my verification system, which verifies only combinatorial circuit portions. Issues of transient behavior and memorization behavior show up only when verifying sequential systems. They translate to structural constraints (constraints on how to compose combinatorial blocks) and timing constraints (constraints on the speed of combinatorial logic for a given clock frequency). Timing constraints require a more advanced circuit model, but the structural constraints fit perfectly in my verification strategy, I believe. It's not demonstrated in this thesis, though.

The correct way to view a combinatorial circuit in the context of correct steady state behavior, is as a partitioning of DCN's. The concept of DCN's was introduced by Bryant [Bryant 81]. He called them "transistor groups". DCN's are maximal subnetworks with DC paths between all its nodes (DCN is short for DC Network). More formally, DCN's are the connected components of the graph with a vertex for each node and an edge between each pair of vertices corresponding to the source and drain of a transistors. (In these definitions Vdd and Gnd are thought of as being split so that they connect to only one transistor.) Figure 2.8 gives an example. $DCN1$, $DCN2$, $DCN4$ and $DCN5$ contain only one node. $DCN3$ is made up of transistors $T1$ through $T7$, and $DCN6$ contains $T8$ and $T9$.

Partitioning a circuit into DCN's is useful because bidirectional effects take place within a DCN whereas connections between DCN's carry unidirectional information (we ignore capacitive coupling between a gate of a transistor and either the source or the drain). A circuit can thus be reduced to a directed graph with DCN's as vertices (see figure 2.8). We can now state a first set of sufficient conditions.

For a combinatorial block to have a correct steady state behavior, it suffices that

1. *all composing DCN's have a correct steady state behavior;*

2. *there are no loops of DCN's, unless there is a DCN in the loop which has*

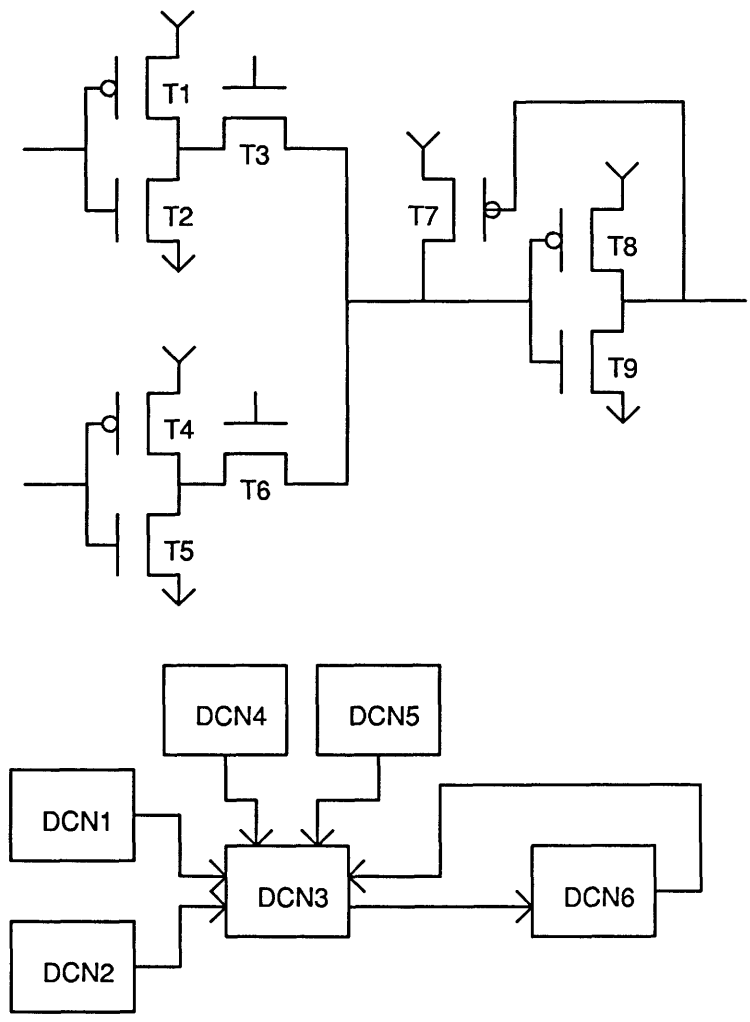

Figure 2.8: Partioning of a circuit into DCN's. $DCN1$, $DCN2$, $DCN4$ and $DCN5$ contain only one node. $DCN3$ is made up of transistors $T1$ through $T7$, and $DCN6$ contains $T8$ and $T9$.

external inputs that dominate over the ones that come from the loop.

Applied to figure 2.8 this means that all the DCN's need to have a correct steady state behavior, and that the output of $DCN3$ should be completely determined by the inputs from $DCN1$, $DCN2$, $DCN4$ and $DCN5$. The input from $DCN6$ is only allowed to reinforce the value that has already been selected by the other inputs. This requirement boils down to a W/L constraint. The W/L of $T7$ should be small enough with respect to the ones of $T2$, $T3$, $T5$ and $T6$.

Correct steady state behavior of a DCN can be further refined along the lines of the switch level model presented in section 1 of this chapter.

For a DCN to have a correct steady state behavior, it must be true that for every output, and **for every possible combination of inputs**, *one of the following conditions holds:*

1. *There is a connection to either Vdd or Gnd (not to both).*
2. *There is a connection to both Vdd and Gnd, but the W/L ratios are such that a valid output is obtained.*
3. *There is no connection to Vdd or Gnd, but the output was precharged during the previous clock phase, and its value is not corrupted by charge sharing.*

A straightforward algorithm for verifying DCN's in which all input combination are considered one after the other, has an exponential complexity in the number of inputs. The computation time blows up for large DCN's. All existing verification systems[2], however, have basic algorithms that work exactly along these lines. The new verification strategy, proposed in this thesis, avoids this trap by bringing to bear more knowledge about CMOS circuits than is contained in the switch level model. Instead of looking at how individual transistors work and building up the behavior of the circuit from there, the new verification system recognizes familiar

[2]I'm referring here to systems that reject all faulty circuits; I'm not including Critic.

circuit configurations like dynamic gates, complementary static gates, pass transistor logic, and so on, and goes on from there. Unlike Critic [Spickelmier 88], the new verification system is organized in such a way that all faulty circuits are rejected. The following chapter presents a more complete view of different verification strategies.

Chapter 3

A View on Verification Strategies

An integrated circuit, in its physical form, is a structure in silicon which performs a certain computational process in accordance with the laws of nature. A VLSI circuit, as a design representation, is a composition of symbols which, in the imagination of the designer, performs a computational process in accordance with some circuit model. The goal of verification is to make sure that a given circuit schematic has the right computational process associated with it. A fully automatic verification system is based on formal, symbolic laws for inferring processes from structures. Writing a good verification program comes down to finding accurate formalisms which allow efficient computations.

Formalisms for relating structures with computational processes constitute a familiar theme in Computer Science. Specifications of the semantics of programming languages are exactly such formalisms. They relate program texts, essentially structural compositions of character symbols, with the computational processes that they describe.

This chapter uses formalisms for describing the semantics of programming languages as a window on different strategies for circuit verification. It discusses existing verification methods within the framework that semantics definitions provide, and it introduces a new approach, based on denotational semantics. The first section reviews semantics formalisms, the second section translates them to the realm of VLSI design, the third section gives a review of existing verification systems, and the last section presents the denotational semantics approach to circuit verification.

3.1 Review of Semantics Formalisms

3.1.1 Operational Semantics

An operational semantics definition of a programming language L assigns a meaning to a program by reference to a formally specified machine M. It provides a formal definition of M and a formal definition of a function that transforms programs in L to programs that run on M . The meaning of a program in L is taken to be the evaluation history that M produces when it runs the transformation of the program. In some operational semantics definitions, M is defined to operate on programs in L directly. Those definitions consist of a specification of M only.

To clarify operational semantics specification, as well as other approaches to semantics that are discussed later on, we introduce the ultra simple language USL. USL has the following grammar:

$$\begin{aligned} \texttt{C} &::= \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid +1 \mid -1 \\ \texttt{I} &::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \\ \texttt{E} &::= \texttt{C} \mid \texttt{I} \mid \texttt{(lambda (I) E)} \mid (\texttt{E}_1\ \texttt{E}_2) \end{aligned}$$

Informally the language can be described as follows. USL has 5 constants (C): 1, 2, 3, +1 and -1. The constants +1 and -1 denote procedures that take a number and increment or decrement it. It has 3 identifiers (I): x, y and z. A valid expression (E) in USL is a constant, an identifier, a lambda expression or an application. A lambda expression (lambda (I) E) denotes a procedure that takes an identifier and evaluates the expression E with I bound to the identifier that it took in. An expression of the form $(E_1\ E_2)$ denotes an application of the procedure represented by E_1 to the value of E_2. An example of a valid expression in USL is the following:

$$\texttt{((lambda (x) (+1 x)) 2)}$$

The result of evaluating this expression is 3.

Here is an operational definition of the semantics of USL. It's one of the kind that only has a definition of a machine M which evaluates expressions in a proper way (programs in USL are not translated to an intermediate form). In order to define M, we introduce one more syntactic variable (one that represents values):

$$\texttt{V} ::= \texttt{C} \mid \texttt{(lambda (I) E)}$$

M is defined by three axioms and two inference rules. The axioms are as follows:

$$\begin{aligned} \texttt{((lambda (I) E)V)} &\stackrel{red}{\Longrightarrow} \texttt{[V/I]E} \\ \texttt{(+1 V)} &\stackrel{red}{\Longrightarrow} \texttt{V + 1} \\ \texttt{(-1 V)} &\stackrel{red}{\Longrightarrow} \texttt{V - 1} \end{aligned}$$

These axioms prescribe reductions of expressions in which all subexpressions are values. The first axiom says that a lambda expression (lambda (I) E) should be reduced to its body E in which V is substituted for I. The two other axioms are obvious. By "V+1" I mean 2 if V is 1, 3 if V is 2, and "error" otherwise. Likewise for V-1.

The inference rules prescribe reductions of expressions in which not all subexpressions are values. They allow reducing subexpressions to values, so that, afterwards, axioms can be applied for further reductions. The inference rules are as follows:

$$\begin{aligned} &\textit{If} && \texttt{E}_1 \stackrel{red}{\Longrightarrow} \texttt{E}'_1 \\ &\textit{then} && \texttt{(E}_1\ \texttt{E}_2\texttt{)} \stackrel{red}{\Longrightarrow} \texttt{(E}'_1\ \texttt{E}_2\texttt{)} \end{aligned}$$

$$\begin{aligned} &\textit{If} && \texttt{E}_2 \stackrel{red}{\Longrightarrow} \texttt{E}'_2 \\ &\textit{then} && \texttt{(V}_1\ \texttt{E}_2\texttt{)} \stackrel{red}{\Longrightarrow} \texttt{(V}_1\ \texttt{E}'_2\texttt{)} \end{aligned}$$

Together, the axioms and inference rules define an abstract machine M that interprets expressions as follows. If the expression is not an application, nothing has to be done. If it is, then evaluate the procedure subexpression first and the argument subexpression next, and apply the value of the procedure to the argument value.

Operational semantics was introduced the 1960's by Peter Landin [Landin 64]. An example of an operational semantics definition of a programming language can be found in [Gifford 87]. [1]

[1] Most of the references to the literature in this section come from "Introduction to the Literature", MIT, 6-821 — Concepts in Modern Programming Languages, Handout #59, December 4, 1986.

3.1.2 Denotational Semantics

The denotational semantics method maps a program directly to a representation of the computational process that it describes. Whereas operational semantics specifies the meaning of a program by reference to a machine that interprets it in a well-defined way, denotational semantics produces a representation of the meaning of the program directly. It doesn't need a machine to say things about computational processes. The processes are described as such.

A denotational semantics definition of a programming language consists of three parts: a description of the syntax, a description of the semantic world and mappings from syntactic entities to objects in the semantic world. To make this more concrete, we give a denotational definition of USL.

Syntax

The description of the syntax of USL can just be copied from above:

$$\begin{aligned} \texttt{C} &::= \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid +\texttt{1} \mid -\texttt{1} \\ \texttt{I} &::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \\ \texttt{E} &::= \texttt{C} \mid \texttt{I} \mid \texttt{(lambda (I) E)} \mid (\texttt{E}_1\ \texttt{E}_2) \end{aligned}$$

Formally, a grammar is a 4-tuple consisting of a set of variables, a set of terminal symbols , a set of production rules and a top variable. The variables of USL's grammar are **C** , **I** and **E**. The terminal symbols are **1**, **2**, **3**, **+1**, **-1**, **x**, **y**, **z**, (,), and **lambda**. Each line in the grammar is a compression of several production rules. For each alternative in the right hand side of each line, there is a production rule. An example of a single production rule is:

$$\texttt{C} ::= \texttt{1}$$

The top variable in the grammar above is **E**. The set of syntactically correct programs in USL is the set of terminal strings that can be produced from the top variable **E** by applying production rules. The tree that shows the productions from the top variable to the terminal symbols is called a parse tree. Parsing a string of terminal symbols is building up a parse tree on top of it.

The semantic world

The semantic world consists of semantic domains and processes that are expressed as mappings between semantic domains. Semantic domains can be understood as sets, although they are a bit more complicated in reality (we leave all subtleties aside in this discussion). Here are the semantic domains that are relevant to USL.

$$
\begin{aligned}
e &\in \mathit{Expressible\_value} = \mathit{Natural\_number} + \mathit{Procedure} \\
p &\in \mathit{Procedure} = \mathit{Expressible\_value} \rightarrow \mathit{Expressible\_value} \\
u &\in \mathit{Environment} = \texttt{Identifier} \rightarrow \mathit{Expressible\_value}
\end{aligned}
$$

There is a domain *Expressible_value* which is the (disjoint) union of *Natural_number* and *Procedure*. We don't want to get into the details of what a disjoint union is and why it is needed (they are given in [Schmidt 86]). Just take it that an expressible value is a natural number or a procedure. Elements of *Expressible_value* are usually written as *e*. *Procedure* is the set of all mappings from *Expressible_value* to *Expressible_value*. It's not exactly that but, again, we don't want to get into subtleties. *Environment* is the set of all mappings from identifiers (these are syntactic objects) to *Expressible_value*. Intuitively, an environment holds bindings between identifiers and expressible values.

Computational processes are described as mappings over semantic domains. These mappings are represented by expressions in the lambda calculus (see [Barendregt 84] for more on the lambda calculus). A lambda expression is of the form:

$$\lambda\langle variable\rangle.\langle expression\rangle$$

expression is a variable, a lambda expression, or an application of an expression to an other expression, written as:

$$(\langle expression1\rangle\langle expression2\rangle)$$

You will notice that expressions in the lambda calculus are not a lot different from expressions in USL. There two reasons for this. First, USL is a subset of Scheme [RCA* 86] (a Lisp dialect) which is rooted in the lambda calculus. And

second, it's too simple to have features that could be semantically ambiguous. The exercise of giving a denotational definition of USL is thus somewhat artificial. Our concern here is with the denotational semantics formalism itself, of course, not with USL.

Valuation functions

The third component of a denotational definition of a programming language consists of valuation functions which map syntactic entities to lambda expressions. There is a valuation function for each variable in the grammar:

$$\begin{aligned} \mathcal{C} &: \texttt{C} \rightarrow \mathit{Expressible\_value} \\ \mathcal{I} &: \texttt{I} \rightarrow (\mathit{Environment} \rightarrow \mathit{Expressible\_value}) \\ \mathcal{E} &: \texttt{E} \rightarrow (\mathit{Environment} \rightarrow \mathit{Expressible\_value}) \end{aligned}$$

The valuation function $\mathcal{C}$ takes a constant and returns an expressible value. $\mathcal{I}$ takes an identifier I and returns a function from environments to expressible values. Given an environment u, this function will return the value that I is bound to in u. $\mathcal{E}$ takes an expression and returns a function from environments to expressible values.

Each valuation function is defined for each production rule that is associated with it in the grammar. Below, we define $\mathcal{E}$ for each production rule that has E as a left hand side. The definitions for $\mathcal{C}$ and $\mathcal{I}$ are obvious.

$$\begin{aligned} \mathcal{E}[[\texttt{C}]] &= \lambda u.\mathcal{C}[[\texttt{C}]] \\ \mathcal{E}[[\texttt{I}]] &= \lambda u.(\mathcal{I}[[\texttt{I}]]\ u) \\ \mathcal{E}[[(\texttt{lambda (I) E})]] &= \lambda u.\lambda e.(\mathcal{E}[[\texttt{E}]]\ u[e/\texttt{I}]) \\ \mathcal{E}[[(\texttt{E}_1\ \texttt{E}_2)]] &= \lambda u.((\mathcal{E}[[\texttt{E}_1]]\ u)\ (\mathcal{E}[[\texttt{E}_2]]\ u)) \end{aligned}$$

$\mathcal{E}$ applied to C returns a function that takes in an environment and returns the result of applying $\mathcal{C}$ to C. $\mathcal{E}$ applied to I returns a function that takes in an environment u and returns the result of applying $\mathcal{I}[[\texttt{I}]]$ to u. In other words, $\mathcal{E}$ applied to I returns a function that takes in an environment u and returns the value that I is bound to in u. $\mathcal{E}$ applied to (lambda (I) E) returns a function that takes in an environment u and returns a procedure from an expressible value e to the result

of applying $\mathcal{E}$[[E]] to the environment u extended with a binding of I to e. $\mathcal{E}$ applied to (E_1 E_2) returns a function that takes in an environment u and returns the result of applying the value of E_1 in u to the value of E_2 in u.

These specifications suffice to map all syntactically valid expressions in USL to a mathematical function that represents a computational process. To valuate an expression E, parse it, take the top variable, recursively valuate its subexpressions, and combine the results in the appropriate way. The recursive nature of the grammar and the valuation functions allows an infinite number of expressions to be mapped according to a finite number of specifications.

Denotational semantics was developed by Dana Scott and Christopher Strachey in the 1970's at Oxford. A comprehensive treatment of the subject can be found in [Stoy 77] or [Schmidt 86]. [Schmidt 86] also contains references to denotational descriptions of real languages.

3.1.3 Axiomatic Semantics

An axiomatic semantics description of a programming language consists of a logic language for expressing properties of computational processes, and axioms, one for each kind of statement in the programming language, which describe the effect of executing a statement. Relying on the axioms for each individual statement, it is possible to derive statements about the execution of a whole portion of program text. An example of such a statement is: "After execution of this portion of program text, the values of x, y and z are such that $z = xy$."

Again, an example helps in getting a more concrete picture of the axiomatic method. We use the following notation to make assertions about portions of program text:

$$\{P\}S\{q\}$$

The meaning of this is as follows: "If P is true before the execution of S is begun, and if S terminates normally, then Q will be true afterwards." In this notation, we can write down axioms about sequences of statements, assignment statements and while-loops:

$$\begin{array}{ll} \textit{If} & \{P\}\ S_1\ \{Q\} \wedge \{Q\}\ S_2\ \{R\} \\ \textit{then} & \{P\}\ S_1; S_2\ \{R\} \end{array}$$

$$\{P\}\ x := E\ \{[E/x]P\}$$

$$\begin{array}{ll} \textit{If} & \{P \wedge B\}\ S\ \{P\} \\ \textit{then} & \{P\}\ \textit{while}(B)\ (S)\ \{P \wedge \neg B\} \end{array}$$

If P leads to Q after execution of S_1, and Q leads to R after execution of S_2, then P leads to R after execution of the sequence $S_1; S_2$. If P is true before executing the assignment $x := E$, then P remains true afterwards if E is substituted for x in P. If P is an invariant for S, as long as B holds before executing S, then executing $\textit{while}(B)\ (S)$ establishes $P \wedge \neg B$.

What can be proven from these axioms is illustrated by the following annotated program:

$$\begin{array}{l} i := 0; \\ z := 0; \\ \{\mathbf{z = iy}\} \\ \textit{while}(i < x)\ (\\ \quad i := i + 1; \\ \quad z := z + y; \\ ) \\ \{\mathbf{z = xy}\} \end{array}$$

$(z = iy)$ is established by the initializing statements, and it is an invariant for the body of the while loop. After executing the while-loop, it is true that $(z = iy)$ and $(i = x)$, and therefore it is true that $(z = xy)$.

As this example illustrates, axiomatic semantics provides the power to prove that a program meets some specification. Whereas denotational semantics derives a description of the computational process *itself*, axiomatic semantics produces assertions *about* the computational process (for instance assertions about the overall

input output relation). In the context of program verification, axiomatic semantics is thus the superior formalism what expressiveness is concerned. On the other hand, when it comes to performance of automatic verification systems, denotational semantics is by far the most attractive. The valuation functions go straight from program texts to descriptions of processes. The proof rules of axiomatic semantics do no better then specifying a search tree where each branch might or might not have interesting assertions on it. Finding the relevant assertions involves extensive search.

Axiomatic semantics was introduced by Hoare in the late 1960's [Hoare 69]. An example of the use of axiomatic semantics is Hoare and Wirth's axiomatic definition of Pascal [HW 73]. Dijkstra and Gries used the axiomatic formalism as a basis for a programming methodology in which a correctness proof is developed at the same time the program is being written [Dijkstra 76,Gries 81].

3.2 A Framework for Understanding Circuit Verification

The theory of programming languages distinguishes syntactic program texts from the computational processes that they describe, and it presents formalisms for relating the two. By translating these ideas to VLSI design, we can set up a framework for getting fundamental insight in circuit verification. In the context of fully automatic verification, we will be exclusively interested in denotational and axiomatic semantics. Operational semantics can tell you how a program works on a particular input, by reference to an abstract machine that interprets the program in a well-defined way. It doesn't yield a description of a computational process as such. Operational semantics matches with circuit simulation. A simulator is like the abstract machine. It tells you how the circuit responds on certain inputs. It doesn't provide a description of how the circuit responds on any input.

In this section we set up a framework for understandings circuit verification that is based on denotational and axiomatic semantics. The framework is established in

the form of answers to the following questions:

1. What is circuit structure, and how can it be represented?
2. What is circuit behavior, and how can it be represented?
3. How can the relation between structure and behavior be represented, and to what kind of formal reasoning does this representation lead?
4. What is the relation between questions 1, 2 and 3?

1. What is circuit structure, and how can it be represented?

Circuit structure is information about *how modules are interconnected.* A structural representation describes a circuit as a network of modules. Structure is a static concept. It tells nothing about sequences of events that can take place on a circuit.

A grammar can be used to describe the set of all possible circuit structures. One grammar allows just any interconnection of transistors. Other grammars put more constraints on which interconnections are valid and which not. If the grammar is explicitly represented, a circuit structure can be transformed into a parse tree. Parsing a circuit is "making sense of it". Once the circuit has been parsed, all the parts of it have been placed in familiar categories.

2. What is circuit behavior, and how can it be represented?

Circuit behavior is information about *how events are causally related.* Behavior is a dynamic concept. It expresses information of the kind: "If this happens, then that will happen". A behavioral representation has to capture the causal relations that underlie chains of events.

There are two ways of representing behavior. One way is to specify the overall input-output relation of the circuit, without saying how it is established. The other way is to give a detailed process description. Such a description gives not only the output signals that correspond with a given set of inputs, but also the algorithm according to which they are computed. The first kind of representation is declarative

(it declares what the input-output relation is without specifying how it is realized), the second one procedural (it specifies the process that establishes a certain input-output relation). The same duality occurs with respect to programming languages: PROLOG is an example of a declarative language, Lisp one of a procedural language.

3. How can the relation between structure and behavior be represented, and to what kind of formal reasoning does this representation lead?

One way to represent the relation between structure and behavior is by means of inference rules in a formal logic. This scheme corresponds to the axiomatic semantics approach. The kind of automatic reasoning that goes with this representation is *deductive reasoning.* The knowledge about the relation between structure and behavior is organized in small chunks (small reasoning steps, expressed as inference rules) with no clue as to how to put them together to infer the behavior of a given circuit. To infer the behavior, different sequences of reasoning steps have to be tried out, until a solution is arrived at. Deductive reasoning is a search process in a space of assertions which might or might not be of interest.

The other way to represent the relation between structure and behavior is with mappings from structural entities to behavioral descriptions (as in denotational semantics). The corresponding form of reasoning is *transformational reasoning.* A variant of the denotational semantics mappings which can still be classified under transformational reasoning are rewrite rules. Rewrite rules are like denotational mappings in that they go straight from structure to behavior, unlike logic inference rules which lead to search. The difference with denotational mappings is that they involve intermediate representations, whereas the mappings go from structure to behavior in one step.

Deductive reasoning and transformational reasoning are familiar concepts in the context of program synthesis. A deductive approach to program synthesis can be found in [MW 81]; a transformational approach is taken in [BD 77]. The notion of transformational reasoning is also in use in the field of digital systems [Johnson 84,Boute 86,Paillet 87], but so far it was never applied as low as the circuit level or the switch level.

4. What is the relation between questions 1, 2 and 3?

The relation between questions 2 and 3 is that deductive reasoning fits with declarative behavior representations and transformational reasoning with procedural ones. Deductive reasoning has the inferential power to prove that a circuit meets a declarative specification. Transformational reasoning maps the circuit to its behavioral mirror. It can't reason *about* the behavior.

Question 1 is related to question 3 in the sense that an explicit (and constraining enough) grammar is a prerequisite for transformational reasoning with direct mappings from circuits to behavior descriptions. If there is only an implicit grammar which allows any interconnection of transistors, rewrite rules with intermediate representations between structure and behavior are the only vehicle for transformational reasoning. With a grammar, a system can "make sense" of the circuit first, so that the subsequent transformation proceeds much more efficiently.

In sum, there are three fundamental approaches to circuit verification. The first one, deductive reasoning, is the least efficient but the most competent. The other extreme, transformational reasoning along denotational semantics mappings, is the most efficient but potentially the least competent: not only is it not able to prove that a circuit meets a declarative specification, but, because it imposes a grammar on circuits, it might reject some circuits that are correct. In between lies the transformational approach with rewrite rules. It has medium efficiency and medium competence. It can't prove that a circuit meets some specification, but it is less conservative than the denotational semantics approach.

3.3 Review of Existing Verification Systems

We are now in a good position to review existing verification systems. Some systems correspond exactly with one of the fundamental approaches mentioned above. Gordon's logic based approach [CGM 87] corresponds with deductive reasoning, and both Weise's Silica Pithecus [Weise 86] and Bolsens' DIALOG [Bolsens 88] can be categorized as transformational systems based on rewrite rules (DIALOG is actually a bit more complex than that; see further for more details). The remaining

strategy, transformational reasoning based on denotational semantics mappings, is the one that I have taken.

Other systems don't fit exactly in the framework because they aren't really verification systems. Bryant's MOSSYM [Bryant 85] is based on a simulation paradigm, and Spickelmier's Critic [Spickelmier 88] can criticize a circuit design, but can't tell whether a design is correct. The rest of this section elaborates on each of these systems.

3.3.1 Gordon's Logic Based System

A number of systems have been built that use formal logic as a vehicle for hardware verification. For most of them, the input is a gate level representation, or a representation on still a higher level (e.g. Barrow's VERIFY [Barrow 84], Hanna's Veritas [HD 86] and Hunt's theorem prover which verified the FM8501 microprocessor [Hunt 87]). To my knowledge, only the work of Gordon and his co-workers [CGM 87,JBG 85] deals with switch level representations.

Gordon's work is based on higher-order logic. Higher-order logic is an extension over the predicate calculus. It extends over the predicate calculus in three ways:

1. Functions and predicates can be quantified.

2. Functions and predicates can be the arguments and results of other functions and predicates.

3. Terms can be lambda expressions.

Higher-order logic is appropriate for describing the structure and the intended behavior of a circuit, and for expressing inference rules about the relation between structure and behavior. In principle, a theorem prover could fully automatically search for a correctness proof, but in practice only a guided theorem prover has been built [JBG 85]. The practicality of the logic approach to verification is illustrated by the following quote from [CGM 87], p 65:

> "Formal verification is very expensive using current theorem-proving technologies. Experts are needed to guide proof generating tools and typical proofs take months of work. In the short term it is likely that verification by formal proof will only be worthwhile for those systems whose failure would result in disasters such as loss of life, destruction of costly equipment, or recall of a mass produced product."

3.3.2 Weise's Silica Pithecus

Silica Pithecus can be categorized as a transformational system based on rewrite rules. It takes a straight path from premises to conclusion, but the path is long and tedious. The key reason for this is the lack of circuit syntax. It doesn't "make sense" of the circuit before starting with the transformations. The transformations are applied to an arbitrary interconnection of transistors.

Silica Pithecus verifies a design in three steps. First, the circuit representation is transformed into a series of "net behavior expressions". A net behavior expression for a node n is a conditional expression with clauses consisting of a predicate and a set of nodes to which n is connected if the predicate holds. For instance, node Out in figure 3.1 has the following net behavior expression:

$$\begin{aligned} Out_{net} = \lambda t. \quad & Pass_b(t) \wedge In_b(t) \rightarrow [Out\ I\ Vdd\ Gnd] \\ & Pass_b(t) \wedge \neg In_b(t) \rightarrow [Out\ Vdd\ I] \\ & \neg Pass_b(t) \rightarrow [Out] \end{aligned}$$

This expression says that the net that Out belongs to (i.e. the set of nodes that n is connected with) is a function of time as follows. If $Pass$ and In are logically 1, Out belongs to the net $[Out\ I\ Vdd\ Gnd]$; if $Pass$ is 1 and In 0, Out is connected to I and Vdd, and if $Pass$ is 0, Out is isolated.

Net behavior expressions are generated for each "interesting" node. "Interesting" nodes are outputs, nodes that are inputs to a transistor, and nodes that carry state information. The net behavior expressions are obtained by merging net behavior expressions of subcircuits, starting with the ones for the gate, source and drain of a single transistor.

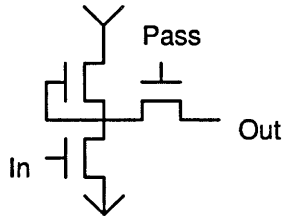

Figure 3.1: Example circuit for illustrating net behavior

It's easy to see that a net behavior expression for a node gets very big if the node is part of a large DCN. Silica Pithecus doesn't look at every possible combination of DCN inputs individually, but the number of clauses in the net behavior expression still grows exponentially with the size of the DCN.

The second step in the verification process involves a transformation from net behavior expressions to a digital representation. A digital representation is (more or less) a representation of the boolean functionality of the circuit. Each net in the net behavior description is analyzed to find the corresponding boolean value of the node. In the third step, the digital behavior is matched against a procedural description of the intended behavior of the circuit.

In addition to abstracting structural descriptions to behavioral descriptions and matching derived behavior with intended behavior, Silica Pithecus passes constraints around. Constraints are things that have to be satisfied for the abstraction from structure to behavior to be valid. Silica Pithecus takes constraints as inputs and produces constraints as outputs. Input constraints are constraints that are guaranteed by the user to be satisfied. An example of such a constraint is the promise that the boolean values of two inputs will always be mutually exclusive. Output constraints are precise conditions under which Silica Pithecus wants to accept a design.

The great thing about constraints is that they allow a circuit to be broken up along arbitrary lines and verified hierarchically. Remember from chapter 2 that DCN's are the natural units with respect to circuit correctness. A DCN contains

all the paths that can connect a node n to Vdd or Gnd. All those paths have to be taken into consideration when ensuring that n always gets a proper logical value. The straightforward algorithm for ensuring a good logical value on n looks, for all combinations of input values, whether there is a connection to Vdd or Gnd. This algorithm is exponential in the number of inputs. Silica Pithecus can get around this problem by breaking up large DCN's, processing the parts separately, and issuing constraints under which the parts are correct. The constraints are handled higher up in the hierarchy, or passed on to the user if they can't be verified.

Constraints and hierarchy are not a panacea for the combinatorial blowup problem, though. They rely on the goodwill of the user to break up large DCN's, and in many cases he is unlikely to do so. A 32-input nor gate, for instance, will most probably remain intact.

3.3.3 Bolsens' DIALOG

DIALOG is different from the former two systems in that it doesn't check whether the behavior of a system corresponds with a given specification. DIALOG checks whether a circuit has a deterministic boolean behavior without matching it against a behavioral specification. It detects electrical bugs like charge sharing, W/L-bugs, sneak paths and races. The system is based on a precise formulation of what it means for a circuit to be free of electrical bugs. A sequential system is viewed as an interconnection of combinatorial blocks (CB's for short) which connect through memory nodes and control nodes; the CB's are required to have a correct steady state behavior and a correct transient behavior, and the memory nodes are required to have a correct memorization behavior (see chapter 2).

DIALOG proceeds roughly in two phases. There is a preprocessing phase in which a sequential circuit is partitioned in CB's, and the CB's in DCN's. Loops within a CB are cut after being checked against certain correctness requirements. This allows the combinatorial block to be broken up in DCN layers, where the outputs of one layer are sufficient to compute the outputs of the next layer.

DIALOG then proceeds to verify the CB's for correct steady state behavior. First it fires rules to detect subcircuits that it is familiar with (e.g. complementary

static gates). For the rest of the CB, it executes a generic algorithm. For every possible combination of inputs to the CB it does the following. It looks at each output of each DCN. For each output, it looks whether it is connected to *Vdd* or *Gnd*. If it is connected to both or to none of them, it checks whether there is a sensitive path from the node to a CB output. If there is, it studies the node in more detail, and prints an error or warning message if needed.

If the whole CB has to be verified with this algorithm, the complexity is exponential in the number of inputs to the combinatorial block. The complexity is smaller if the system found circuit portions in a familiar design style. For instance, if a CB has a complementary static gate with n inputs at its front end, the system can check off this gate as correct, and consider its output as input to the remainder of the CB. Thereby the number of inputs decreases by $n - 1$, which cuts the computation time by a factor of 2^{n-1}.

The generic algorithm incorporates a rewrite rule approach to verification. The path from premises to conclusion is known from the beginning, but it takes a long time to reach the destination. The system needs to accumulate a lot of intermediate results before reaching a final conclusion. In particular, it needs to derive the behavior of a combinatorial block for each combination of inputs. The heuristic rules that are added to relieve the blowup problem incorporate a mapping approach. They go directly from premises to conclusions. Still, they form only loose patches. There is no coherent organization of knowledge about familiar circuit configurations which would allow complete circuits to be mapped to a statement about their correctness.

3.3.4 Bryant's MOSSYM

Bryant's MOSSYM comes close to a verification system, but it's a bit different in spirit. MOSSYM is a symbolic simulator. It's a simulator which can take symbols as inputs to circuits. It produces symbolic expressions as outputs. The system has exactly the structure of a numerical simulator, except that it performs operations in an algebra over boolean functions instead of an algebra over boolean values. The system is typically used to simulate datapath portions with symbolic data inputs and numerical control inputs.

In principle, MOSSYM could be used as a verification system by giving it symbolic values for all circuit inputs. If MOSSYM produces symbolic expressions for all outputs, the circuit is correct in the sense that there are no electrical bugs that prevent it from having a deterministic boolean functionality. It would verify correctness in the same sense as DIALOG. Two comments are to be made on using MOSSYM in this fashion:

1. MOSSYM can't detect races and can't handle static feedback loops. A simulator uses a certain scheme for ordering the computations for each component of the circuit. The scheme that MOSSYM uses is too rudimentary to be able to handle races and static feedback. Furthermore, its circuit model doesn't account for threshold drops over transistors, and the user has to map capacitances and W/L-ratios onto a discrete ordering.

2. MOSSYM is at least as inefficient as the other systems. Among other things, it has to prove boolean equivalence of some expressions, which is an NP-complete problem. The problem occurs where it has to compare two boolean expressions that it computed for an output. The two expressions should evaluate to respectively 1 and 0 if the output carries a 1, to 0 and 1 if it carries a 0, and to 1 and 1 if it carries an X. One boolean expression is computed from paths that connect the output to a node with a 1 value, the other from paths that connect it to a node with a 0 value. For the output to have a correct logical value, the two expressions should be complementary. The problem is reminiscent of the straightforward algorithm for checking correct steady state behavior of a DCN. The problem occurs in one form or another in all systems discussed so far.

In short, the algebraic machinery underlying switch level simulators is too narrow to cover all aspects of circuit correctness, and it doesn't have any advantage from the standpoint of efficiency. Bryant didn't carry on this approach to full verification. Instead, he continued working on simulators. [Bryant 87].

3.3.5 Spickelmier's Critic

Critic is a knowledge based system for critiquing circuit level designs. It has a loose enumeration of possible error configurations, and looks whether one of them occurs in a given design. Critic can criticize a design, but it can't tell whether a design is correct or not.

Critic operates in two phases. The first one is a structure finding phase. A structure is a certain configuration of transistors. Critic has a number of those in its knowledge base, and it can recognize them in the circuit. After that comes the error checking phase. The knowledge base has, for every structure, a list of possible error situations. Critic looks at every structure instance in the circuit, and goes over the appropriate list of errors to see if any of them actually occurs.

There was no attempt in the design of Critic to come up with a complete enumeration of possible error configurations. To cite Spickelmier in [Spickelmier 88]:

> "The main purpose of Critic was not to collect knowledge of circuit design, but to build a circuit critiquer that was integrated into a design system, with interactive control."

3.4 The Denotational Semantics Approach

My denotational semantics based system operates in three steps. In the first step, the circuit is parsed according to a circuit grammar. On the resulting parse tree, the system applies valuation functions to derive an internal behavior description. This is the second step. While performing this mapping, the system issues and checks circuit level constraints that have to be satisfied for the mappings to be valid. Some of these constraints can be verified right where they emerge (e.g. W/L-constraints and capacitive constraints). Others are issued explicitly and verified later on (e.g. the NORA constraints for avoiding input delay races). In the third step, the derived behavior is matched with a procedural specification supplied by the user.

The system is less competent than Gordon's logic based system, because it takes a procedural behavior specification that mirrors the structural representation. This

approach is imperative, however, in order to arrive at an efficient system. Checking whether two combinatorial functions are equivalent, is an NP-complete problem. The way to tackle this is to impose the same hierarchy on the two functions, and then decompose the problem. Imposing the same hierarchy in structure and behavior means that the behavioral description has to be procedural, that it describes the detailed process of how the outputs are computed from the inputs. My system is a bit more flexible than this. It allows boolean portions of the behavior description not to be a perfect mirror of the structure (it does tautology checking on them), and it allows the behavioral hierarchy to be a subset of the structural hierarchy: all modules in the structural hierarchy need to have an equivalent in the behavioral hierarchy, but not vice versa. For example, an *and* circuit consisting of a *nand* subcircuit and a *not* subcircuit, is successfully matched with a single *and* function.

The denotational semantics approach is potentially also less competent then the rewrite rule approach in Silica Pithecus and DIALOG, because it imposes a grammar on circuits. The difficult part in designing a denotational semantics based system is to come up with a tolerant enough grammar. The current grammar comes close to being successful in this respect, but there are some correct circuits that it rejects (see chapter 4 for more on this). Still better grammars can be realized by modifying the grammar formalism in which the current grammar is written.

The major advantage of the denotational semantics approach is efficiency. The key difference between this approach and other approaches is the coherent representation of large scale knowledge about VLSI circuits. Instead of figuring out the behavior of a circuit starting form the behavior of single transistors, my system parses the circuit into a hierarchy of configurations that it is familiar with. For each such configuration it knows exactly where to look for circuit level bugs and how to derive a behavioral description of it.

The approach has much in common with the concept of frames for knowledge representation. Frames are large structures that represent familiar situations. It has slots for things that always occur in that situation. When an agent is confronted with a new situation, it selects the proper frame and fills the slots with the right elements of the situation (some slots may have default values as well). The idea of

frames was brought into AI by Marvin Minsky [Minsky 81,Minsky 85], but existed in other fields before (as he acknowledges). Minsky presented frames as an alternative for small, independent chunks of reasoning, which dominated much of the early research in AI, typically in the form of inference rules in a formal logic.

Chapter 4

A Circuit Grammar with High Coverage

The preceding chapters gave an extensive introduction to circuit verification. We discussed what design correctness means and how it can be verified. In particular, we saw that certain aspects of correctness lead to exponential running times in systems that are organized around small chunks of reasoning. All existing verification systems see no more than arbitrary transistor interconnections in circuits, and employ the behavior model of a single transistor to figure out how circuits work. My system recognizes a richer structure in VLSI circuits, and it relies on this structure to give meaning to the circuit, much like humans rely on the grammatical structure of a sentence to give meaning to it. Once the syntactic structure of the circuit has been revealed, the remaining work is much easier.

We now have to argue that we indeed built a working system that is efficient and that covers all aspects of correctness accurately. Arguing efficiency is the easy part. It follows easily from the denotational semantics paradigm on which the program is based. Accuracy is the difficult part. We have to show that the program comes close to accepting all correct circuits and rejecting all faulty ones. This chapter contains part of the argument. It presents the circuit grammar that the program is based on. We will show that it comes close to covering all correct combinatorial CMOS circuits, while still excluding certain circuit bugs and revealing enough structure to make the semantic part easy. The grammar doesn't have to include all circuit level constraints. Certain constraints can be checked at the semantic level. The following chapter discusses the program as a whole, and shows that it detects all bugs. The

burden of this chapter is to demonstrate that the grammar covers a wide range of correct circuits.

The chapter is organized in three parts. The first part reviews Cyrus Bamji's work. It discusses the abstract parser that he built, and it differentiates the scope of his work on verification from the scope of my work. The second part gives an overview of the grammar that I wrote, and the last part contains a conclusion.

4.1 Cyrus Bamji's GRASP

Cyrus Bamji was the first researcher to introduce grammars into the world of VLSI circuits [Bamji 89]. He wrote GRASP, an abstract parser, which takes a circuit grammar, and produces a concrete parser. He used the program for design style verification. As an example, he implemented a grammar which captures the NORA design style. NORA is a design discipline for combining dynamic and static logic in a way which makes internal delay races impossible [Goncalves 83]. [1] NORA imposes structural rules to avoid input delay races. It constrains the way dynamic and complementary static gates may be interconnected. Therefore, it is perfectly fit for implementation as a set of grammar rules. Apart from constraints for avoiding charge sharing in dynamic gates, Bamji's grammar captures all correctness requirements within the NORA design discipline.

My verification system is intended to handle all correct circuits (not only ones that adhere to a certain design methodology), and it is intended to cover all correctness requirements. It's clear that, for this purpose, grammars are not sufficient any more. A grammar can't capture constraints on capacitances or W/L ratios, for instance. Constraints that are not checked by the parser are verified after parsing. My grammar is a tolerant one. Its range space is a superset of the set of all correct circuits.

My circuit grammar still sits on top of GRASP, the abstract parser that Bamji wrote. What follows in this section is a quick introduction to GRASP. Much of this

[1] NORA has also rules for avoiding clock races with a two-phase clock, but those were not implemented in Bamji's grammar. Bamji used a four-phase non-overlapping clock.

will become clearer and more concrete when we look at grammar rules in the next section.

GRASP takes in a grammar and produces a concrete parser corresponding with the grammar. The concrete parser takes in a network of modules and nets. Modules have a module-type and an indexed set of pins. For each module-type, there is a fixed number of pins. For instance, modules of type "n-trans" have 3 pins. Connections between modules are established by nets. A net has pointers to all the pins that it is connected to, and vice versa.

Module-types correspond to variables and terminal symbols in string grammars. There is one particular module-type which plays the same role as the start variable in a string grammar. And there are production rules that have a module-type on the left-hand side and an internal structure, possibly with a presence and/or absence condition, on the right-hand side. Internal structures, presence conditions and absence conditions are all networks of modules and nets. A subnetwork that matches with the internal structure on the right-hand side of a production rule can be reduced to a module of the type on the left-hand side, if there is a surrounding network matching with the presence condition (when there is one in the rule), and if there is not a surrounding network matching with the absence condition (if there is one).

When a network is parsed, it is reduced to fewer and fewer modules, while the number of nets that connect it to the outside world remains constant. In order to retain a fixed number of pins for each module-type, net bundles had to be introduced. A net bundle is simply a set of nets which is treated as one object. It plays the same role as single nets. For ease of exposition, we refer to both as net bundles from here on. Apart from connections between a net bundle and a module pin, there are also connections between a net bundle and another net bundle. Those connections are inferior, superior and adjacent pointers. If net bundle $n1$ has an inferior pointer to net bundle $n2$, $n2$ has emerged from bundling $n1$ with some other bundles. $n2$ then has a superior pointer to $n1$. If $n1$ has an adjacent pointer to $n2$, $n1$ and $n2$ have exactly one subbundle in common.

The concrete parser that GRASP produces does no backtracking. Once it has

reduced a subnetwork to a certain module, it never undoes the reduction again. Therefore, a grammar that is given to GRASP should be deterministic. There should never be more that one grammar rule according to which a module can be absorbed in a bigger module. Deterministic reduction leads to a very efficient parsing algorithm. The algorithm uses an queue of modules to schedule its work. Initially the queue contains all the primitive modules in the network. The parser performs the following action until the queue is empty: it removes the first module from the queue, fires any applicable grammar rule in which it appears in the internal structure or in the presence condition, puts the resulting module on the queue, and removes from the queue all modules that were absorbed in the new module. The complexity of this algorithm depends on the density of the network (i.e. on how heavily the modules are interconnected), but it is linear for practical VLSI circuits.

4.2 Summary of the Grammar

This section presents the circuit grammar that is used by my verification system to parse a circuit. What is shown below is a summary of the actual grammar that was written for GRASP. I have filtered out a number of details that have to be taken care of when writing grammar rules in GRASP's grammar format. What is presented as a single production rule below, often requires multiple rules in the actual grammar. Also, a sequence of rules that is almost identical to a sequence that appeared before, is often left out. Furthermore, I have often used textual annotations to summarize the effect of presence and absence modules in a rule. The pseudo grammar below is only intended to provide an idea of what circuits are covered, how they are covered, what circuits are not covered, and why they are not covered. The reader who is interested in the precise details of the grammar is referred to the code.

Still, this section has a lot of meat. In order to convince the reader of the competence of my verification system, the grammar is more than other parts in need of strong argumentation. By consequence, I'm going a far way towards the details of the code to demonstrate the coverage of my system. Of course, the coverage isn't perfect and the grammar is indeed the weak link in my system. Improving

the grammar is the main avenue for further research. That's another reason for elaborating extensively on the grammar that I have so far.

This being said, let's focus on the grammar now. The main idea behind the grammar is that a combinatorial circuit is an interconnection of DCN's which have a general form as depicted on figure 4.1. As mentioned in the third section of chapter 2, there may be loops of DCN's, so long as there is a DCN in the loop whose output is completely determined by inputs external to the loop. In other words, the loop is only allowed to reinforce values that are determined by inputs external to the loop.

We are already hitting one case where my grammar falls short of covering all correct circuits. Except in certain cases (that occur very frequently), my grammar doesn't cover loops at all. In order to cover legitimate loops and reject illegitimate ones, the parser would have to look inside modules that it formed before (to see whether a W/L requirement is met, for instance). This is not possible within GRASP's framework. There are two ways out. One way is to add features to GRASP so that it can look inside modules. An alternative solution would be to accept all loops, and to check them after parsing. The first solution is the most appropriate one. If the loop is not legal, it's better not to continue parsing the circuit. The new feature that would have to be added to the parser, is also necessary to make up for an other shortcoming of the current grammar that we will encounter later.

So, a combinatorial circuit is viewed as a loop free interconnection of DCN's, and a DCN has the form of figure 4.1: it has 1/0 generators in the front and (possibly) a pass transistor network in the back. A 1/0 generator can just be a *Vdd* or a *Gnd* node, or it can be a gate: a complementary static gate, a dynamic gate, or a more general structure with a pull and a pull down structure, where the logical context ensures proper behavior (an example of this follows later). A pseudo nMOS gate is also possible, but I left it out of my grammar to exclude static power dissipation. The pull up and pull down branches in gates can consist of parallel/series interconnections of transistors, but they don't have to (again, an example follows later).

A pass transistor network consists of nMOS transistors or parallel combinations

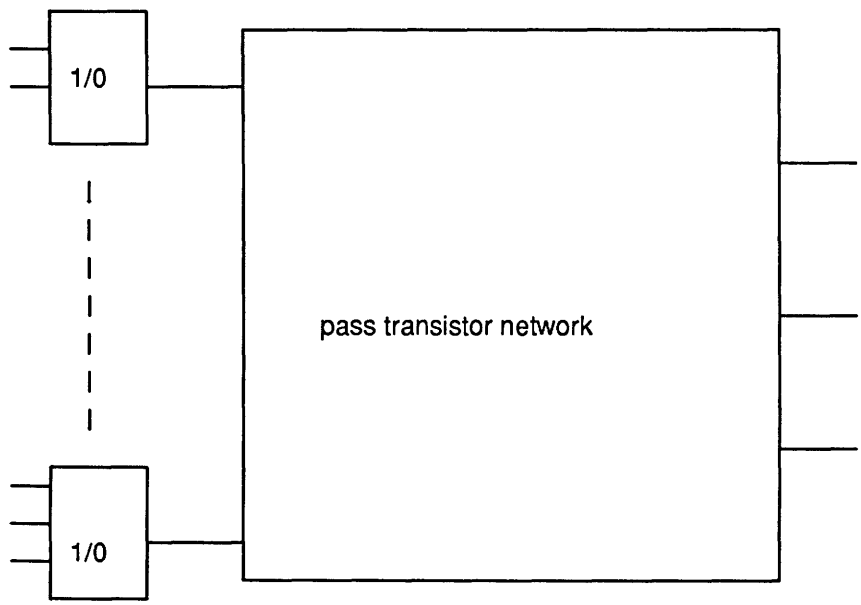

Figure 4.1: general form of a DCN

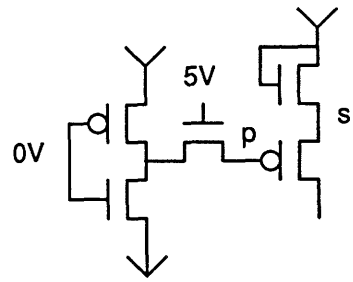

Figure 4.2: Compensation for a threshold drop over an nMOS pass transistor

of an nMOS transistor and a pMOS transistor. In the case of nMOS pass gates, there has to be level restoration or precharging at the outputs to make up for the threshold drop, unless the next stage has a compensation for it (see figure 4.2). Nodes in a pass transistor network may be connected to a gate or a gate-like structure, like in a manchester carry chain or a pass transistor exor.

We are now ready to look at the grammar. It has five levels of production rules. The first level contains rules that define intermediate transistor blocks from single transistors. These blocks form the building blocks for gates and pass transistor extensions of gates, which are defined at the second and third level respectively. The rules at the fourth level convert pass transistor extensions of gates to DCN's, and the fifth level defines blocks of combinatorial logic from DCN's.

4.2.1 Transistor Blocks

I have defined two kinds of transistor blocks as building blocks for more complicated structures: ones that contain parallel/series interconnections, and ones that have non parallel/series interconnections. In the first category, I defined the module-types N*, P*, and V*: parallel/series interconnections of nMOS transistors, pMOS transistors and complementary transmission gates respectively. V* modules are used in pass transistor networks that don't need level restoration. I used V as in "valve". The * character is meant to suggest "one or more (of something)".

The production rules for N*, P* and V* are almost identical. The ones for

N* are shown on figure 4.3. Figure 4.3 says that an N* type module is an nMOS transistor, a parallel combination of two N* modules, or a series combination of N* modules.

A few words about graphical conventions are in place here. Figure 4.3 represents all the production rules with module-type N* as a left-hand side. A module-type is represented as a box with a name and an indexed set of pins. The different right-hand sides are separated by straight horizontals. The last one is followed by a broken horizontal. Each right-hand side is a network of modules and nets, with a box around some portion of the network. Nets and net bundles in network are represented by little circles, modules by boxes with indexed pins (little stubs on the box) and with the name of the module-type that they are an instance of. A plain line between a module pin and a net indicates that the pin is connected to that net. An arrow from a net or a net bundle to another net bundle indicates that the first one is inferior to the last one (i.e. it forms a subset of the bundle that it points to). The box with indexes in each right-hand side indicates how the pins of the module-type of the left-hand side map to nets on the right-hand side.

From a parsing point of view, the parallel rule, for instance, reads like this. Whenever you see two N* type modules whith both their 1 pins and 2 pins connected to common nets, you can combine them into a new N* type module with pin 0 connected to the bundle consisting of the nets at the 0 pins of the composing modules (you have to create that bundle now), pin 1 connected to the net at the 1 pins, and pin 2 connected to the net at the 2 pins. Everything inside the large box has disappeared now. The new N* module has been substituted for it. The N* module has three pins, just like an nMOS transistor: pin 0 is the gate, pin 1 the source, and pin 2 the drain.

Not all transistor configurations in VLSI circuits are parallel/series interconnections. Figure 4.4 shows a parity ladder, which is a non parallel/series interconnection. At each bit slice, the input pair at the left is swapped if the controlling bit is 0, and it is maintained if the controlling bit is 1. The input pair at the extreme left can be connected to Vdd and Gnd, or it can be connected to the drains of two transistors that connect both to Gnd. In this case, each intermediate pair consists

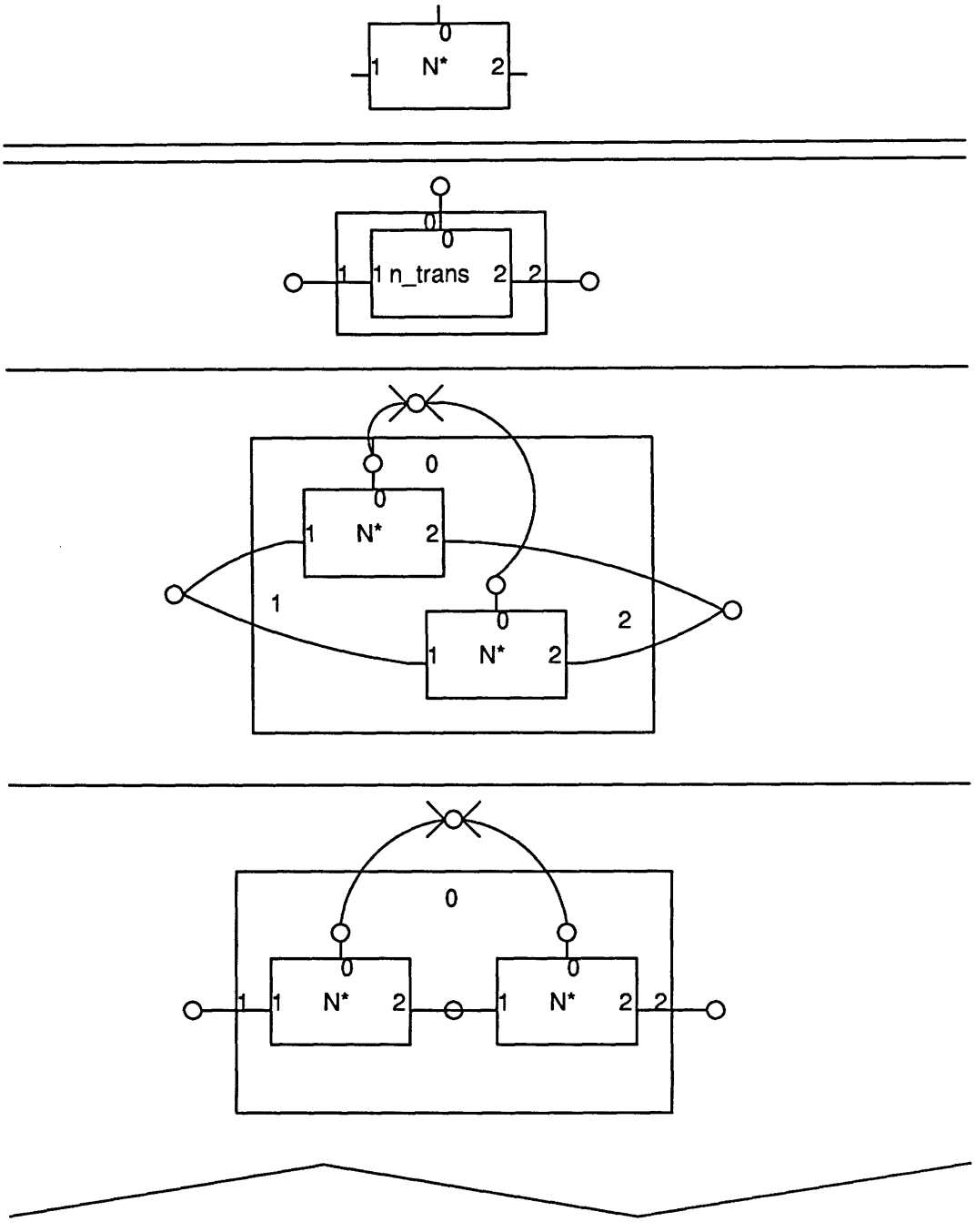

Figure 4.3: Production rules for module-type N*

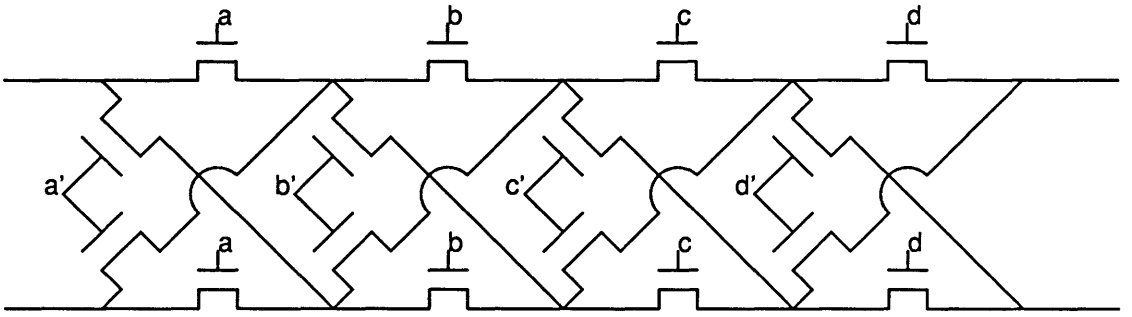

Figure 4.4: Parity ladder, example of a non parallel/series interconnection of transistors

of one node with a 0 voltage and one node with a high impedance voltage. The outputs at the right can be precharged (we then have a dynamic gate), or they can be connected to a pMOS transistor portion with a complementary functionality (to form a compementary static gate). Full blown illustrations of such circuits can be seen in [Weste 85] on page 334, figures 8.23.b and 8.23.c.

How do we write grammar rules that cover non parallel/series transistor configurations? We need a rule with a seed for such a configuration, and further rules that glue more transistors to it. The general form of a seed for a non parallel/series interconnection is shown on figure 4.5. Each block represents a transistor (or a V* block), and the dashed lines represent zero or more transistors in series. The problem is that these dashed lines can't be expressed in a circuit grammar. In a one dimensional string grammar, you can say something like "one or more of X" because the parser knows in which direction to look for more X's (there is only one direction), but in a two dimensional grammar, the parser doesn't know that any more. In order to detect an internal structure as in figure 4.5, the parser would have to perform search over an unbounded portion of the circuit.

Figures 4.6 and 4.7 show the production rules for module-type RN, a non parallel/series combination of transistors. The letter R comes from "recursive". Very often, the behavioral specification of these transistor blocks will be recursive, although they don't have to: they can also consist of a flat series of conditional

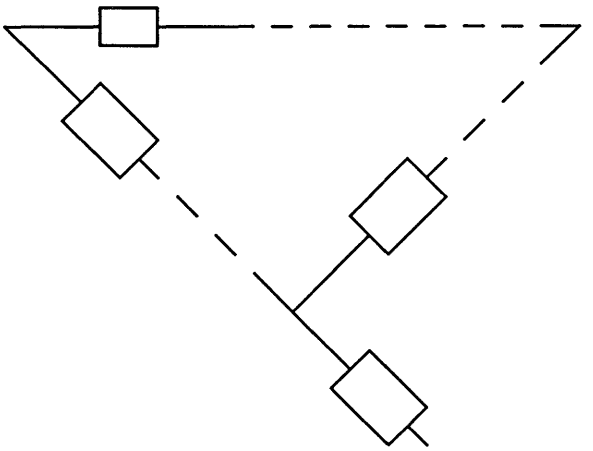

Figure 4.5: general form of a seed for a non parallel/series interconnection

statements. Boolean algebra is not appropriate any more to describe these circuits. The algebraic expressions for these circuits get exponentially big, and are neither suggestive for the functionality of the circuit, nor for the circuit implementation.

The first production in figure 4.6 contains the seed for an RN type module. The N* module below is a presence module. It has to be there for the rule to apply, but it is not taken in in the new RN module. The seed is less general than the one on 4.5, but it covers all the circuits that I have encountered.

An RN module has five pins. Pin 0 and 1 form the inputs to the left and the right portion of the module respectively. Pin 2 is the source (or bundle of sources), pin 4 the drain (or bundle of drains), and pin 3 the bundle of intermediate nodes. The second and third production rule for RN add N* modules to the left portion of it. The second rule takes in an N* module with its drain connected to the intermediate bundle, and the third rule takes in an N* module with its source connected to the source bundle of RN. The fourth rule adds an N* module, connected with its source to the intermediate bundle, to the right portion of the RN module. The last rule applies when nothing is connected to the intermediate bundle, and there is an additional layer of pass transistors behind the RN module. The RN module forms the left portion of the new RN module, and the first transistor in the next layer

forms the right portion.

With those rules, the parser can handle the parity circuit on figure 4.4. There are similar rules for module-types RP and RV with P* modules and V* modules as building blocks.

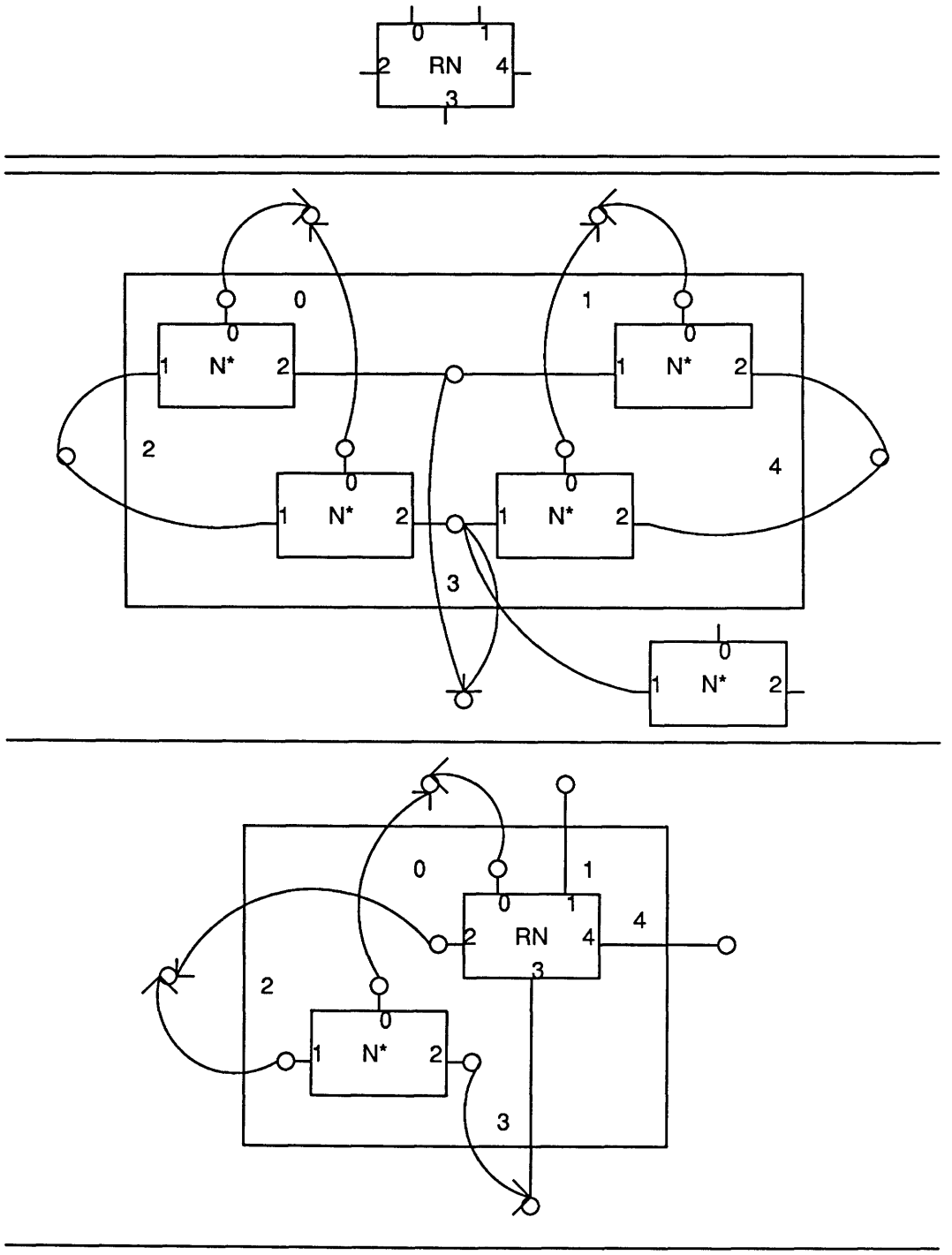

Figure 4.6: Production rules for module-type RN, first part

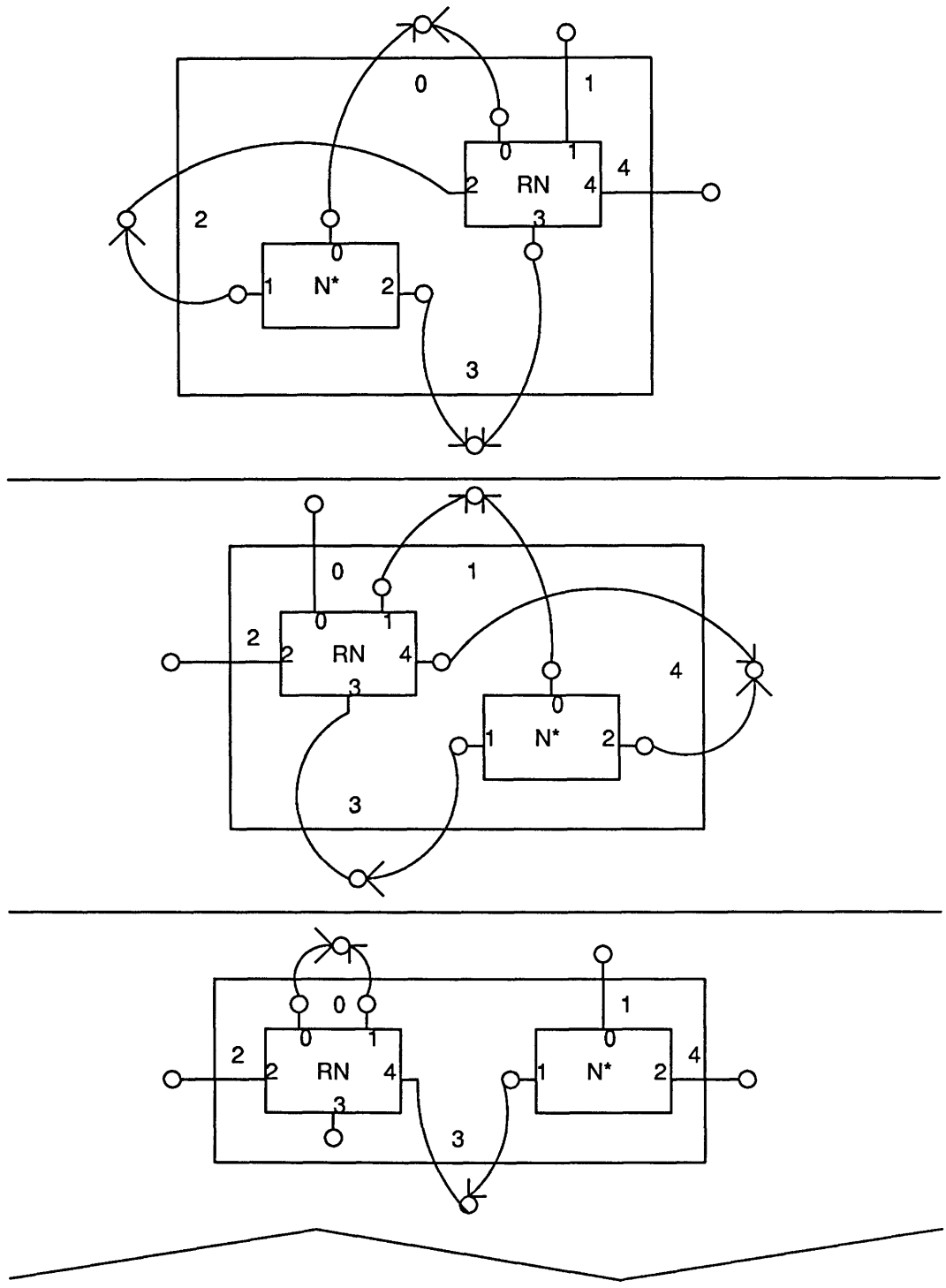

Figure 4.7: Production rules for module-type RN, second and last part

4.2.2 Gates

I defined three types of gates: complementary static ones (module-type S), dynamic ones (module-types DN and DP) and more general ones (module-type PS, to be read as "possibly static", depending on the logical context).

Module-type S represents complementary static gates. The production rules for it are shown on figures 4.8 and 4.9. The first rule covers the case where the pull up and pull down branches are parallel/series connections. The supply module in the rule is a presence module. It has to be there for the rule to apply, but it is not part of the internal structure of the new S module that gets created. Note that the parser hasn't checked whether the pull up and pull down branches are complementary. This is done at the semantic level, by doing tautology checking.

The second rule covers the case where the pull up and pull down branches are non parallel/series interconnections. As mentioned before, an example of this can be seen in [Weste 85] on page 334, figure 8.23.c.

The third rule covers gates with parallel/series blocks and more than one output. It only covers the case of two outputs. A similar rule for 3 or more outputs can also be written, but I don't know of any practical circuits for which this would be necessary. An example of such a circuit appears on figure 4.10 (the functionality of this circuit is explained later): the 6 transistor portion in the front matches with the internal structure of the grammar rule. It may be confusing that the net at pin 0 of the above most P* module in the internal structure remains inside the enclosing box, and that it isn't connected to anything else. Remember that the net bundle at pin 0 of the new module is created after firing the rule. It isn't there when a circuit portion is matched with the internal structure. Note also that, for the rule to apply, all the connection between the inside and the outside of the enclosing box should go over the nets that are connected to the pins of the newly created module. With this in mind, the rule in the bottom of figure 4.9 can be read like this: whenever you see two N* modules and two P* modules with their sources and drains connected as on the figure, you can merge them into an S module if all the nets connected to pin 0 of the above most P* module appear in the union of the nets connected to pin 0 of the other modules.

The circuit on figure 4.10 is a tristate buffer. Whenever C is 0, P becomes 1 and N 0, and OUT gets in a high impedance state. If C is 1, the 6 transistor portion behaves like an invertor on D, and OUT just follows D. The 6 transistor portion can be viewed as a nand gate and a nor gate merged together.

The two transistors in the back of figure 4.10 form an example of a PS type module. The rule for this module-type appears on figure 4.11. PS stands for "possibly static". The semantic machinery has to check whether the logical conditions for correct operation are satisfied.

Figure 4.12 shows the grammar rules for module-type DN. A DN type module is a dynamic gate which is precharged high. The rules for module-type DP, covering dynamic gates which are precharged low, are almost identical. The first production for DN covers the case where the pull down branch is a parallel/series block. Pins 0, 1, 2 and 3 on the clock module correspond to $phi1$, $-phi1$, $phi2$, and $-phi2$. On the figure, $-phi1$ is used as input to the gate. The real grammar has analogous rules with $-phi2$ as input.

The second production covers dynamic gates with a non parallel/series pull down branch. The rule on the figure covers only the case where there are two outputs. The real grammar is more general than that. It defines an intermediate module which takes in an arbitrary number of pMOS precharge transistors, and uses that as a building block for a DN type module.

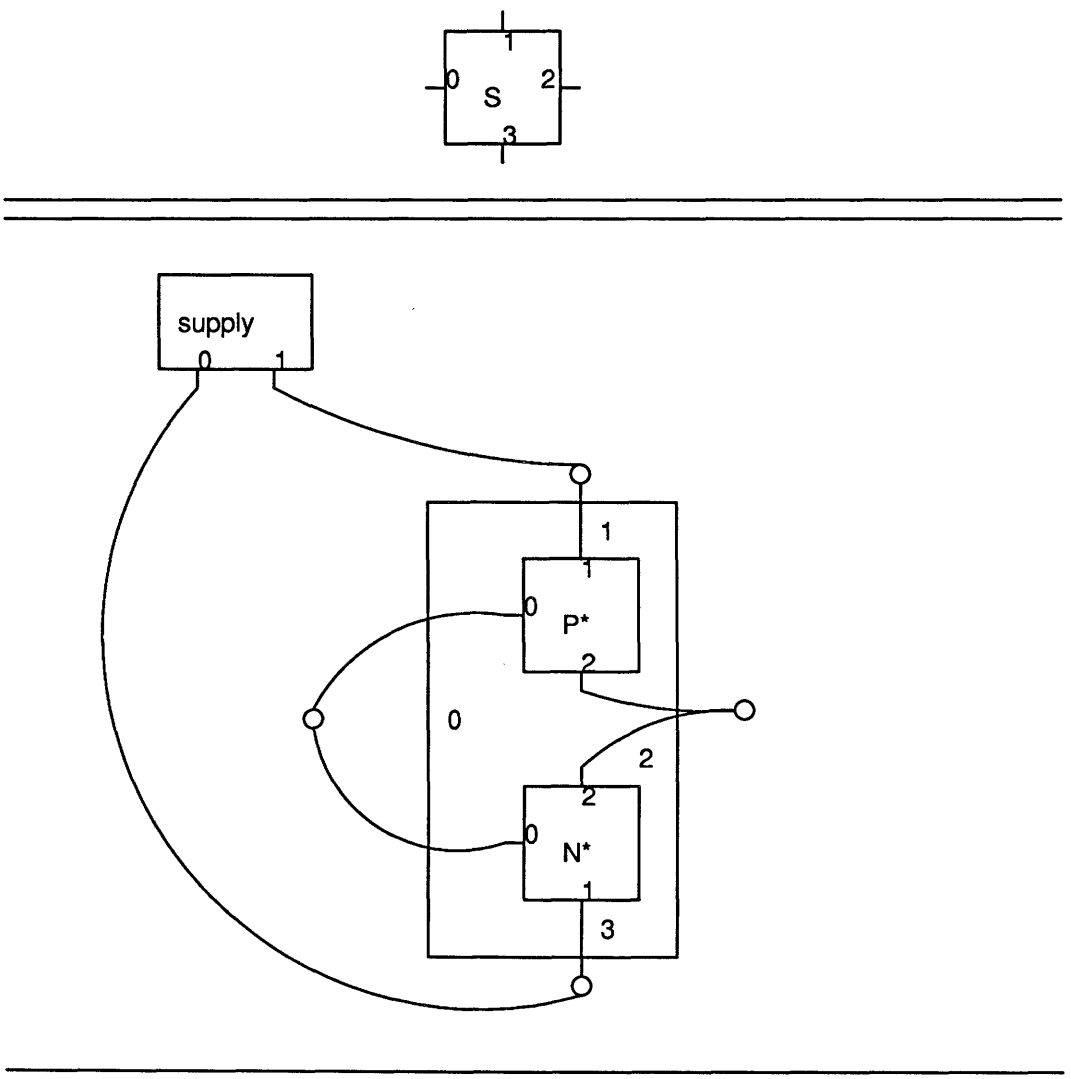

Figure 4.8: Production rules for module-type S1, first part

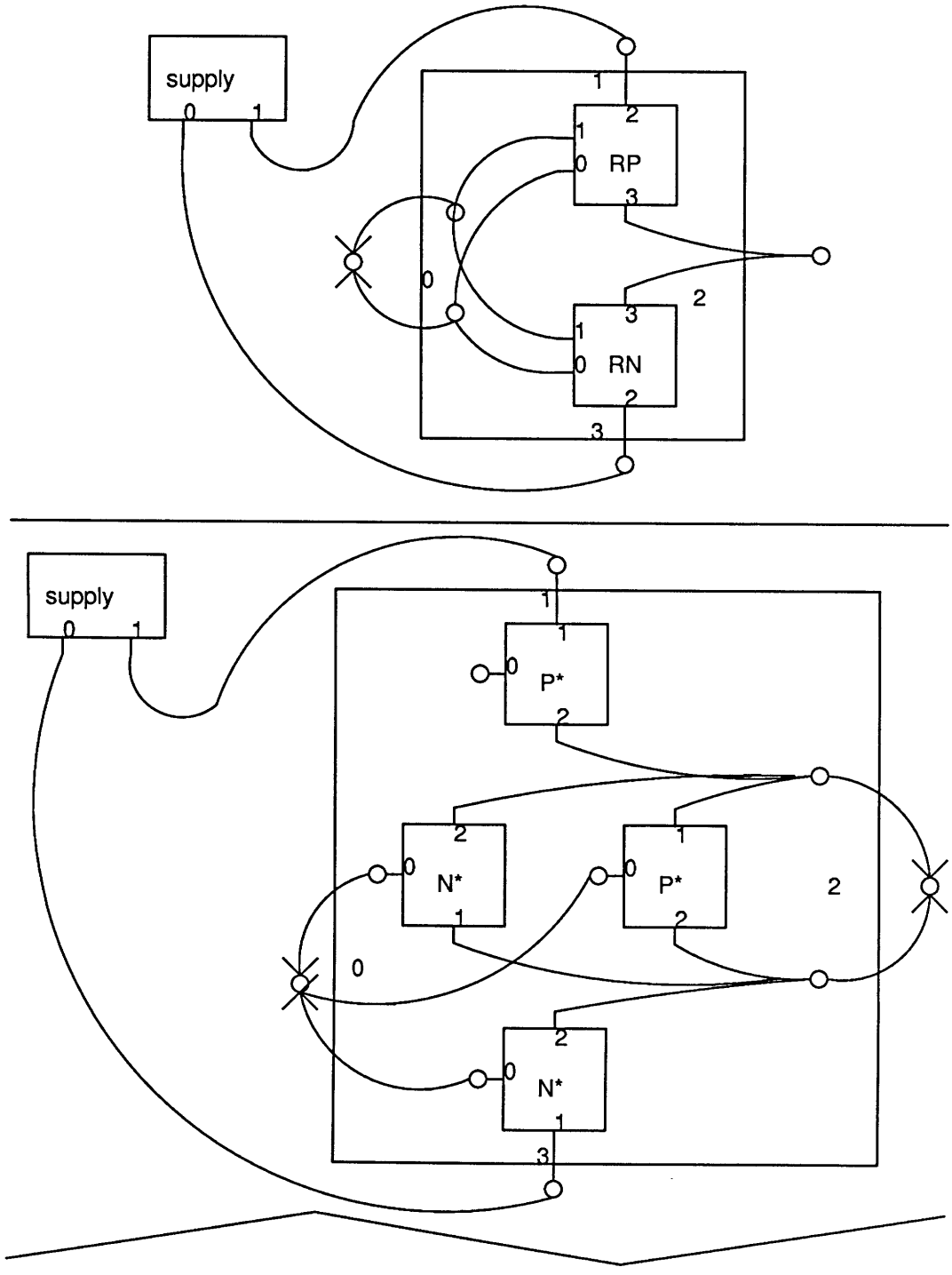

Figure 4.9: Production rules for module-type S1, second and last part

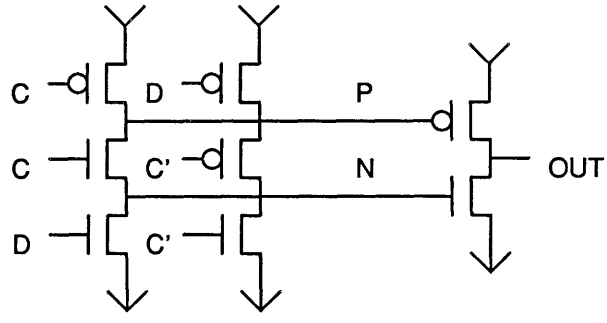

Figure 4.10: Tristate buffer: the 6 transistor portion in the front forms an S type modules with two outputs; the 2 transistor portion in the back forms a PS type module

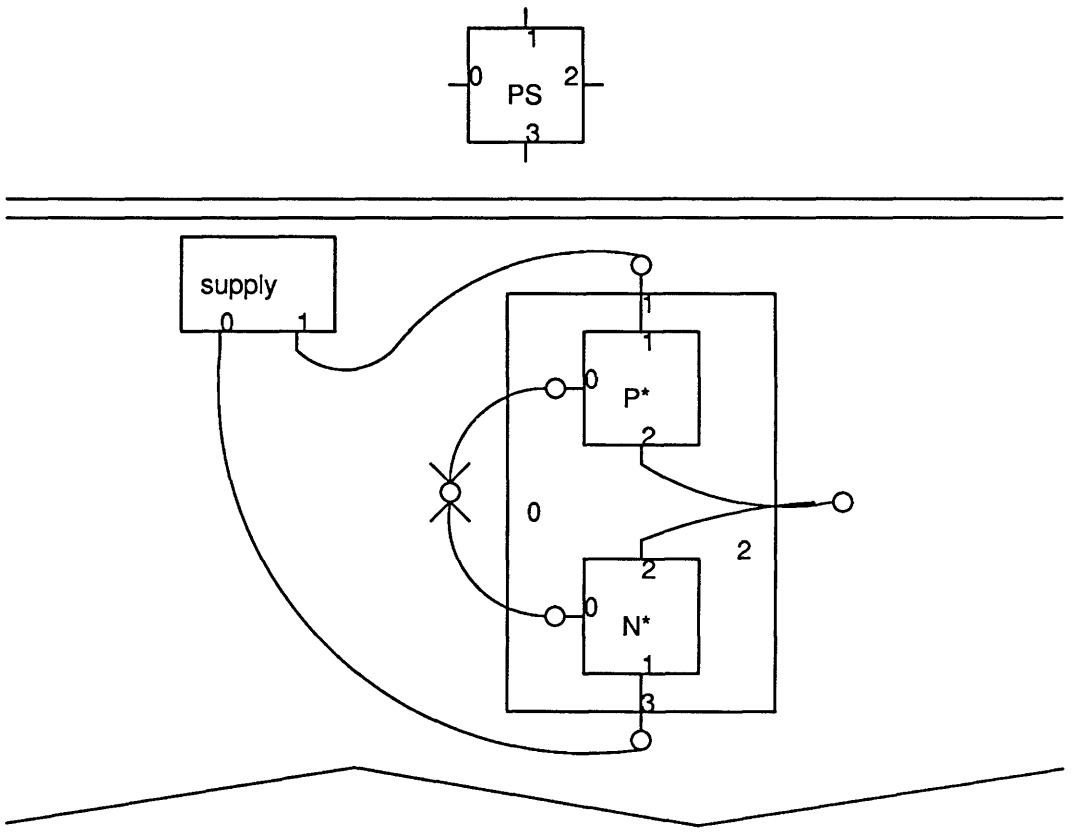

Figure 4.11: Production rule for module-type PS

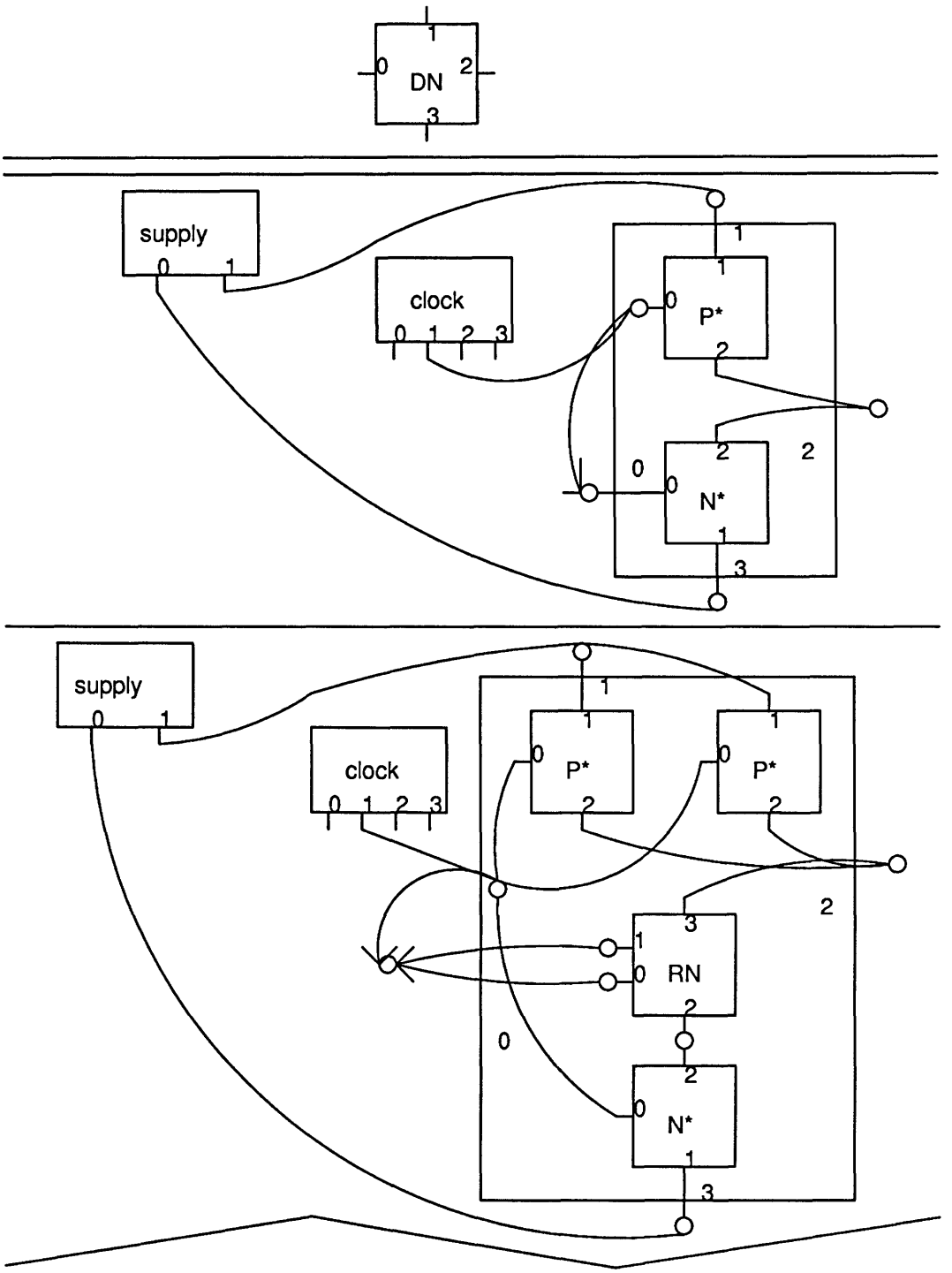

Figure 4.12: Production rules for module-type DN

4.2.3 Pass Transistor Extensions of Gates

Only pass transistor extensions of S modules are defined in this section. Extensions of DN and DP modules are very similar. Pass transistor extensions of S modules are covered by module-type S*. The first two productions for S* cover the base cases: their internal structures consist simply of an S module and a PS module respectively. The next production covers a pass transistor exor with input buffers, as shown on figure 4.15. The last two productions cover circuit portions with one or more S modules in the front and transmission gates in the back. S*FV! type modules contain complementary transmission gates; S*FN!p! type modules have nMOS transmission gates with the outputs connected to invertors with a feedback pMOS transistor for level restoration. Figure 4.16 shows an example of such a circuit.

Only productions that build up S*FV! type modules are shown here. The overall flow of these productions is this. First, an S* module is extented whith V* transismission gates at its output to form an S*FV module. The letter F in this name comes from "fanning out". The V* modules fan the output of the S* module to what eventually will be multiple outputs of an S*FV module. When nothing else is connected to the output of the original S* module, the S*FV module becomes an S*FV+ module, possibly combining with other S*FV+ modules which share outputs with it, forming a composite S*FV+ module. If no more drains are connected to the output of the composite S*FV+ module, it is converted into a S*FV! module (the ! character is meant to suggest "finished"; you can read it as "bang"). This module is then converted into a S* module, which can take in a new layer of transmission gates at its outputs. This being said, the productions for S*FV, S*FV+ and S*FV! should be clear by themselves. They appear on figures 4.17, 4.18, and 4.19.

Note that these rules don't cover transmission circuits with bypassing, as in figure 4.20. (Assume that the transmission gates on the top can't be merged into one transistor block for some reason). I didn't find a way to include bypassing and to still end up with a parse tree that makes sense of a circuit (i.e. a parse tree that contains enough information for the semantic process). In order to cover bypassing,

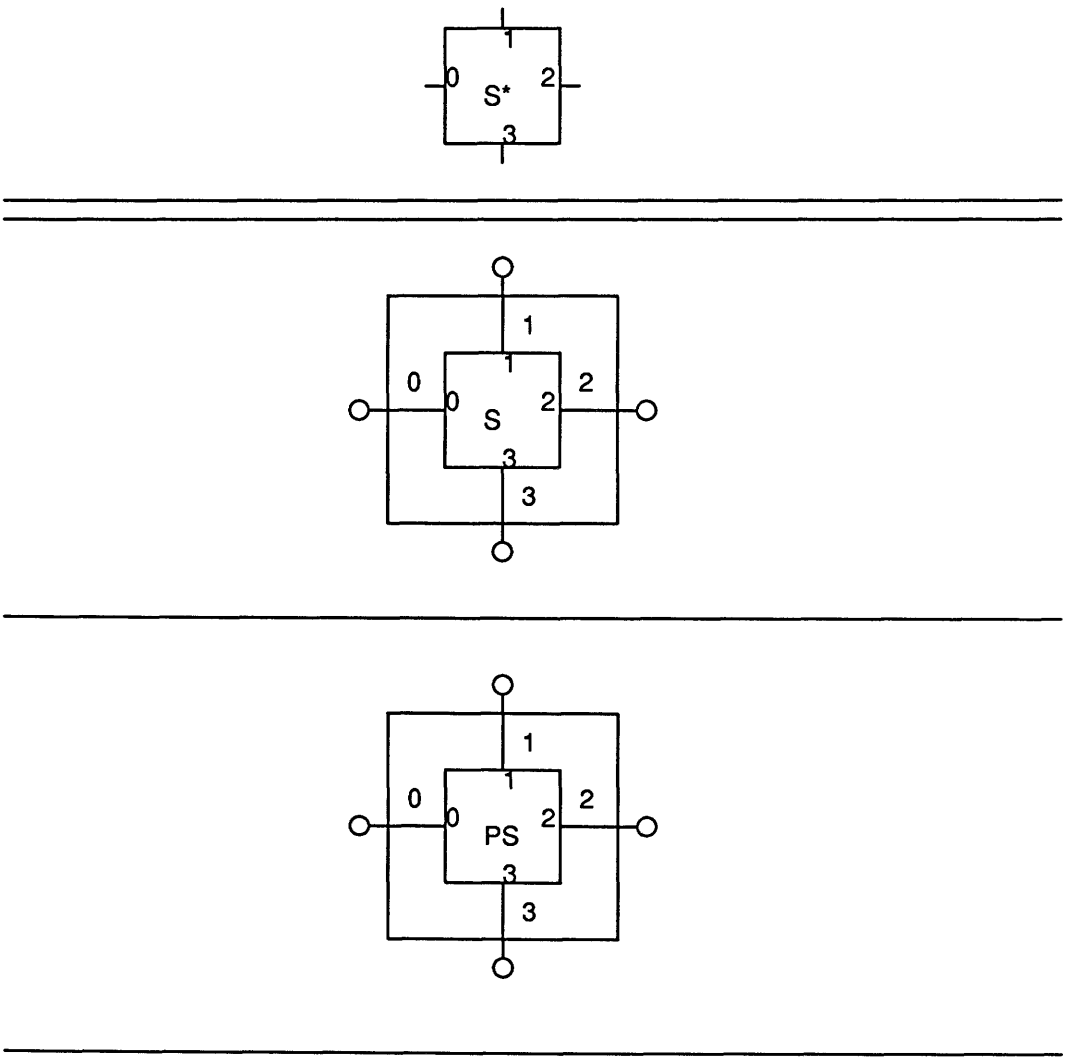

Figure 4.13: Production rules for module-type S*, first part

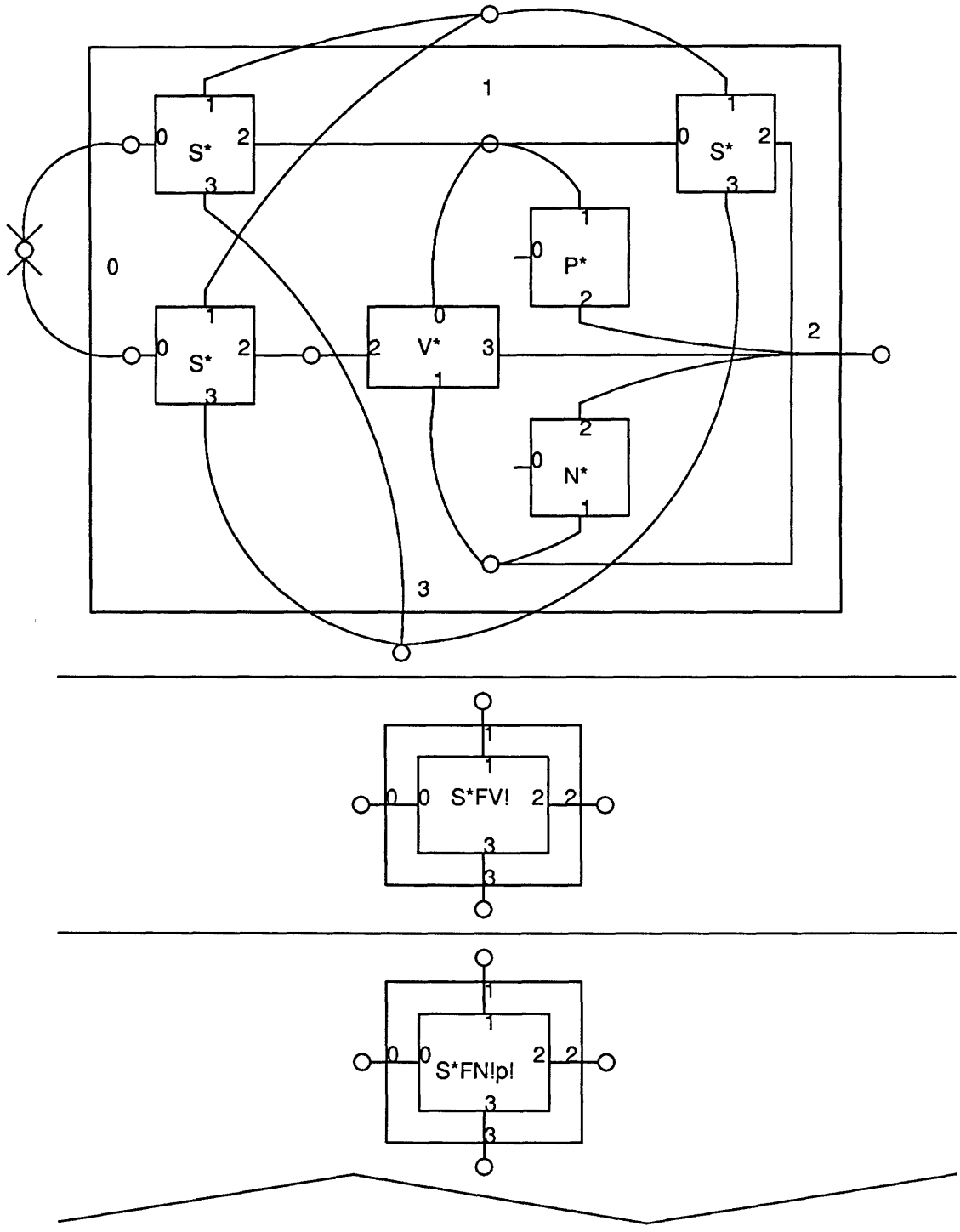

Figure 4.14: Production rules for module-type S*, second and last part

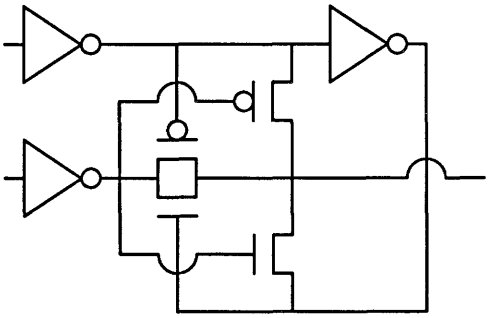

Figure 4.15: Pass transistor exor with input buffers: example of the right-hand side of the third production rule for module-type S*.

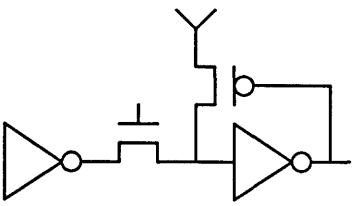

Figure 4.16: Example of an S*FN!p! type module

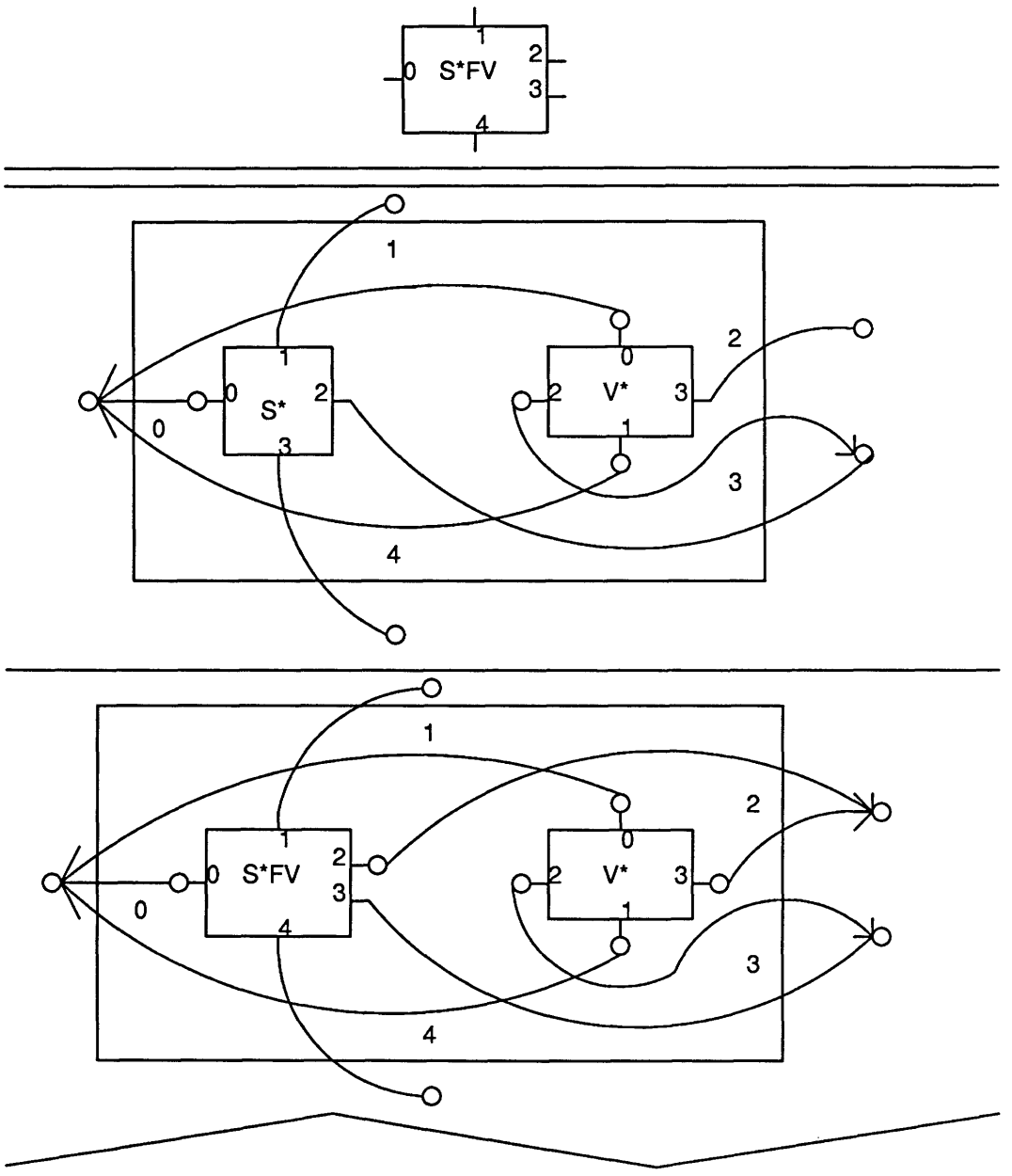

Figure 4.17: Production rules for module-type S*FV

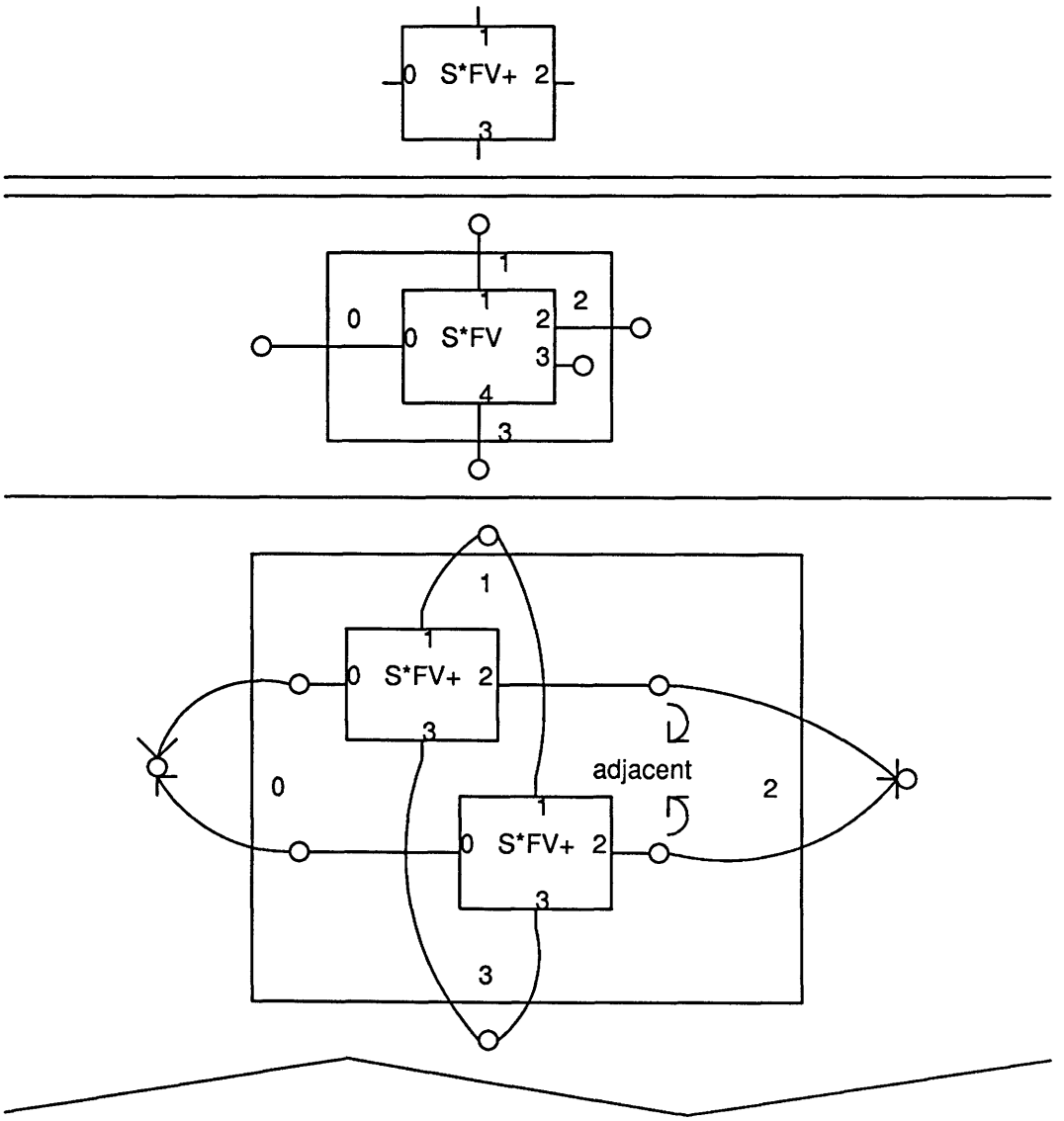

Figure 4.18: Production rules for module-type S*FV+

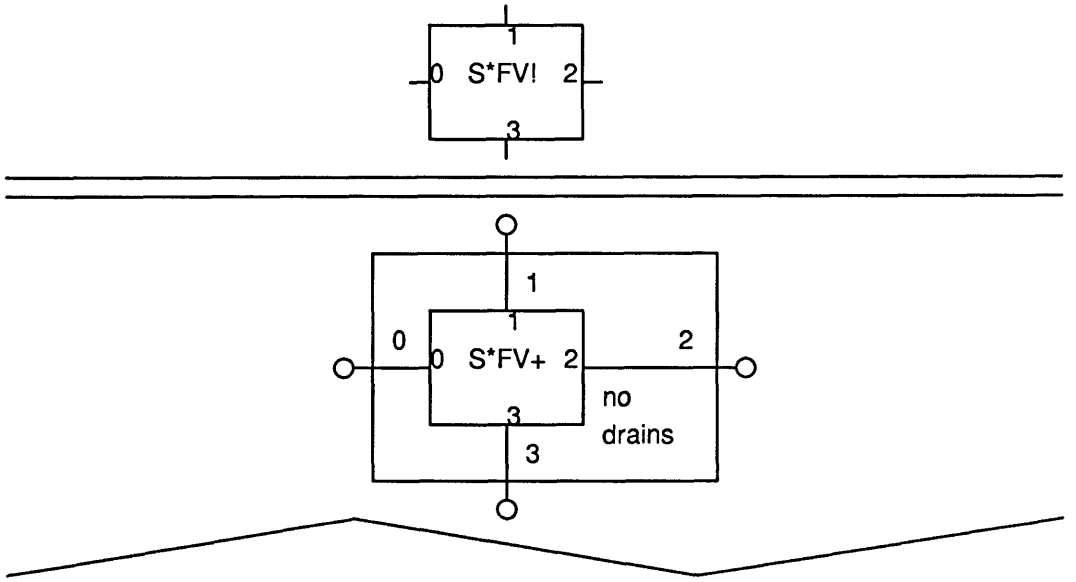

Figure 4.19: Production rule for module-type S*FV!

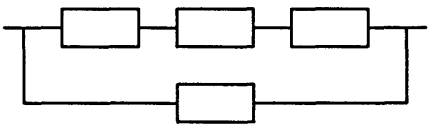

Figure 4.20: Bypassing in a transmission circuit

we need an additional feature in GRASP, similar to the one that I proposed to handle feedback loops. GRASP should be able to get its hands on modules that it already formed. In this case, it should be able to make changes in the parse tree it already built up. With the grammar rules above, GRASP would put the left most transmission gate on figure 4.20 and the bottom one in a first layer, and then get stuck somewhere. It should somehow be able to detect that it shouldn't have put the bottom transmission gate in the first layer, and put it in a different module in the parse tree.

The flow from module-type S* to S*FN!p! goes like this. The beginning is the same as the flow from S* to S*FV!: it goes from S* over S*FN and S*FN+ to

S*FN!. The following module-type in the flow is S*FN!p. It is an S*FN! with some outputs connected to a level restoration circuit. If all outputs are connected to such a circuit, an S*FN!p! gets formed, which can then be converted into a new composite S*.

The rules for pass transistor extensions of dynamic gates are similar, with one difference. A dynamic gate followed by pass transistors can be connected directly to the output of an other dynamic gate. This occurs in a manchester carry chain, for instance.

4.2.4 DCN's

DCN's are just S* modules, DN* modules or DP* modules that are "complete", in the sense that they don't connect to sources or drains of transistors or transistor blocks any more. This is expressed with a textual annotation in figure 4.21. The real grammar contains the appropriate absence module to express this. There is also a presence module in the rule. It is there for the following reason. It could be that an S* module has no sources or drains connected to it in the subcircuit that is being parsed, but when the subcircuit is implanted in a bigger circuit, there may be sources or drains connected to the S* module. If the S* module has its output connected to the input of an other gate, it is likely that the S* module is "finished", i.e. that there are no transmission extensions any more. This is only a heuristic, of course. It worked for all the test cases that I ran. If a circuit doesn't get parsed properly, the user can always change the hierarchical partitioning to arrive at a succesful parse.

Figure 4.21 shows only one production for a DCN type module. The other ones differ only in the presence module. There are also productions with a DN* type module and ones with a DP* module in the internal structure.

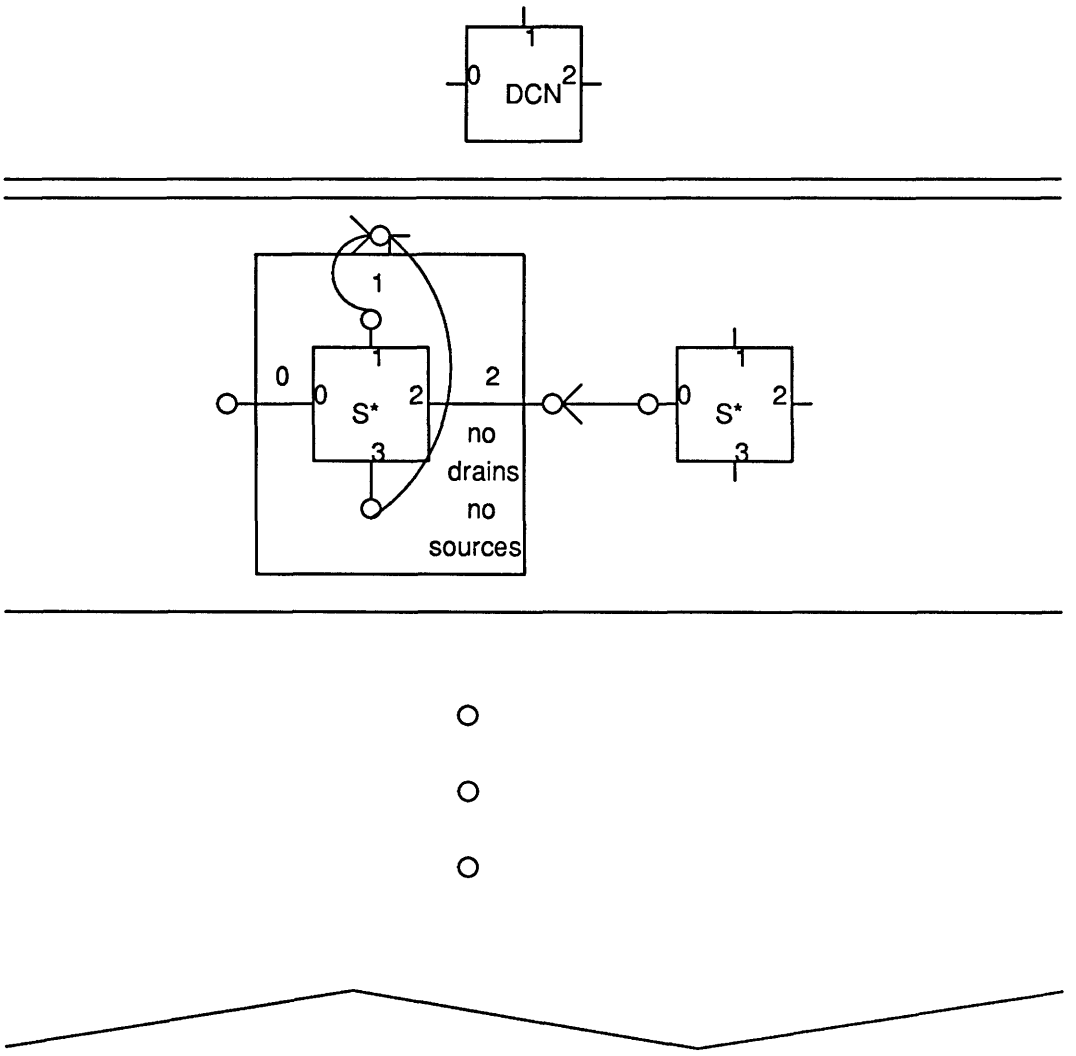

Figure 4.21: Production rules for module-type DCN

4.2.5 Combinatorial Logic

Combinatorial logic is covered by module-type CL. The production rules are shown on figures 4.22 and 4.23. The base case is a DCN which has inputs that are only connected to external inputs, clocks, other DCN inputs or other CL inputs. (External inputs are represented by a module of type "in", which has one pin.) The DCN should not be connected to the output of something. The reason why the input is bundled with the output will become clear later.

The CL module is then expanded in breadth and in depth. Let's look at the last production rule first. It leads to expansion in depth. A DCN is taken in if all its inputs appear in the outputs of the old CL, and if the outputs of the old CL and the ones of the DCN are disjoint (this excludes loops). The second production rule is necessary to build up broader CL's until a CL is formed whose outputs form a superset of the inputs of the next DCN, so that the third rule can be applied.

When a CL module is built up, all inputs and intermediate nodes get bundled with the outputs. This is necessary for the following reason. Suppose that a DCN A and a DCN B are connected up like in figure 4.24. If the inputs of A were not bundled with its outputs, the CL productions would not allow to parse the circuit. If A's inputs are bundled with its outputs, they are still available as inputs for the next layer of DCN's.

We have finally arrived now at the top module-type of the grammar. A CB type module is a complete combinatorial block. A CB is formed from a CL, if the CL is only connected to external things: external outputs, external inputs or clocks. External outputs are, similar to external inputs, represented by a module of type "out", with one pin. External inputs and clocks must be allowed at the output because the inputs of the CL module are bundled with the outputs.

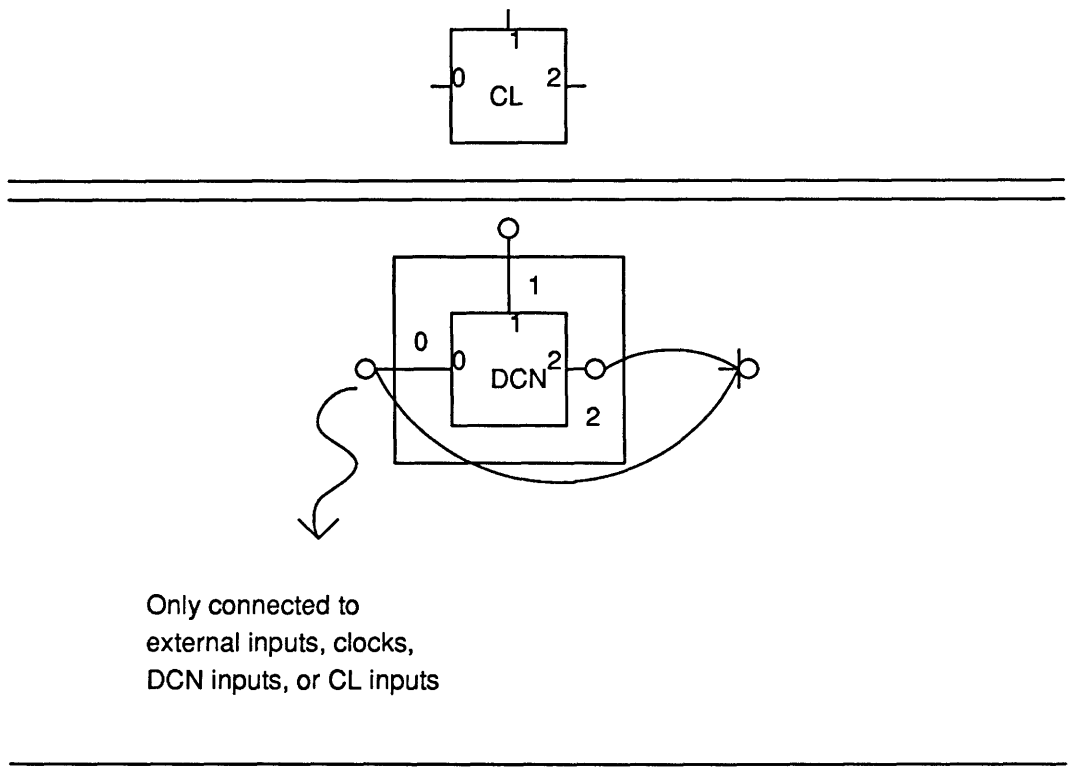

Figure 4.22: Production rules for module-type CL, first part

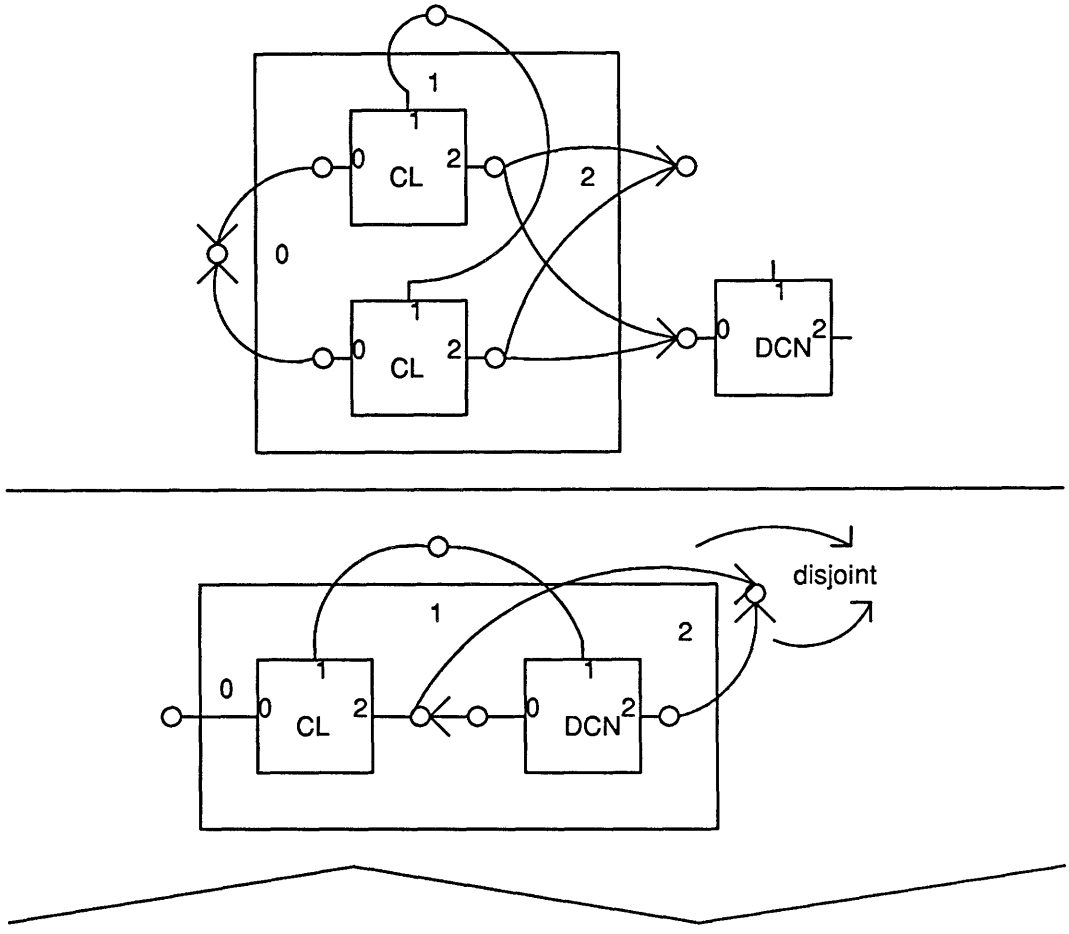

Figure 4.23: Production rules for module-type CL, second and last part

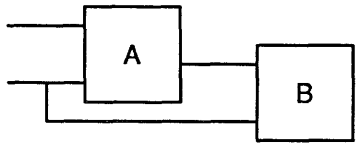

Figure 4.24: Example of a circuit where bundling of CL inputs with CL outputs is needed

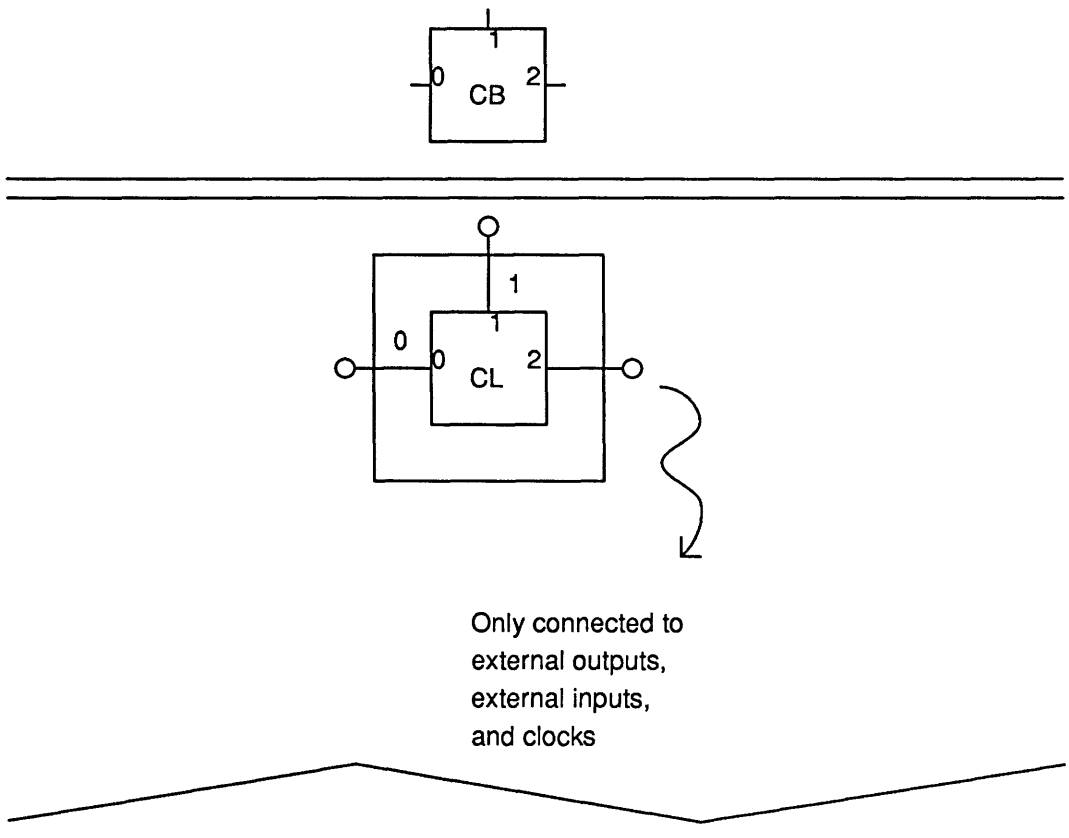

Figure 4.25: Production rule for module-type CB, the start module-type

4.3 Conclusion

This section concludes the chapter by stating succinctly what is covered by the grammar, what is not covered by it, and which checks still have to be performed at the semantic level.

4.3.1 What is covered by the grammar

The grammar covers loop free combinations of DCN's. The DCN's are made up of 1/0 generators and a pass transistor network.

A 1/0 generator can be one of the following:

1. A Vdd or Gnd.

   Currently, this is only covered where the Vdd or Gnd node feeds into a non parallel/series pass transistor network. Covering the case of parallel/series networks just requires writing more rules.

2. A complementary gate.

   The pull up and pull down branches may be parallel/series interconnections or non parallel/series interconnections. There may be one or more outputs. In the case of parallel/series branches, only gates with one or two outputs are covered.

3. A dynamic gate.

   Both parallel/series interconnections and non parallel/series interconnections are covered. In the latter case, there may be an arbitrary number of outputs.

4. A more general gate structure with a parallel/series pMOS pull up branch and a parallel/series nMOS pull down branch.

A pass transistor network can be one of the following:

1. One with complementary transmission gates.

This is only allowed if all the 1/0 generators are complementary static gates or Vdd or Gnd nodes. For the case of Vdd and Gnd nodes as 1/0 generators, only non parallel/series networks are covered. The specific case of a pass transistor exor is also covered.

2. One with nMOS transmission gates and level restoration at all outputs,

   This is allowed if the 1/0 generators are complementary gates or Vdd or Gnd nodes. For the case of Vdd and Gnd nodes as 1/0 generators, only non parallel/series networks are covered.

3. One with nMOS transistors and precharging at the outputs.

   This is allowed if the 1/0 generators are dynamic gates. The intermediate nodes and the outputs of the pass transistor network may be connected to the output of a dynamic gate.

4.3.2 What is not covered by the grammar

The following is a list of transistor configurations which are correct according to the switch level model that we use, but that are not covered by the grammar. The list is probably not complete, because it is difficult to envisage the set of all correct circuits. To the best of my knowledge, these are the configurations that are correct but not covered by the grammar:

1. Legal feedback loops over DCN's.

2. Bypassing in pass transistor networks.

3. Non parallel/series blocks that don't have a four transistor seed.

4. Parallel/series pass transistor networks that start directly from Vdd and Gnd.

5. A gate followed by pass transistors where both the gate output and the output of the pass transistors are inputs to other parts of the circuit.

6. Pass transistor networks with nMOS transistors and no level restoration, where the next stage is compensated for a threshold drop.

7. Pass transistor extensions of dynamic gates that are precharged low.

8. Complementary gates with parallel/series pull up and pull down branches and more than two outputs.

9. Combinations of pass transistor exors where certain nodes are shared.

The first two items can be covered by modifying the parsing framework. The third item can't be covered in general, but additional specific cases can be included by writing more grammar rules. The other ones just require writing more rules.

4.3.3 Checks that still have to be done at the semantic level

The following things still have to be checked at the semantic level:

1. That there are no W/L bugs.

2. That there are no charge sharing bugs.

3. That there are no input delay races in dynamic logic.

4. That there are no sneak paths or undesired high impedance states in pass transistor logic.

5. That the pull up and pull down branches in complementary gates have a complementary functionality.

Chapter 5

Denotational Semantics Implemented

The claim of this thesis is that the denotational method for defining the semantics of a programming language forms a powerful paradigm for circuit verification, one that provides efficiency and accuracy. That a denotational semantics strategy leads to efficiency is straightforward. The part of the claim that is most in need of support is the point of accuracy: the fact that a denotational semantics based verification system rejects incorrect circuits and comes close to accepting all correct ones.

The current chapter, together with the former one, build up this support by going down to the details of how denotational semantics can be implemented. The previous chapter showed that a circuit grammar can be written with a range space that comes close to covering all correct circuits. This chapter will show that the denotational semantics method forms an adequate framework for checking circuit level constraints that are not incorporated in the grammar, and for deriving a behavioral description which can be matched with the user supplied specification.

The overall architecture of Semanticist, my implementation of denotational semantics, is shown on figure 5.1. Semanticist takes in a circuit and a behavior description. It uses two intermediate representations, a network environment and a block diagram environment. A network environment contains bindings between cell names and network objects. Network objects in turn have pointers to module objects and net objects. The network environment can be viewed as an object oriented world containing structural information: information about how things are interconnected. After parsing, the network environment contains a parse tree of the

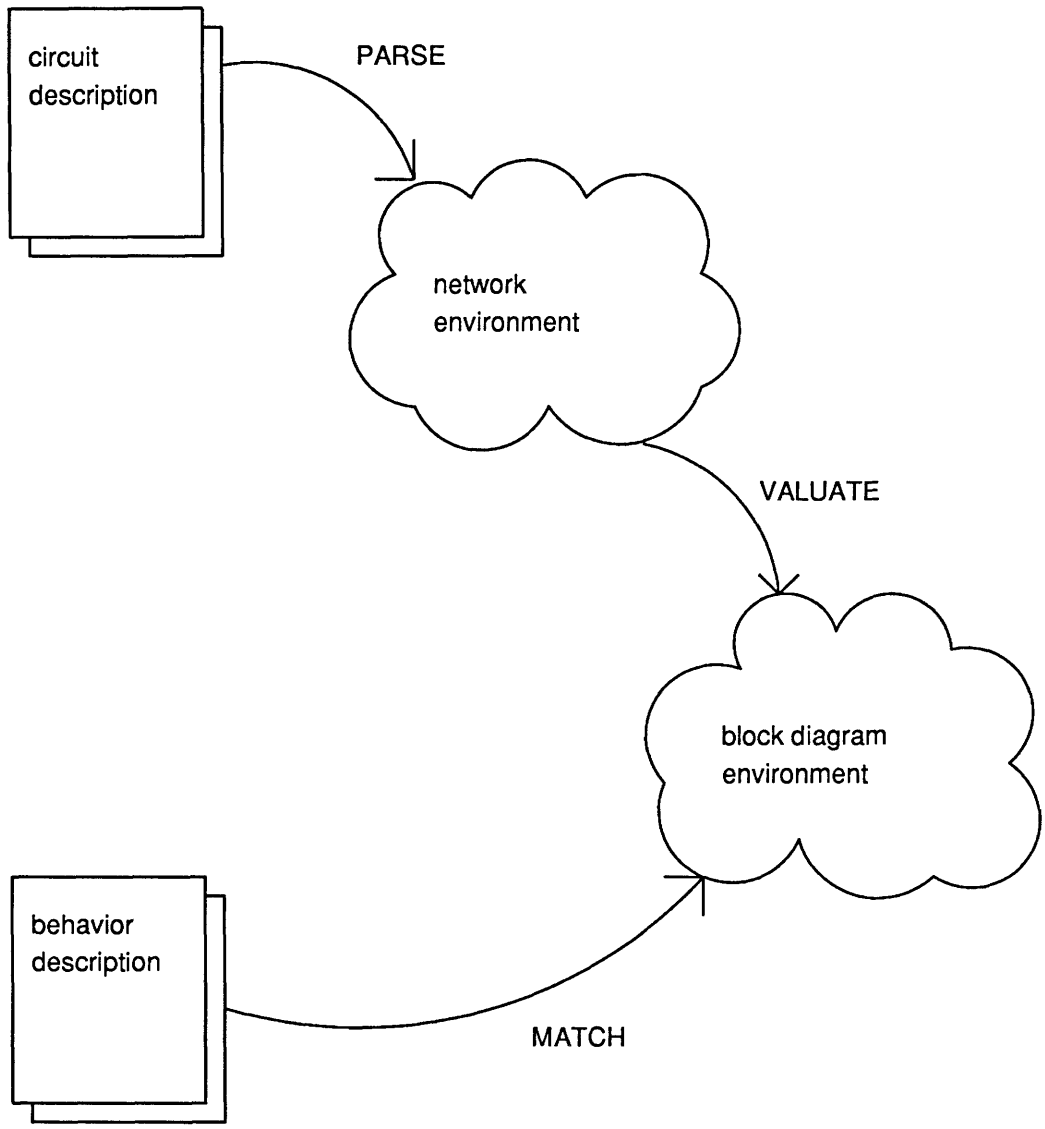

Figure 5.1: Semanticist's overall architecture

circuit. A block diagram environment contains bindings between cell names and block diagrams. Block diagrams are objects that contain behavioral information. They represent a computational process in the form of functional blocks with inputs and outputs, and with arrows from an output of one block to an input of an other. Once a parse tree has been formed in the network environment, a valuation function looks at it, performs circuit level checks that are not covered by the grammar, and produces behavioral information in the block diagram environment. Once that is done, the matcher goes through the behavioral description and traces the block diagrams to check whether the derived behavior corresponds with the specified one.

The following sections provide more details on each component of this architecture. They cover successively the circuit description, the behavioral description, the network environment, the block diagram environment, parsing, valuation, matching, and an example of a verification run. The discussion focuses on the most important objects that are used, and on the functionality of the most crucial procedures (what they take in and what they produce). The character of the most important objects and the functionality of the major procedures form this framework of the verification program. Once the framework is established, everything else falls pretty much in place. Finding a framework that is able to do the job, is really what the implementation exercise is all about.

5.1 The Circuit Description

Circuits are described in a hierarchical way. Each cell in the hierarchy is described by a list of primitive modules, or a list of subcircuits, or both. The primitive modules in our grammar are of type "p-trans", "n-trans", "in", "out", "supply" or "clock". For each primitive module, the user gives the names of the nets that connect to the pins of the module. He gives a list of net names that correspond to pins 0, 1, etc. in that order. For instance, a pMOS transistor with its gate (pin 0) connected to "in", its source (pin 1) connected to "vdd" and its drain (pin 2) connected to "out", is described as:

(p-trans (in vdd out))

A net name may be accompanied by a capacitance value. For the modules of type p-trans or n-trans, a W/L value may be specified. The modules in and out have one pin that connects to an external input and an external output respectively. A supply and clock module are inserted automatically by the system. They are connected to the nets with names "vdd", "gnd", "phi1", "-phi1", "phi2" and "-phi2".

Each element in a list of subcircuits consists of a cell name and a renaming list. For instance, if a cell "invertor" is instantiated with its "in" node renamed to "a", and its "out" node renamed to "not-a", the corresponding element in the subcircuit list reads:

(invertor ((in a) (out not-a)))

Below, we give a circuit description for an 8 bit transmission gate adder (with some repetitive parts omitted). The corresponding behavioral specification will be given in the next section, and the output of Semanticist on these input descriptions will be shown in the last section of this chapter.

Figures 5.2, 5.3 and 5.4 show the cells exor, mux and add-bit (one bit-slice of the adder). You will notice that add-bit contains a number of redundant invertors. These were introduced to get rid of loops over DCN's, which can't be handled by the grammar. For the same reason, we had to introduce two nodes for the carry bit.

```
(defc 'invertor
  '((primitives (p_trans (in vdd out) 1)
                (n_trans (in gnd out) 1))))

(defc 'inv%
  '((primitives (p_trans (in vdd out))
                (n_trans (in gnd out)))))

(defc 'exor
  '((primitives (p_trans (a b a_exor_b) 1)
```

```
                (n_trans (not_a b a_exor_b) 1)
                (p_trans (b a a_exor_b) 1)
                (n_trans (b not_a a_exor_b) 1))
    (subcircuits (invertor ((in a) (out not_a))))))

(defc 'mux
  '((primitives (p_trans (-contr x out))
                (n_trans (contr x out))
                (p_trans (contr y out))
                (n_trans (-contr y out)))))

(defc 'add-bit
  '((subcircuits (invertor ((in ain) (out -a)))
                 (invertor ((in bin) (out -b)))
                 (invertor ((in -a) (out a)))
                 (invertor ((in -b) (out b)))
                 (exor ())
                 (invertor ((in a_exor_b) (out -a_exor_b)))
                 (inv% ((in bin) (out -b%)))
                 (invertor ((in c) (out -c)))
                 (mux ((x -c) (y c%) (contr -a_exor_b) (-contr a_exor_b)
                       (out -sum)))
                 (mux ((x -b%) (y -c) (contr -a_exor_b) (-contr a_exor_b)
                       (out -cout)))
                 (invertor ((in -cout) (out cout)))
                 (inv% ((in -cout) (out cout%)))
                 (invertor ((in -sum) (out sum))))))

(defc 'adder
  '((subcircuits (add-bit ((a a1) (-a -a1) (ain ain1)
                           (b b1) (-b -b1) (-b% -b%1) (bin bin1)
```

```
                    (c c1) (c% c1%) (-c -c1)
                    (cout c2) (cout% c2%) (-cout -c2)
                    (sum sum1) (-sum -sum1)
                    (a_exor_b a_exor_b_1) (-a_exor_b -a_exor_b_1)))

                .
                .
                .

          (add-bit ((a a8) (-a -a8) (ain ain8)
                    (b b8) (-b -b8) (-b% -b%8) (bin bin8)
                    (c c8) (c% c8%) (-c -c8)
                    (cout c9) (cout% c9%) (-cout -c9)
                    (sum sum8) (-sum -sum8)
                    (a_exor_b a_exor_b_8) (-a_exor_b -a_exor_b_8)))
          (invertor ((in -cin) (out c1)))
          (inv% ((in -cin) (out c1%))))
(primitives (in (ain1)) (in (bin1)) (out (sum1))

                .
                .
                .

          (in (-cin)) (out (c9)) (out (c9%)))))
```

Reading the circuit description is the first thing that Semanticist does. When reading it, it builds up a "circuit environment" in which cell names are bound to cell descriptions, just as they appear in the input file.

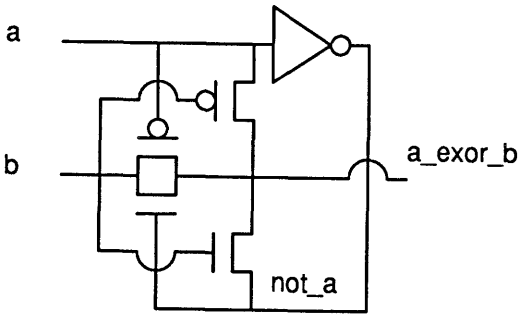

Figure 5.2: Cell "exor"

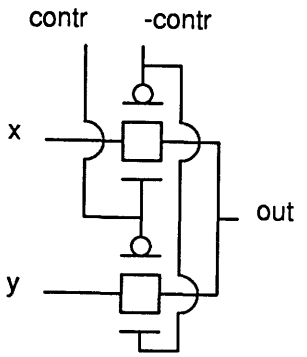

Figure 5.3: Cell "mux"

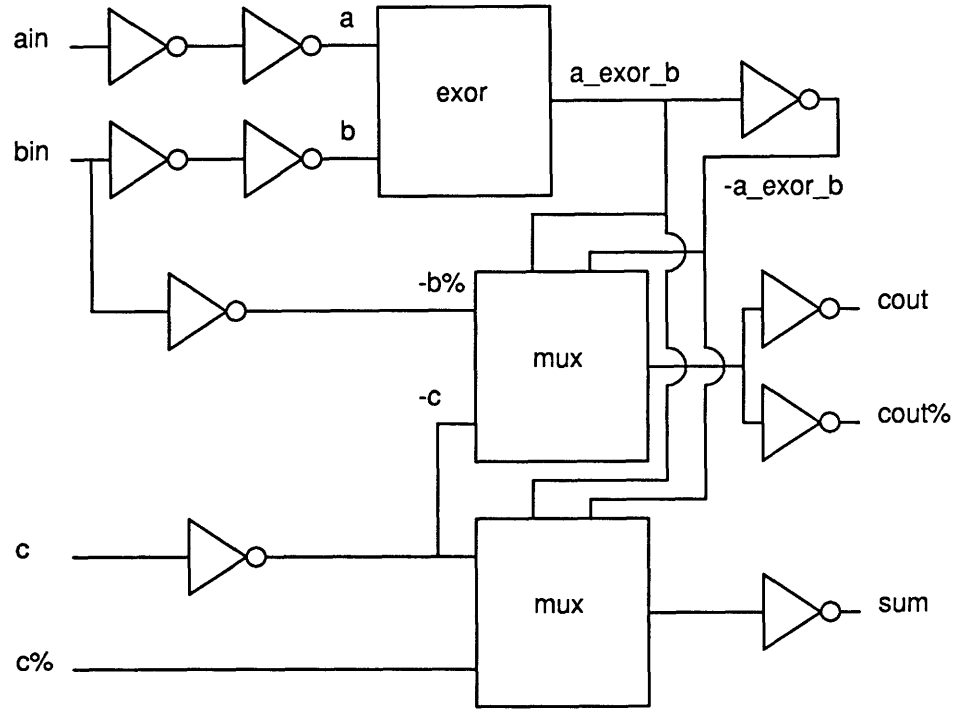

Figure 5.4: Cell "add-bit"

5.2 The Behavioral Description

The behavioral description is again hierarchical. The hierarchy is not necessarily the same as the one in the circuit description. All the elements in the behavioral hierarchy should have a corresponding element in the circuit hierarchy, but not vice versa. A circuit description consisting of an *and* cell with a *nand* and an *invertor* subcell, can have a behavioral specification consisting of only an *and* cell.

The behavior of each cell is described by a Lisp function (a Scheme function to be precise, [RCA* 86]). Lisp, with its applicative order evaluation, [1] was sufficient because we restricted ourselves to combinatorial circuits. If you want to handle sequential circuits, you need a functional language with some kind of lazy evaluation. [2] Sequential circuits are to be described by functions with take in streams and produce streams. A stream is an infinite list of values. In order to implement streams, you need a lazy evaluation scheme so that the values in a stream are computed as they are needed. For more information on this point, and on functional languages for behavioral specifications in general, the reader is referred to chapter 7 in [Weise 86]. Our behavior language is very similar to Weise's.

The behavioral description language is a subset of Scheme. It has six data types: booleans, logicals, floats, vectors, lists and procedures.

1. Booleans can have two values: 0 or 1. A boolean is a digital abstraction of a voltage level. There are three primitive procedures which take booleans and produce a boolean: `and`, `or` and `inv`. `not` doesn't work on 0's and 1's in Scheme, so we used `inv` instead.

2. A logical has a `true` or a `false` value. Logicals are used by the Scheme interpreter when it runs a behavioral description. It uses them to make decisions (for instance to decide when to stop with a recursion). `logic` is a primitive procedure that converts a boolean into a logical.

[1] In applicative order evaluation, the arguments to a procedure are evaluated first before the body of the procedure is executed.

[2] Lazy evaluation means that the arguments to a procedure are not evaluated first. At the latest, they are evaluated when they are needed in the body of the procedure.

3. Something of type float can have only one value: `float`. `float` describes a high impedance state. Only one primitive procedure takes floats as inputs: `join`. `join` takes an arbitrary number of arguments. If all of them are `float`, the result is `float`. If all the non `float` arguments are 0, the result is 0. If they are all 1, the result is 1.

4. A vector is a sequence of booleans and floats. It is used to describe a set of values which is accessed in parallel by a circuit, or produced in parallel. Vectors are constructed with the primitive procedure `vector`, which takes an arbitrary number of arguments. A component of a vector is selected with the primitive procedure `vector-ref`, which takes a vector and an index.

5. Lists are also sequences of booleans and floats. They are used to describe a set of values that are accessed or produced one after the other (e.g. the input bits of a carry ripple adder). Lists are constructed with the primitive procedure `lst-accum`, which takes a list and an element, and produces a new list with the new element in the back of the list. The selectors for lists are : `car`, which selects the first element, `cdr`, which selects the sublist which doesn't have the first element, `last`, which selects the last element of the list, and `without-last`, which selects the sublist which doesn't have the last element. The predicate `null?` checks whether the list is empty, and returns a logical.

6. Procedures are functions from one or more of the things above to one or more of the things above. They are produced by a lambda form (see further).

There are five special forms (i.e. forms which are not procedure applications): define forms, lambda forms, let forms, cond forms and if forms.

1. A lambda form is of the form:

```
(lambda (x y) expression)
```

   It produces a procedure.

2. A define form establishes a binding between a name and a procedure.

```
(define (fun x y) expression)
```

   is a shorthand for

```
(define fun (lambda (x y) expression))
```

3. A let form binds names to intermediate values in a computation.

4. A cond form is of the form

```
(cond ((condition1 result1) (condition2 result) ... ))
```

   It evaluates to the value of `result1` if `condition1` is true, to the value of `result2` if `condition2` is true, and so on. A cond form describes the behavior of multiplexer structures in a circuit.

5. An if form is of the form

```
(if condition result1 result2)
```

   It evaluates to the value of `result1` if `condition` holds, and to the value of `result2` otherwise. If forms that correspond to multiplexer structures in the circuit are converted to cond forms before matching. Only ones that are used to check when a recursion comes to an end (i.e. ones of the form `(if (null? xxx-lst) yyy zzz)`) are maintained. These forms have nothing equivalent in the circuit description.

As an example, here is the behavioral description of the 8 bit adder, the circuit description of which was given in the former section.

```
(defb 'inv%
  '(lambda (in)
     (inv in)))

(defb 'exor
  '(lambda (a b)
     (if (not (logic a))
         b
         (inv b))))

(defb 'mux
  '(lambda (contr -contr x y)
     (cond ((logic contr) x)
           ((logic -contr) y))))

(defb 'add-bit
  '(lambda (ain bin c c%)
     (let ((-a (inv ain))
           (-b (inv bin)))
       (let ((a-exor-b (exor (inv -a)
                             (inv -b))))
         (vector
          (inv  (mux (inv a-exor-b) a-exor-b (inv% bin) (inv c)))
          (inv% (mux (inv a-exor-b) a-exor-b (inv% bin) (inv c)))
          (inv (mux (inv a-exor-b) a-exor-b (inv c) c%)))))))

(defb 'adder
  '(lambda (ain-lst bin-lst -cin)
     (define (adder-aux ain-lst bin-lst c c% sum-lst)
       (if (null? ain-lst)
```

```
        sum-lst
        (let ((vec (add-bit (car ain-lst) (car bin-lst) c c%)))
          (adder-aux (cdr ain-lst) (cdr bin-lst)
                     (vector-ref vec 0)
                     (vector-ref vec 1)
                     (lst-accum sum-lst (vector-ref vec 2))))))
  (adder-aux ain-lst bin-lst (inv -cin) (inv% -cin) '()))))
```

Reading the behavioral description is the second thing Semanticist does (after reading the circuit description). While reading it, it adds bindings to three environments: the top level Scheme environment, the behavior environment and the block diagram environment. An environment is something that contains bindings between variables and values. If you evaluate a variable in a certain environment, you get the value that is bound to it in that environment (hence the name "environment").

All the environments that are talked about in this chapter contain bindings for the same cell names. The circuit environment binds the cell names to circuit descriptions, the network environment to network objects, the block diagram environment to block diagram objects, the behavior environment to behavior descriptions, and the top level Scheme environment (the "user-initial-environment") to evaluated behavior descriptions. These last bindings can be used to run the behavior descriptions on the Scheme interpreter.

When the behavior description is read, three of the environments get bindings: the behavior environment gets bindings of cell names to unevaluated behavior descriptions, the top level Scheme environment to evaluated behavior descriptions, and the block diagram environment to initialized block diagram objects.

5.3 The Network Environment

The network environment contains bindings between cell names and network objects. Before parsing, these objects just contain a flat network description. After

parsing, they have a parse tree on top of them. Network objects have pointers to module objects and net objects.

A **network object** can be accessed through the following procedures:

1. `(add-primitive-module module-type list-of-nets)`

   You specify a primitive module-type and list of nets to which the pins have to be connected, and the network object puts the module in its internal data structure.

2. `(add-net net-name)`

3. `(add-subnetwork subnetwork renaming-list)`

   The subnetwork is copied with certain nets renamed, as specified in the renaming list. The subnetwork should have been parsed by now. Only the top level modules after parsing are copied.

4. `(parse)`

   The network parses itself. It uses the concrete parser that GRASP produced for the circuit grammar in the previous chapter.

5. `(give-top-modules)`

   This returns the top modules of the network. The network should have been parsed by now.

Module objects contain the obvious information. They can give the nets to which the module is connected, the type of the module, the production rule (if any) according to which they emerged, the submodules of the module (if it emerged through application of a production rule), its model (if it is a leaf module in a certain network, copied from an other network), the name of the network to which it belongs, and a W/L value (if any). A **Net object** knows its name, the modules to which it is connected, its superior, inferior and adjacent bundles, and its capacitance (if any).

The binding between cell names and networks is established in a table which, for each network, also contains the parent cells in the circuit hierarchy.

5.4 The Block Diagram Environment

The block diagram environment contains bindings between cell names and block diagram objects. A block diagram object can be visualized as a set of blocks representing functions with arrows between them. It is made up of net objects which know to which other net objects they are inputs, and how their own values are a function of other nets.

A **block diagram object** can be accessed through the following procedures:

1. `(add-function net-name lambda-form)`

   `lambda-form` gets attached to the net object corresponding with `net-name`, and the net objects corresponding to the formal parameters of the lambda form get pointers to the object corresponding with `net-name`.

2. `(give-lambda-form net-name)`

   This returns the lambda-form associated with net.

3. `(give-net net-name)`

   This returns the net object corresponding with `net-name`. Each block diagram object has its own bindings between net names and net objects.

4. `(insert-race-constraints inputs outputs dyn-type)`

   This says that there is a dynamic gate of type `dyn-type` between `inputs` and `outputs`. `dyn-type` is either `n` (the type of gates whose outputs are precharged high) or `p` (precharged low). The diagram object will register proper constraints for avoiding input delay races in the dynamic gate.

   An input delay race can occur when an input to a dynamic gate is itself determined by a dynamic gate. Suppose an input i to an n-type dynamic gate is itself output of an n-type dynamic gate. Suppose the steady state value of i is 0. i only gets this 0 value after a precharging period, and if this period lasts longer than the precharging period for the next gate, the output of the next gate gets connected to ground for a short time, potentially loosing its

charge. In order to avoid this, we impose an alternation of gates that are precharged high and ones that are precharged low. (This is the NORA rule for avoiding input delay races [Goncalves 83].) If a node is only determined by static gates, it may be input to a dynamic gate of any type. If a node is precharged high (e.g. it is output of an n-type dynamic gate, or of a p-type dynamic gate followed by an invertor), it may only be connected to an input of a p-type dynamic gate. If it is output of a dynamic gate followed by a static gate other than an invertor, it may become precharged high in some circumstances and low in others. Such a node may not be connected to a dynamic gate any more.

The block diagram object takes care of this by attaching race constraints and race constraint challengers to net objects. There are two kinds of race constraints: `no-precharging-high` (for inputs of n-type dynamic logic) and `no-precharging-low` (for inputs of p-type dynamic logic). There are three types of race constraint challengers: `precharged-high`, `precharged-low` and `precharged-high-or-low`.

The grammar rules for combinatorial logic are such that the valuation function runs through the circuit from inputs to outputs. This means that, when race constraints are inserted at the inputs of dynamic logic, all the race constraint challengers have already been attached to those inputs. The constraints can then be checked against the constraint challengers.

5. `(add-to-parent-diagrams parent-diagram renaming-list)`

   Once the current block diagram has been matched with its behavioral specification, a function block which is an abstraction of its behavior has to be copied in all the diagrams above it in which it is instantiated. For this to be possible, the diagram has to know all the parent diagrams in which it should be copied (together with a list of net renamings for each copy).

6. `(set-inputs inputs)`

   `(set-outputs outputs)`

In order to be able to copy itself in its parent diagrams, the diagram should know what its inputs and outputs are. The inputs and outputs are filled in after the diagram has been matched. The outputs are the result of tracing the behavioral specification through the diagram.

7. `(matched)`

   This tells the diagram that its inputs and outputs are set, that its parent diagrams have been filled in, and that it can copy itself in its parent diagrams.

Each block diagram has its own environment with bindings between net names and net objects. A **net object** can be accessed with the following procedures.

1. `(give-diagram)`

   Give the diagram to which you belong.

2. `(add-lambda-form lambda-form)`

   If you don't have a lambda-form attached to you yet, attach this one. If you already have one, do one of the following:

   - If `lambda-form` has a boolean expression as body, check whether it is tautologous with the one you already have attached to you.
   - If `lambda-form` has a conditional statement as body, then check whether the lambda-form you already have also has a conditional as body. If so, just add to clauses of the new conditional to it, and add the conditions of these clauses to your list `exclusive-1-constraints`. This list contains nets of which exactly one should have a 1 value at any time.

     This case occurs if there is a set of pass transistors that feed into the same output. Each pass transistor is valuated to a lambda-form with a conditional as body. All these lambda forms are added to the same net object. The net object takes care of accumulating all the conditionals to form one big expression, and it builds up a logic constraint which ensures that, at any time, exactly one pass transistor passes a value to the output.

- If `lambda-form` has neither a boolean expression nor a conditional as body, flag an error.

3. `(add-to-outputs output)`

   Add `output` to your list of outputs. This list contains nets which have a lambda-form with the current net in its formal parameters.

4. `(give-lambda-form)`

5. `(give-outputs)`

6. `(add-to-input-instances input-instances)`

   `(add-to-output-models output-models)`

   `(give-input-instances input-instances)`

   `(give-output-models output-models)`

   These features make it possible to walk through the hierarchy of block diagrams as though it was one flat block diagram. This is needed to check logic constraints. In order to check whether exactly one net in a list of nets has a 1 value at any given time, you need to find common inputs to all these nets, and express the functions of all the nets in terms of these inputs. The only form of walking through the block diagrams that is needed for this is walking backwards from outputs to inputs. This is exactly what is supported by the four procedures above. If you have an input at the edge of some block diagram, you have pointers to the instances of this input higher up in the hierarchy, so that you can go further left. Once you found a block to which your net is an output, you can go down again by using pointers to the models of the output (i.e. the nets of which the output is an instance).

7. `(set-race-constraint race-constraint)`

   `(set-race-constraint-challenger race-constraint-challenger)`

   `(check-race-constraint)`

   These procedures were explained above.

8. `(send-exclusive-1-constraints)`

   This causes the net-object to put its list `exclusive-1-constraints` (see above) in a global list of logical constraints. These constraints can be verified after matching, when the hierarchy of block diagrams is complete, so that you can walk through the hierarchy as described above.

   In the current implementation of Semanticist, only the most simple logic constraints are verified, namely exclusive-1-constraints for two nets (i.e. constraints which say that exactly one of two nets should have a 1 value at any given time). Checking logic constraints in general is extremely hard, and not really necessary to prove that a circuit matches with a behavioral specification. The user can always simulate his behavioral description to see whether it is correct.

   Here is why exclusive-1-constraints are so hard to check. In order to find a set of common inputs to the nets (of which exactly one should have a 1 value), you need to look at the trees which originate at each net, and point, for each net, to inputs at deeper and deeper levels. To find a common set of inputs, you need to look at all combinations of cuts in the trees, and there are exponentially many of them.

5.5 Parsing

The function `parse` takes a description of one circuit cell, creates a network object for that cell, fills in all the primitive modules and subnetworks, and tells the network object to parse itself. Every time it needs a subnetwork that hasn't been parsed yet, it calls `parse` recursively for the subcell, and copies the top level modules in the current network. A subnetwork that is used several times in the circuit is only parsed once. To parse the whole circuit, it suffices to call `parse` on the top cell. If parsing is done, the system checks whether the top modules consist of one CB type module and a number of in and out modules (representing external inputs and outputs). If this is the case, Semanticist goes on with valuation. If not, it flags a

syntax error and prints out all the top modules the parser ended up with.

Note that the hierarchy of the parse tree differs from the one of the input description. The hierarchy of the parse tree is based on a partitioning into DCN's, whereas the input description may have any partitioning. Having those two different hierarchies around is an essential ingredient of circuit verification. It's because the user partitions a circuit into parts that are not the natural units with regard to circuit correctness that he makes mistakes. Circuit level bugs have often to do with electrical interactions across cell boundaries. For instance, if a multiplexer structure consisting of an nMOS pass transistors followed by an invertor with a feedback pMOS transistor for level restoration is contained in one cell, and the static gates that produce the inputs to the multiplexer in another cell, there is a good chance that the designer didn't think of the W/L effects when he designed the static gates at the inputs.

5.6 Valuation

`valuate` is a recursive function which takes in a module, produces a valuation, and (possibly) modifies the block diagram environment. It calls `valuate` recursively on the children modules in the parse tree, and uses the the resulting valuations to produce its own valuation, and possibly to add information to the block diagram environment. Again, it suffices to call `valuate` on the top level CB type module.

There are two kinds of valuations: handled valuations and valuations that are not handled. The valuations that are passed up from the bottom of the parse tree are ones that are not handled. They accumulate a behavioral function that is eventually going to be inserted into the block diagram environment. Once this behavioral information is filled in in the block diagrams, handled valuations are passed up. They contain only information that is necessary for checking well-formedness constraints.

A valuation gets handled if one of the following conditions applies:

1. The module for which the valuation contains a behavioral description, covers more than one leaf diagram. The chunk of behavioral information doesn't

correspond to any one chunk in the specification any more, so the parts of the behavioral information are filled in in the leaf diagrams.

2. The behavioral description in the valuation is a mixture of boolean and non boolean expressions. It makes no sense to combine these expressions into one chunk. There won't be any corresponding chunk in the specification. Therefore, the parts are inserted in the corresponding block diagram.

3. The valuation function decides to handle the valuation.

As long as neither of these conditions apply, the behavioral information is accumulated without writing anything into the block diagrams. This is done to allow for flexible matching of boolean portions in the circuit with boolean portions in the behavioral specification. A boolean portion in the derived behavior is inserted in the block diagrams as one entity, which will be matched with a corresponding expression in the specification by tautology checking. A circuit portion consisting of a *nand* gate followed by an *invertor* is transformed into one combined function, so that it can be matched with a single *and* function. If separate functions would have been created for the *nand* and the *invertor*, the matcher would not be able to find corresponding portions in the behavioral specification.

Here is the exact contents of handled and non handled valuations. A non handled valuation consists of a function, a list of diagrams, a capacitance and a W/L value.

1. A function consists of a renamed lambda form and a renamed net. A renamed lambda form is just like a normal lambda form, except that it has renamed nets instead of normal net names. Renamed nets had to be introduced because the hierarchy of the behavioral description can be sparser than the hierarchy of the circuit description. This difference in hierarchy implies that a cell in the circuit description may not have an equivalent in the behavior description. The behavioral information for that cell has to go to the diagrams immediately above it. To identify a net in such a cell, we use a renamed net: a list of pairs consisting of a diagram and a name of the net in that diagram. There is one such pair for each diagram immediately above the cell.

So, a function consists of a renamed lambda-form and a renamed net. The renamed net represents the node whose value is determined by the renamed lambda form.

2. The list of diagrams in a non handled valuation, is just a list of all the diagrams in which the behavioral information is to be inserted. It contains all the diagrams in the renamed net in the function component of the valuation.

3. The capacitance slot is used to accumulate capacitances of transistor branches in dynamic logic. These are used to make sure that the output of the dynamic logic is not corrupted by charge sharing. The output capacitance is required to be larger than some multiple of the total capacitance with which it can share charge.

4. The W/L slot accumulates an equivalent W/L value of a transistor branch. This value is obviously used to check for W/L constraints.

A handled valuation has only components that are necessary to check for circuit level constraints. It consists of a list of inputs, a list of outputs, a capacitance and a W/L value.

- The inputs and outputs are the global inputs and outputs of the module that has been valuated. They are necessary to issue and check race constraints. If some of the inputs are dynamic (i.e. precharged high, or precharged low, or sometimes precharged high and sometimes precharged low), the race constraint challengers at the outputs are set to the proper values (see also under net objects in the section on the block diagram environment). If the output already has a `precharged-high` or `precharged-low` challenger, nothing happens. If it hasn't, and if inputs and outputs contain each one element and the behavior between them is inversion, the challenger of the output is set to the complement of the one at the input. In all other cases, the challenger at the output is set to `precharged-high-or-precharged-low`.

  The inputs and outputs are maintained up to the level where a DCN disappears into a CL module. They are used at the time an internal structure of the

third production rule for module-type CL is encountered (see section 4.2.5). Roughly speaking, this rule combines a CL module in series with a DCN module into one CL module. It's at that time that the net objects corresponding to the inputs of the DCN are triggered to check their race constraints against their race constraint challengers, and it's also then that race constraint challengers are propagated from the inputs of the DCN to the outputs. This is the right time for these operations because the grammar allows an order of valuation in which modules at the external inputs of the whole circuit are valuated first and following modules next. This order ensures that, when a DCN in series with a CL is valuated, the inputs of the DCN have the right race constraints and race constraint challengers.

- The capacitance and W/L value are used for checking charge sharing constraints and W/L constraints, as in non handled valuations.

So far for the general functionality of the valuation function. Within this framework, specialized valuation functions do their own specific thing. The general `valuate` function dispatches to specialized valuation functions for each module-type, and each such function does different things for different forms of the internal structure of the module.

Below is an excerpt from the valuation function for module-type N*, a parallel/series combination of nMOS transistors. The specialized valuation function for this module-type has three cases corresponding to the three production rules for it: one for the case it has an nMOS transistor as internal structure, one for the case where it has a parallel combination of two N* modules as internal structure, and one for a series combination. One other possibility is that the module is a copy of a top level module in another network.

What `N*-valuate` does in case of a parallel combination as internal structure is given in full. First it applies `val-accum` to the two submodules. `val-accum` computes the valuations of the submodules, and accumulates the result. If the combination of submodules is crossed by a diagram boundary, it handles the valuations automatically. If `val-acc`, the accumulation of valuations that `val-accum`

returned, contains handled valuations, `N*-valuate` constructs a handled valuation, consisting of a list of inputs, a list of outputs, a combined capacitance and a combined W/L value. If not, it combines the behaviors of the submodules, and produces a non handled valuation from that. The behavior of an N* module is represented by a conditional statement. The conditionals of the submodules are ored together in the proper way. This combination forms a renamed lambda-form (written as lambda-form% in the program), which, together with a renamed net (written as net%), namely the common drain of the two submodules, forms a function. This function, combined with a list of diagrams, is used to construct a valuation. The capacitance and W/L slot in that valuation function are initially empty, but they are filled in by the commands that follow. When that is done, `N*-valuate` returns the resulting valuation `val`.

```
(define (N*-valuate module)
  (cond ((N*_from_trans? module)
         ... )
        ((N*_from_paral? module)
         (let ((val-acc (val-accum
                         ((module 'give-submodule) 0)
                         ((module 'give-submodule) 1))))
           (if (handled? val-acc)
               (let ((inputs (merge (val-inputs (get-val val-acc 0))
                                    (val-inputs (get-val val-acc 1))))
                     (outputs (val-outputs (get-val val-acc 0)))
                     (cap (cap-add (val-cap (get-val val-acc 0))
                                   (val-cap (get-val val-acc 1))))
                     (W/L (W/L-add (val-W/L (get-val val-acc 0))
                                   (val-W/L (get-val val-acc 1)))))
                 (make-handled-valuation inputs outputs W/L cap))
               (let ((val
                      (make-valuation
                       (make-function
```

```
                    (or-cond-lambdas%
                     (fun-lambda-form% (val-function
                                        (get-val val-acc 0)))
                     (fun-lambda-form% (val-function
                                        (get-val val-acc 1))))
                    (fun-net% (val-function (get-val val-acc 1))))
                   (get-diagrams val-acc))))
          (val-add-cap val (val-cap (get-val val-acc 0)))
          (val-add-cap val (val-cap (get-val val-acc 1)))
          (val-set-W/L val
                       (W/L-add (val-W/L (get-val val-acc 0))
                                (val-W/L (get-val val-acc 1))))
          val))))
((N*_from_series? module)
 ... )
((copy? module)
  ...)
(else (error ... ))))
```

After the valuation function returns from valuating the top level CB module, the following things have been done:

1. All W/L constraints have been verified.

2. All charge sharing constraints have been verified.

3. All race constraints have been verified.

4. All logic constraints in pass transistor logic have been accumulated.

5. Complementarity of the pull up and pull down branches in complementary static logic has been checked for.

6. The block diagrams have been partially filled in with behavioral information. What has been filled in is the "bottom level" behavioral information. To complete the block diagrams, lower level diagrams have to be instantiated as one function in their parent diagrams. This is done each time a lower level diagram is matched with the corresponding description in the behavioral specification. When a block diagram is complete with "bottom level" behavioral information and function blocks describing lower level diagrams, it should match exactly with the description of the corresponding cell in the behavioral specification. The example at the end of this chapter will probably make this clearer (if it isn't already).

5.7 Matching

The key function of the matcher is `trace`, which looks at a portion of behavioral specification and follows the flow of it on one of the block diagrams. `trace` takes in an expression, a block diagram, a list of net bindings (bindings between net names and net objects in the block diagram), and a list of procedure bindings (bindings between procedure names and Lisp code), and returns a pointer in the diagram to a net object whose value is computed according to the expression that was given as argument. `trace` has exactly the shape of a Lisp interpreter, except that it returns pointers to net objects instead of values.

`trace` looks at the syntactic form of the expression that it received as argument and dispatches to a specialized trace function. For instance, the specialized trace function for a sequence of define forms and a main expression

```
(define procedure1 lambda-form1)
(define procedure2 lambda-form2)
...
main-expression
```

calls `trace` recursively on the main expression with the procedure bindings extended with the bindings that are given in the define forms. The specialized trace

function for a let form

```
(let ((variable1 expression1)
      (variable1 expression2)
      ...)
  body)
```

calls `trace` recursively on the body of the let form with the net bindings extended with bindings between the variables in the let form and the pointers that result from tracing the expressions corresponding with the variables. The specialized trace function for an application

```
(procedure argument1 argument2 ...)
```

calls `trace` recursively on the arguments, looks at the common outputs of the resulting net objects, and selects the one which has the right lambda form attached to it. In case of a boolean application, the trace function checks whether the arguments are boolean applications themselves, looks for the ultimate inputs, takes the common outputs of those inputs on the diagram, and performs tautology checking to select the right one.

The top level form in the behavioral specification of a cell name is a lambda-form:

```
(lambda (formal1 formal2 ...)
  body)
```

The matcher localizes net objects with names `formal1`, `formal2`, etc in the block diagram corresponding to the cell, and traces the body of the lambda form with the resulting net bindings. As a result of that, it gets a list of net objects that are the outputs of the block diagram. It communicates the list of inputs and the list of outputs that it found to the block diagram, and gives it a `(matched)` message, which causes the diagram to insert a function block (a summary of itself) in its parent diagrams.

In order for this to work, the cells have to be matched in the right order (lower level ones first), and the block diagrams need to know their parent diagrams. These

things are arranged for before matching begins. The block diagrams get pointers to their parents, and a queue of cell names is formed with low level diagrams first and high level ones last. The matcher then goes through an iteration in which it takes the first element from the queue and matches it.

The only thing that hasn't been done after that is checking the logic constraints associated with pass transistor structures. These logic constraints are accumulated in a global list. In the current implementation of Semanticist, only the simplest constraints are checked, namely ones that require exactly one of two nets to have a 1 value at any time. As mentioned in the discussion of net objects in the section on block diagrams, handling logic constraints in general is exponentially hard, and it isn't necessary to prove that a circuit has the behavior specified by the user. The user can always simulate his behavioral description to make sure it is correct.

5.8 An Example Verification Run

This section shows how the adder of section 1 and section 2 is verified. The first thing Semanticist does is reading the circuit and behavior descriptions. Once the behavior description is read, we can run it on the Scheme interpreter. To make this easier, we can write a layer on top of the `adder` function for type conversion between integers and lists of booleans.

```
(define int-adder
  (lambda (a b c)
    (bin->int (adder (int->bin a 8) (int->bin b 8) (inv c)))))
```

This leaves us with an adder function which operates on integers. Also, the carry bit is inverted because the adder circuit takes an inverted carry bit as input. We can run `int-adder` on some inputs to make sure it works properly.

```
(int-adder 7 8 0)
;Value: 15
```

```
(int-adder 23 32 0)
;Value: 55
```

Once we know the behavioral description is all right, we can go on and verify the design.

```
(verify 'adder)
```

After some time, the system says it's done with parsing the circuit, and goes on valuating the parse tree.

```
** Parsing done **
```

The parse tree has also been checked by now to consist of one top level CB module, and a number of in and out modules (representing external inputs and outputs). If the parse tree wasn't correct, the system would have printed all the top level modules that it ended up with. If the user can't locate the error in the circuit with that information, he can verify subcircuits first, make sure that they are correct, and move on gradually to higher levels in the circuit hierarchy.

In our case the parse tree was correct and Semanticist went on valuating the top level CB module. It recursively calls the valuation function for the children on the CB module, and so on. When it returns form valuating the CB module, it has traversed the whole parse tree. At that time, it says:

```
** Valuation done **
```

There are no violations of W/L constraints, charge sharing constraints, race constraints or complementarity constraints in our circuit. If there were, the system would have given an error message by now.

Semanticist now starts with the matching part. It matches the lowest level cell first, and higher level cells later on. Before it matches a cell, it prints out the behavioral information in the corresponding block diagram.

The first cell that is matched is the exor cell.

```
EXOR

NOT_A  lambda-form:(LAMBDA (A) (NOT A))
  input to: (A_EXOR_B)

A  lambda-form:NONE
  input to: (NOT_A A_EXOR_B)

B  lambda-form:NONE
  input to: (A_EXOR_B NOT_B)

A_EXOR_B  lambda-form:(LAMBDA (A NOT_A B NOT_B) (COND (A NOT_B) (NOT_A
B)))
  input to: ()

NOT_B  lambda-form:(LAMBDA (B) (NOT B))
  input to: (A_EXOR_B)

EXOR matched
```

The behavioral description in the block diagram matches with the one supplied by the user. If the system would have failed to match the two descriptions, it would have said at which point in the user supplied description it couldn't find anything that matches with that in the block diagram. The behavioral contents of the block diagram printed out above makes it easy to find the bug.

Semanticist goes on matching the other cells now.

```
INV%

IN  lambda-form:NONE
  input to: (OUT)
```

```
OUT  lambda-form:(LAMBDA (IN) (NOT IN))
  input to: ()

INV% matched


MUX

-CONTR  lambda-form:NONE
  input to: (OUT)

CONTR  lambda-form:NONE
  input to: (OUT)

Y  lambda-form:NONE
  input to: (OUT)

OUT  lambda-form:(LAMBDA (CONTR X -CONTR Y) (COND (-CONTR Y) (CONTR X)))
  input to: ()

X  lambda-form:NONE
  input to: (OUT)

MUX matched


ADD-BIT

-A  lambda-form:(LAMBDA (AIN) (NOT AIN))
  input to: (A)
```

```
A  lambda-form:(LAMBDA (-A) (NOT -A))
  input to: (A_EXOR_B)

-B  lambda-form:(LAMBDA (BIN) (NOT BIN))
  input to: (B)

-A_EXOR_B  lambda-form:(LAMBDA (A_EXOR_B) (NOT A_EXOR_B))
  input to: (-COUT -SUM)

-C  lambda-form:(LAMBDA (C) (NOT C))
  input to: (-COUT -SUM)

-B%  lambda-form:(LAMBDA (BIN) (INV% BIN))
  input to: (-COUT)

-SUM  lambda-form:(LAMBDA (C% -C A_EXOR_B -A_EXOR_B) (MUX C% -C A_EXOR_B
-A_EXOR_B))
  input to: (SUM)

-COUT  lambda-form:(LAMBDA (-C -B% A_EXOR_B -A_EXOR_B) (MUX -C -B% A_EXOR_B
-A_EXOR_B))
  input to: (COUT COUT%)

B  lambda-form:(LAMBDA (-B) (NOT -B))
  input to: (A_EXOR_B)

AIN  lambda-form:NONE
  input to: (-A)

A_EXOR_B  lambda-form:(LAMBDA (B A) (EXOR B A))
  input to: (-A_EXOR_B -COUT -SUM)
```

```
BIN  lambda-form:NONE
  input to: (-B -B%)

C  lambda-form:NONE
  input to: (-C)

SUM  lambda-form:(LAMBDA (-SUM) (NOT -SUM))
  input to: ()

COUT  lambda-form:(LAMBDA (-COUT) (NOT -COUT))
  input to: ()

C%  lambda-form:NONE
  input to: (-SUM)

COUT%  lambda-form:(LAMBDA (-COUT) (INV% -COUT))
  input to: ()

ADD-BIT matched


ADDER

-CIN  lambda-form:NONE
  input to: (C1 C1%)

C1  lambda-form:(LAMBDA (-CIN) (NOT -CIN))
  input to: (C2 C2% SUM1)

BIN8  lambda-form:NONE
```

```
  input to: (C9 C9% SUM8)

AIN8  lambda-form:NONE
  input to: (C9 C9% SUM8)

AIN7  lambda-form:NONE
  input to: (C8 C8% SUM7)

AIN6  lambda-form:NONE
  input to: (C7 C7% SUM6)

AIN5  lambda-form:NONE
  input to: (C6 C6% SUM5)

AIN4  lambda-form:NONE
  input to: (C5 C5% SUM4)

AIN3  lambda-form:NONE
  input to: (C4 C4% SUM3)

AIN2  lambda-form:NONE
  input to: (C3 C3% SUM2)

AIN1  lambda-form:NONE
  input to: (C2 C2% SUM1)

BIN7  lambda-form:NONE
  input to: (C8 C8% SUM7)

BIN6  lambda-form:NONE
  input to: (C7 C7% SUM6)
```

```
BIN5  lambda-form:NONE
  input to: (C6 C6% SUM5)

BIN4  lambda-form:NONE
  input to: (C5 C5% SUM4)

BIN3  lambda-form:NONE
  input to: (C4 C4% SUM3)

BIN2  lambda-form:NONE
  input to: (C3 C3% SUM2)

BIN1  lambda-form:NONE
  input to: (C2 C2% SUM1)

C1%  lambda-form:(LAMBDA (-CIN) (INV% -CIN))
  input to: (C2 C2% SUM1)

C8%  lambda-form:(LAMBDA (C7% C7 BIN7 AIN7) (VECTOR-REF (ADD-BIT C7% C7
BIN7 AIN7) 1))
  input to: (C9 C9% SUM8)

C8  lambda-form:(LAMBDA (C7% C7 BIN7 AIN7) (VECTOR-REF (ADD-BIT C7% C7
BIN7 AIN7) 0))
  input to: (C9 C9% SUM8)

C7%  lambda-form:(LAMBDA (C6% C6 BIN6 AIN6) (VECTOR-REF (ADD-BIT C6% C6
BIN6 AIN6) 1))
  input to: (C8 C8% SUM7)
```

```
C7  lambda-form:(LAMBDA (C6% C6 BIN6 AIN6) (VECTOR-REF (ADD-BIT C6% C6
BIN6 AIN6) 0))
  input to: (C8 C8% SUM7)

C6%  lambda-form:(LAMBDA (C5% C5 BIN5 AIN5) (VECTOR-REF (ADD-BIT C5% C5
BIN5 AIN5) 1))
  input to: (C7 C7% SUM6)

C6  lambda-form:(LAMBDA (C5% C5 BIN5 AIN5) (VECTOR-REF (ADD-BIT C5% C5
BIN5 AIN5) 0))
  input to: (C7 C7% SUM6)

C5%  lambda-form:(LAMBDA (C4% C4 BIN4 AIN4) (VECTOR-REF (ADD-BIT C4% C4
BIN4 AIN4) 1))
  input to: (C6 C6% SUM5)

C5  lambda-form:(LAMBDA (C4% C4 BIN4 AIN4) (VECTOR-REF (ADD-BIT C4% C4
BIN4 AIN4) 0))
  input to: (C6 C6% SUM5)

C4%  lambda-form:(LAMBDA (C3% C3 BIN3 AIN3) (VECTOR-REF (ADD-BIT C3% C3
BIN3 AIN3) 1))
  input to: (C5 C5% SUM4)

C4  lambda-form:(LAMBDA (C3% C3 BIN3 AIN3) (VECTOR-REF (ADD-BIT C3% C3
BIN3 AIN3) 0))
  input to: (C5 C5% SUM4)

C3%  lambda-form:(LAMBDA (C2% C2 BIN2 AIN2) (VECTOR-REF (ADD-BIT C2% C2
BIN2 AIN2) 1))
  input to: (C4 C4% SUM3)
```

```
C3  lambda-form:(LAMBDA (C2% C2 BIN2 AIN2) (VECTOR-REF (ADD-BIT C2% C2
BIN2 AIN2) 0))
  input to: (C4 C4% SUM3)

C2%  lambda-form:(LAMBDA (C1% C1 BIN1 AIN1) (VECTOR-REF (ADD-BIT C1% C1
BIN1 AIN1) 1))
  input to: (C3 C3% SUM2)

C2  lambda-form:(LAMBDA (C1% C1 BIN1 AIN1) (VECTOR-REF (ADD-BIT C1% C1
BIN1 AIN1) 0))
  input to: (C3 C3% SUM2)

C9  lambda-form:(LAMBDA (C8% C8 BIN8 AIN8) (VECTOR-REF (ADD-BIT C8% C8
BIN8 AIN8) 0))
  input to: ()

C9%  lambda-form:(LAMBDA (C8% C8 BIN8 AIN8) (VECTOR-REF (ADD-BIT C8% C8
BIN8 AIN8) 1))
  input to: ()

SUM8  lambda-form:(LAMBDA (C8% C8 BIN8 AIN8) (VECTOR-REF (ADD-BIT C8%
C8 BIN8 AIN8) 2))
  input to: ()

SUM7  lambda-form:(LAMBDA (C7% C7 BIN7 AIN7) (VECTOR-REF (ADD-BIT C7%
C7 BIN7 AIN7) 2))
  input to: ()

SUM6  lambda-form:(LAMBDA (C6% C6 BIN6 AIN6) (VECTOR-REF (ADD-BIT C6%
C6 BIN6 AIN6) 2))
```

```
  input to: ()

SUM5  lambda-form:(LAMBDA (C5% C5 BIN5 AIN5) (VECTOR-REF (ADD-BIT C5%
C5 BIN5 AIN5) 2))
  input to: ()

SUM4  lambda-form:(LAMBDA (C4% C4 BIN4 AIN4) (VECTOR-REF (ADD-BIT C4%
C4 BIN4 AIN4) 2))
  input to: ()

SUM3  lambda-form:(LAMBDA (C3% C3 BIN3 AIN3) (VECTOR-REF (ADD-BIT C3%
C3 BIN3 AIN3) 2))
  input to: ()

SUM2  lambda-form:(LAMBDA (C2% C2 BIN2 AIN2) (VECTOR-REF (ADD-BIT C2%
C2 BIN2 AIN2) 2))
  input to: ()

SUM1  lambda-form:(LAMBDA (C1% C1 BIN1 AIN1) (VECTOR-REF (ADD-BIT C1%
C1 BIN1 AIN1) 2))
  input to: ()

ADDER matched

** Matching done **
```

At this point the whole behavior has been matched. The only thing that remains to be done is handling the logic constraints. The system prints out all the logic constraints, and verifies the simple ones.

```
the-excl-1-constraints:
(-CONTR CONTR)
```

```
(CONTR -CONTR)
(A NOT_A)
** Logic constraints verified **
```

All the lists under `the-excl-1-constraints` should have exactly one net with a 1 value at any time. In the case of this circuit, they are all simple ones, which can be checked automatically.

Once that is done, the system ensures us:

```
Correct circuit: ADDER
```

Chapter 6

Results and Further Work

The result of this research is a working program, Semanticist. Semanticist is implemented in Scheme, a Lisp dialect [RCA* 86], on an HP workstation 9000/350 running UNIX. The Scheme code consists of about 5000 lines. Semanticist interfaces to GRASP for circuit parsing. GRASP is implemented in C. The portion of GRASP that is used by Semanticist contains about 6000 lines, and the grammar that I wrote for it takes about 3000 lines. Semanticist also interfaces to the Espresso program from UC Berkeley for tautology checking.

Section one of this chapter takes a close look at the quality of Semanticist. It discusses its performance, competence, and practicality as a CAD tool. Section 2 points to opportunities for further research.

6.1 Results

6.1.1 Performance

Table 6.1 contains run times of Semanticist on a number of circuits. The adder is the transmission gate adder that was verified in the previous chapter. It is based on a pass transistor exor.

The shifter is a variable logarithmic shifter with an 8 bit up section and an 8 bit down section. A multiplexer at the output chooses between one of these sections. Each section contains three stages: one which can shift the input over 1 bit, one

adder	2 bit version (72 transistors)	1 min. 20 s.
	4 bit version (140 transistors)	2 min. 00 s.
	6 bit version (208 transistors)	2 min. 50 s.
	8 bit version (276 transistors)	3 min. 40 s.
	16 bit version (548 transistors)	8 min. 30 s.
shifter	8 bit version (232 transistors)	11 min. 50 s.
	8 bit version (232 transistors)	15 min. 30 s.
ALU	2 bit version (126 transistors)	8 min. 20 s.
	4 bit version (228 transistors)	14 min. 50 s.

Table 6.1: Run times of Semanticist on a number of circuits

which can shift it over 2 bits, and one which can shift it over 4 bits. All these sections consist of pass transistor logic. By controlling these three stages, you can shift over any number of bits between 0 and 7. (For it to make sense to have an up and down section, the input vector should have been at least 16 bits long.) The first run time in the table is for input descriptions which have exactly the same hierarchy. The second run time covers the case where the behavioral hierarchy is sparser than the circuit hierarchy.

The ALU is a CMOS version of a Mead and Conway ALU. It has pass transistor blocks which produce propagate and generate signals, generalized static gates (PS type gates in my grammar) which produce the carry signal for the next bit based on the propagate and generate signals, and again pass transistor blocks for producing the result bits from the carry and propagate signals.

Although the efficiency of the overall verification strategy mattered a lot in my research, the implementation was not written with a concern for efficiency in all its details. The implementation exercise has served in the first place to clarify my ideas about circuit verification and denotational semantics. The result of it is a clear understanding of how denotational semantics can be applied to circuit verification.

There are two facts that account for the inefficiency of the current implementation.

1. Semanticist runs on interpreted Lisp. Compiling the code would give a speedup

factor of 5 to 10.

2. Although the backbone algorithm has a linear complexity, there are individual steps which, in the worst case, can take time proportional to the square of the circuit size. These operations give rise to a cubic term in the overall complexity function. There are two such operations:

   - Operations on renamed nets.

     A renamed net is a list of pairs consisting of a diagram and the name the net has in that diagram. If the hierarchy of the behavioral specification matches exactly with the one of the circuit description, each renamed net consists of only one pair. If not, a renamed net can have a length proportional to the circuit size. There are several operations on renamed nets (e.g. comparing whether two renamed nets identify the same net) which take time proportional to the square of the length of the renamed net. This accounts for the difference between the two run times for the shifter. When the behavioral representation was written in the same hierarchy as the circuit description, the run time went down from 15'30" to 11'50".

     As a way out, we can either try to find better representations to identify nets, and better operations on them, or present the user with a choice between efficiency and sparse behavioral descriptions.

   - Operations on the `inputs` and `outputs` slots in a handled valuation.

     A handled valuation contains a list of all the inputs of the module and all the outputs. These are necessary to check race constraints. They are maintained up to the level where a DCN disappears into a CL block. On the level of CL blocks, the inputs and outputs are not necessary any more. Again there are operations on these lists that take quadratic time (e.g. merging the input lists of two submodules). If there are large DCN's in the circuit, they have again a large impact on the overall verification time. This is the case in the shifter and the ALU in table 6.1, which both have DCN's that span over all bitslices.

> The solution here is to check race constraints on the syntactic level, i.e. to incorporate them into the grammar. This will be eased if we have a more flexible grammar formalism, one in which the parser can look inside modules that it already formed. This way, it is easy to distinguish an invertor from an other static gate, for instance.

The 8 bit shifter and the 4 bit ALU were also verified by DIALOG [Bolsens 88]. Bolsens reported a run time of 10' for the shifter and 4'20" for the ALU (in a personal communication). DIALOG is a mixture of an inefficient general purpose algorithm and fast heuristics for specific circuit configurations. This makes it difficult to compare DIALOG's run times on some specific circuits with Semanticist's. A lot depends on how much the system can rely on heuristics for a specific circuit. Furthermore, DIALOG doesn't check whether a circuit meets a behavioral specification.

Weise is very optimistic about the performance of Silica Pithecus, but (as far as I know from the literature) he never implemented it completely. He projects a run time of 51 seconds for a circuit of more than 1100 transistors, and one of 80 seconds for a circuit of 9000 transistors [Weise 86]. It's difficult to appreciate the value of these estimates in comparing with my run times. Also, it depends on the hierarchy of the input description whether the system suffers from combinatorial explosion or not.

I didn't find any run times for Gordon's logic based system in the literature. It is common knowledge, however, that logic based systems are very inefficient.

6.1.2 Competence

There are three issues with regard to competence in circuit verification. Does the system reject all incorrect circuits (within a simple switch level model)? Does the system accept all correct circuits (within a simple switch level model)? Is the system able to check whether the circuit meets a behavioral specification, and, if so, does the behavioral specification have to be procedural, or can it be declarative?

1. Does the system reject all incorrect circuits?

Semanticist does, as well as DIALOG, Silica Pithecus and Gordon's logic based system.

2. Does the system accept all correct circuits?

   This is where my system has some weaknesses that DIALOG, Silica Pithecus and Gordon's system don't have. The degree to which my system covers all correct circuits depends on the range space of the grammar. Section 4.3.1 of this thesis summarized the range space of the current grammar, and section 4.3.2 listed a number of circuit configurations that are not covered. In a nutshell, the grammar doesn't cover all cases of feedback over DCN's, it doesn't cover bypassing in pass transistor networks, it doesn't cover all conceivable non parallel/series combinations of transistors, and there is a small number of configurations that aren't covered in the grammar yet, but that could be.

   The current grammar comes fairly close to covering all correct circuits, and an even better coverage is possible if we develop a new parsing framework. Most of the imperfections of the current grammar can be overcome if we would have a parser which can look inside modules that it already formed, and which can modify portions of the parse tree that it already constructed.

3. How well does the system in checking whether a behavioral specification is met?

   Just like Silica Pithecus and Gordon's system, Semanticist checks whether a behavioral specification is met. DIALOG doesn't. Both Silica Pithecus and Semanticist need procedural specifications. Gordon's system can handle declarative specifications, but the efficiency penalty is very heavy, and it doesn't bring a great advantage from a practical viewpoint (as we will argue in the next subsection).

6.1.3 Practicality as CAD tool

One objection I could imagine against Semanticist from a practical viewpoint, is that it needs a procedural behavior description that mirrors the circuit description. First,

we did include some features that allow behavioral descriptions that are not perfect mirrors of the circuit: the correspondence between boolean portions of circuit and behavior is checked by tautology checking, and the hierarchy of the behavioral description is allowed to be sparser than the one of the circuit description. Second, a functional language is very appropriate as a behavioral representation. By simple procedural abstraction, it extends all the way from the logic level to the register transfer level and even the system level. Other levels of description require only an introduction of more data types besides booleans and lists and vectors of booleans (e.g. integers). Semanticist's procedural behavior language is indeed close to the circuit description, but it is also close to higher levels of representation.

A further argument in favor of Semanticist in terms of practicality is that it can be used incrementally (in some sense). Cells that are below in the hierarchy can be designed and verified first, and the user can gradually move up in the hierarchy. However, if he changes a cell down in the hierarchy, he has to verify all cells above it again.

A last argument in favor of Semanticist's practicality is that it gives reasonable guidance in case of errors. Errors other than syntax error are perfectly located and explained by the system. In case of a syntax error, the user is less fortunate but he still gets enough information to locate the error himself. If the parser fails to reduce a circuit to the start module, it prints out all the top level modules of the parse tree that it ended up with. If the user can't locate the error with that information, he can parse lower subcircuits in the hierarchy, and see whether the top level modules there correspond with what he expects to get. Focusing on small enough circuit portions makes is possible to find out why the syntax error occurred.

6.2 Further work

In my view, the way to go from here is to interleave syntax and semantics in some way. When people process a sentence, they don't deal with syntax and semantics separately. In order to improve on the shortcomings of Semanticist that were mentioned in the previous section, we have to give up on the notion of having a pure

syntactic analysis first, and a semantic analysis next. We have to give up on a parser that only reduces modules to higher level modules. The parser needs to have the ability to look at the contents of modules that it already formed, and base its decisions on that contents. This would make it possible to cover feedback loops over DCN's. It would also make it easier to cover the NORA rules, thereby improving the efficiency of the system: the parser could look at a static gate and see whether it is an invertor or not. In order to handle bypassing in pass transistor logic, it would also be handy if the parser could make changes to the parse tree that it already formed.

A further topic to be dealt with is how useful the formalisms and mechanisms in Semanticist are for circuit synthesis. An interesting question in natural language processing is to which extent the machinery for understanding sentences is also used to produce ones. A similar question can be asked with respect to VLSI design. How far does syntax and semantics bring us in solving the synthesis problem? Maybe synthesis and verification are not that far apart.

Bibliography

[Agre 88] Philip E. Agre, *The Dynamic Structure of Everyday Life*, Ph.D. thesis, MIT, AI-TR 1085, 1988.

[Bamji 89] Cyrus Bamji, "GRASP: A Grammar-based Schematic Parser", to appear in *Proceedings of the 1989 Design Automation Conference*; a comprehensive presentation of GRASP can be found in Bamji's Ph.D. thesis, which is appearing soon (MIT).

[Barendregt 84] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, New York, 1984, Revised Edition.

[Barrow 84] H. Barrow, "Proving the Correctness of Digital Hardware Designs", *VLSI Design*, Vol. 5, No. 7, 1984.

[Bolsens 88] Ivo Bolsens, W. De Rammelaere, C. Van Overloop, L. Claesen, H. De Man. "A formal approach towards electrical verification of synchronous MOS circuits", *Proceedings of the 1988 ISCAS conference.*

[Boute 86] R.T.Boute, "Current Work on the Semantics of Digital Systems", in G. Milne, and P.A. Subrahmanyam (editors), *Formal Aspects of VLSI design*, North-Holland, 1986.

[BD 77] R. M. Burstall, J. Darlington, "A Transformation System for Developing Recursive Programs", J. ACM 24, 1, pp 44-67, January 1977.

[Bryant 81] Randal E. Bryant, *A Switch-level Simulation Model for Integrated Logic Circuits*, Ph.D. thesis, MIT, VLSI memo 81-50, 1981.

[Bryant 84] Randal E. Bryant, "A Switch-level Model and Simulator for the MOS Digital Systems", *IEEE Trans. Comput.*, vol. C-33, pp 160-177, Feb 1984.

[Bryant 85] Randal E. Bryant, "Symbolic Verification of MOS circuits", *1985 Chapel Hill Conference on Very Large Scale Integration*, pp 419-438, 1985.

[Bryant 87] Randal E. Bryant, "Boolean Analysis of MOS circuits", *IEEE Transactions of Computer-Aided Design*, vol. CAD-6, July 1987.

[CGM 87] A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification using Higher-Order Logic", in D. Borrione (editor), *From HDL descriptions to guaranteed correct circuit designs*, North-Holland, 1987.

[Dijkstra 76] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N. J., 1976.

[Gifford 87] David Gifford *et al.*, *FX-87 Reference Manual*, MIT/LCS/TR-407, 1987.

[Goncalves 83] Nelson Goncalves, Hugo De Man, "NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures", *IEEE Journal of Solid-State Circuits*, Vol. SC-18, No 3, pp 261-266, June 1983.

[Gries 81] David Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

[HD 86] F. K. Hanna, and N. Daeche, "Specification and Verification using Higher-Order Logic: A Case Study", in G. Milne, and P.A. Subrahmanyam (editors), *Formal Aspects of VLSI design*, North-Holland, 1986.

[Hoare 69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, 12(10):576-583, October

1969.

[Hunt 87] Warren A. Hunt, "The Mechanical Verification of a Microprocessor design", in D. Borrione (editor), *From HDL descriptions to guaranteed correct circuit designs*, North-Holland, 1987, pp 89-129.

[HW 73] C. A. R. Hoare, N. Wirth, *An Axiomatic Definition of the Programming Language Pascal*, Acta Informatica, 2(4):335-355, 1973.

[Johnson 84] S. Johnson, *Synthesis of Digital Designs from Recursion Equations*, MIT press, 1984.

[JBG 85] J. Joice, J. Birtwistle, M. Gordon, *Proving a Computer Correct in Higher Order Logic*, Technical Report, University of Calgary, 1985.

[LB 85] Hector J. Levesque, Ronald J. Brachman, "A Fundamental Tradeoff in Knowledge Representation and Reasoning", in Ronald J. Brachman, Hector J. Levesque (editors), *Readings in Knowledge Representation*, Morgan Kaufmann, 1985, pp. 41-70.

[Landin 64] Peter J. Landin, "The Mechanical Evaluation of Expressions", *Computer Journal*, 6:308-320, 1964.

[Minsky 81] Marvin Minsky, "A Framework for Representing Knowledge", in J. Haugeland (editor), *Mind Design*, pp 95-128, The MIT Press, Cambridge, Massachusetts, 1981.

[Minsky 85] Marvin Minsky, *The Society of Mind*, Simon and Schuster, New York, 1985.

[MW 81] Zohar Manna, Richard Waldinger, "A Deductive Approach to Program Synthesis", in Bonnie Lynn Webber, Nils J. Nilsson (editors), *Readings in Artificial Intelligence*, Morgan Kaufmann, Los Altos, California, 1981, pp. 141-172.

[Paillet 87] J.-L. Paillet, "A Functional Model for Descriptions and Specifications of Digital Devices", in D. Borrione (editor), *From*

HDL descriptions to guaranteed correct circuit designs, North-Holland, 1987, pp 21-42.

[RCA* 86] Jonathan Rees, William Clinger, H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, M. Wand, *Revised*^{3} *Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21(12), December 1986.

[Schmidt 86] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon Inc., Boston, 1986.

[Spickelmier 88] Rick L. Spickelmier, A. Richard Newton, "Critic: A Knowledge-Based Program for Critiquing Circuit Design", *Proceedings of the 1988 ICCD Conference*, pp. 324-327.

[Stoy 77] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, Mass., 1977.

[VAVO 87] Filip Van Aelten, Cris Van Overloop, *Formele theorie over correct gedrag van CMOS VLSI circuits*, Ingenieursthesis, K.U.Leuven, 1987.

[Weise 86] Daniel W. Weise, *Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits*, Ph.D. Thesis, MIT, AI-TR 978, 1986; a summary of this appeared in *Proceedings of the 1987 Design Automation Conference*, pp 265-270.

[Weste 85] Neil Weste, Kamran Eshraghian. *Principles of CMOS VLSI design*, Addison Wesley, 1985.