THE APPLICATION OF KNOWLEDGE-BASED SYSTEMS TO DESIGN VERIFICATION

by

Ricky Lee Spickelmier

Memorandum No. UCB/ERL M89/126

29 November 1989

THE APPLICATION OF KNOWLEDGE-BASED SYSTEMS TO DESIGN VERIFICATION

by

Ricky Lee Spickelmier

Memorandum No. UCB/ERL M89/126

29 November 1989

TITLE PAGE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

THE APPLICATION OF KNOWLEDGE-BASED SYSTEMS TO DESIGN VERIFICATION

by

Ricky Lee Spickelmier

Memorandum No. UCB/ERL M89/126

29 November 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Application of Knowledge-Based Systems to Design Verification

Ricky Lee Spickelmier

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Science

Abstract

As a circuit design proceeds, it is continually checked by the designer for errors, by visual inspection, by simulation, or by using other verification tools (such as electrical rule checkers). Before most designs can proceed to fabrication, they must go through a formal design review where other designers analyze the design and look for errors. Errors found during the verification process range from easily recognizable problems, such as a power supply short, to a CMOS static gate having N and P cores that are not logical duals, to hard-to-recognize problems, such as those dealing with charge sharing and race-conditions.

A circuit critic is a tool that finds errors in a circuit design and may recommend corrections. There are three main uses of a circuit critic: finding errors not easily found by other verification tools (*e.g.* timing, charge sharing), finding errors for novices, and checking design style compliance. There are many errors that novice designers can make as they do circuit designs. Novices learn the circuit configurations that lead to these errors by experience. A circuit critic can have the patterns that describe these errors in it and the novice can use the critic to check the circuit. Many design styles have a set of rules, that if followed will produce a "correct" design. A circuit critic can check for compliance with these rules.

The work presented in this dissertation focuses on exploring the ideas of technology independence of critics, tight integration with a CAD system, and the representation of knowledge for the critic. To explore these ideas a test-bed, called **Critic**, was developed. **Critic** reads a description of the technology and design style to be used in the check and is tightly integrated into the Berkeley Design Environment, both in terms of the data input to the system and the control of **Critic**. An example of the use of **Critic** and the results of using **Critic** are presented.

[illegible]

Professor A. Richard Newton
Thesis Committee Chairman

Contents

Acknowledgements

First, I would like to thank my research advisor, Professor Richard Newton, for his direction, support, and employment throughout my long stay at Berkeley.

I would like to thank the past and present members of the Berkeley CAD group for making my long stay at Berkeley a pleasant experience, in particular, my friends: Wendell Baker, Giles Billingsley, Jeff Burns, Professor Ron Gyurcsik (and Peggy), David Harrison, Mark Hofmann, Tom Laidig, Brian Lee, Grace Mah, Karti Mayaram, Peter Moore, Tom Quarles, Fabio Romeo, Richard Rudell, Professor Res Saleh, and Ellen Sentovich.

I would like to thank my friends at Tektronix for putting up with me over many summers, especially Randy Young, Ian Getreu, and John Crawford, who continued to let me work at Tektronix for eight summers (probably four summers longer than they should have). My employment at Tektronix led directly to my desire to attend Berkeley.

I had many interesting discussions with circuit designers from industry. Of those, I would especially like to thank Karlheinz Horniger from Seimens and Pat Bosshart from Texas Instruments. I also had many helpful discussions with Professor Hugo DeMan of the Catholic University at Leuven.

Margo Seltzer helped me finish my thesis by bugging my about my progress, I am indebted to her. Cindy Mason has been a good friend over the entire Ph.D process, I can't thank her enough for her encouragement. I wish her the best of luck in finishing her thesis.

I would like to thank my friends in the Experimental Computing Facility at Berkeley for providing me with a place to relax and for bugging me to finish during the last few months of thesis writing. I would especially like to thank Mark Hastings, Anne Hendry, Phil Lapsley, Jonathan Lee, and Kurt Pires. It would have been difficult to finish without their help.

The following people provided much needed editorial comments on drafts of my thesis: Paul Cohen, Professor Ron Gyurcsik, David Harrison, Theologos Kelessoglou, Professor Don Pederson, Professor Peter Pirolli, and Pete Simanyi.

The financial and equipment support provided by the Semiconductor Research Corporation, the Digital Equipment Corporation, Tektronix, Texas Instruments, and the XEROX Corporation (University Grant Program) is greatfully acknowledged.

Finally, I would like to thank myself for finishing. It took far longer than it should of, but now it's done!

Chapter 1

Introduction

As a circuit design proceeds, it is continually checked by the designer for errors either by visual inspection, by simulation, or by using other verification tools. Errors found during the verification process range from easily recognizable problems, such as a power supply short or a CMOS static gate having N-type and P-type cores that are not logical duals, to hard-to-recognize problems such as those dealing with charge sharing and race conditions. As circuits get larger and more complex, more errors and more types of errors are found. The size and complexity of current circuit designs take them far beyond the realm of manual verification. To simplify the problems inherent in large and complex designs, tools have been created to automate the verification process. Another way to deal with the complexity is to introduce *design styles*. Design styles consist a set of rules for circuit construction that if followed will eliminate certain classes of errors[24, 40]. Simple design styles can be composed of a small set of rules, such as limiting the number of series connected MOSFETs in a CMOS static-gate. Other more complex design styles exist. One such design style is used for NORA logic[24]. NORA is a design style for dynamic CMOS with complementary clocks of arbitrary skew. There are nine rules that must be followed when designing NORA circuits. By following these rules, no race conditions (because of clock skew) can exist in the circuit. Thus by following a set of rules, many problems associated with dynamic design can be eliminated. However, the rules further complicate the design process, and tools have been developed to check for compliance with the design style rules.

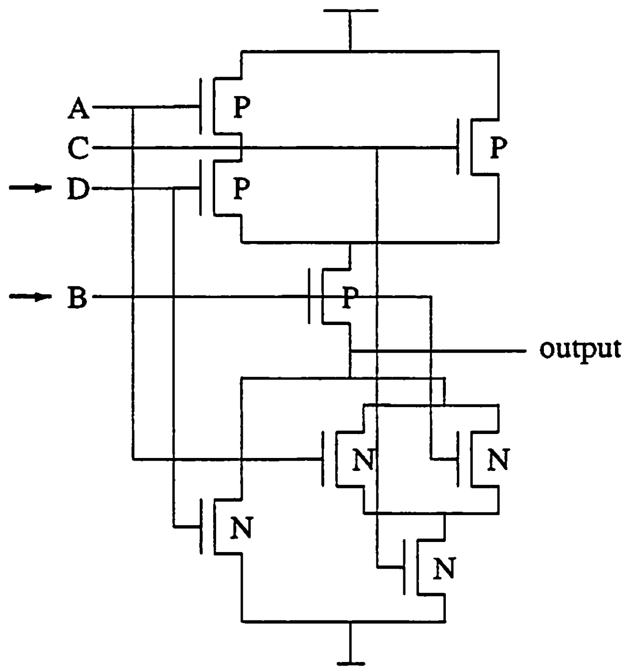
Circuit Critics

Many types of CAD tools can show errors, but not look for errors. For example, simulation results can show that a circuit is functioning incorrectly, but the designer then must choose what test vectors to give to the circuit and what nodes to plot to track down the error. A circuit critic looks for errors or "bad design style" in circuit designs[47, 48, 15, 14, 3, 68, 35] and may, in fact, suggest an approach to correction of the errors. There is a large amount of knowledge about how circuits should be properly designed and what might cause problems. Circuit critics are tools that use this knowledge to critique a circuit design. There are three main uses of a circuit critic: finding errors not easily found by other verification tools, finding errors commonly overlooked by novices, and checking design style compliance.

Many errors are easily handled by circuit critics but can be difficult to find using other verification techniques. For example, if the inputs to the P-type and N-type cores of a CMOS static-gate are connected incorrectly, the supplies may be shorted or the output node of the gate may float (see Figure 1.1). Unless the designer happened to test the circuit with the proper set of test vectors such that the inputs to the gate (possibly many levels into the circuit) caused the error, the error would not be found. As another example, if the internal capacitance of a dynamic gate is too large compared to the load capacitance, the load capacitance may not be charged to the full value and thus may not turn on the load devices (see Figure 1.2). While this error can be found using simulation, it is rare that all outputs of dynamic-gates would be plotted in a simulation run, and thus the error would have to be tracked down based on some error detected on the nodes plotted.

Novices learn how to design circuits by experience. Along the way they create designs with errors that an experienced designer would not make. The description of these errors are be given to a circuit critic, and a novice uses the critic to not only find errors, but to help in the learning process.

Another use of circuit critics is in enforcing design style compliance. Many design groups have a set of rules for design that if followed will eliminate certain classes of errors. A circuit critic can use the rules to verify that a circuit complies with the design style[15, 14].



Inputs	Output
0000	1
0001	float
0010	1
0011	0
0100	float
0101	0
0110	0
0111	0
1000	1
1001	short
1010	0
1011	0
1100	float
1101	0
1110	0
1111	0

Figure 1.1: Incorrectly Connected Gate

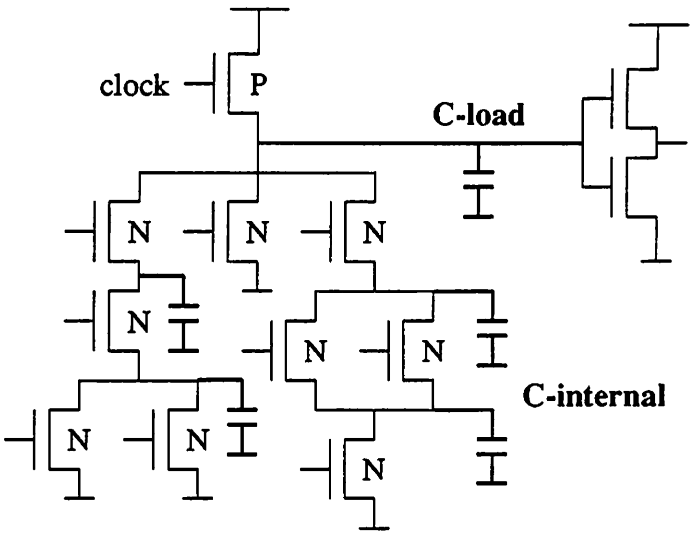


Figure 1.2: Possible Charge Sharing Problem

Scope

The work in this dissertation explores data representations for circuit critiquing programs that are technology and design style independent. and the tight integration of these programs into CAD systems. The work in representation covers methods for representing the knowledge used by circuit critiquers to find errors in a circuit and methods for representing the circuits. Representation of knowledge explores how process, design style, and technology specific information can be represented along with how to represent the rules used for finding errors in the circuit. Both external representations (what the user sees and creates) and internal representations (what the algorithms of the critiquer work on) are presented. The use of existing forms of representation, in particular, text-based formats and circuit design databases, along with generalizing descriptions from examples is explored. The purpose of technology and design style independence is to reduce the work needed to check a circuit designed in a new technology and design style. The work in integration partially overlaps the area of representation since the facilities provided by the CAD system affect how to represent data. The integration work covers the use of the facilities provided by a CAD system to control the execution of the critic and how the results of the critic execution can be presented.

Outline of the Dissertation

The remainder of this dissertation is composed of eight chapters and one appendix. In Chapter 2 an overview of circuit critiquing and background on previous work is given. The representation of primitive circuit elements in explored in Chapter 3. How collections of primitive circuit elements are represented and searched for in a circuit design is presented in Chapter 4. The representation of error checks and how errors are located in a circuit design is explored in Chapter 5. **Critic**, a test bed for the ideas presented in the previous chapters, is described in Chapter 6. An example of the use of **Critic** and results for the use of **Critic**, using a CMOS technology description, in a VLSI design course are are given in Chapter 7. Conclusions and directions for future research are presented in Chapter 8. The appendix contains an example of automatically generated ruleset documentation.

Chapter 2

Overview and Previous Work

In this chapter an overview of circuit critiquing and a summary of previous work in the area of circuit critics is presented. The chapter is divided into three sections: an overview of critiquing, a description of the basic flow of critics, and a description of the previous work in circuit critics.

Overview

Verification and analysis tools have been used to automate the task of searching for and finding many types of errors. However, many of these tools tend to miss the class of errors that can be attributed to 'experience'. The errors in this class are encountered as designers learn their craft. The designers remember the reasons why these errors occurred and make sure they do not introduce them again (either by never introducing them or by looking at their design at various stages to see if they have introduced these errors). These errors are generally referred to as *rules of thumb*. While most of these errors have good theoretical foundation for why they occur, they are usually thought of as something not to do, without much thought as too why they are bad. As examples of errors that fit this classification: don't put more than N mosfets in series in a CMOS static gate, don't put more than M pass gates in series without intervening level-restoring logic, and do not use this two-phase gate if the clocks overlap by more than X nanoseconds. The problem with these types of errors is that they are rarely written down and thus new designers must rediscover them, either by peer review or by discovering the error and the cause in their design (by the use of verification and analysis tools).

One method of reducing the complexity of design to manageable levels is to develop *design styles*[1]. Design styles are a set of rules for circuit construction that if followed will eliminate certain classes of errors, such as DOMINO[40] and NORA[24] for dynamic CMOS logic. Design styles can be composed of a small set of simple rules, such as limiting the number of series connected MOSFETs in the N-type core of a CMOS static-gate. Most companies have design styles that place restrictions on designs, *e.g.* only a finite number of transistor sizes allowed, certain circuit configurations not allowed (various XOR's and LATCHes fall into this category). Other more complex design styles exist.

One such design style is used in NORA logic [24]. NORA is a design style for dynamic CMOS with complementary clocks of arbitrary skew. There are nine rules that must be followed when designing NORA circuits. By following these rules, no race conditions because of clock skew can exist in the circuit. The following are two example rules:

- If the number of static inversions between two dynamic blocks is even, the two dynamic blocks must be complementary. This implies that alternating N and P blocks must be used for direct connections between dynamic blocks.
- If the number of static inversions between two dynamic blocks is odd, the two dynamic blocks must be of the same type.

Other design styles insure that a circuit can be tested[17, 41]. Level-sensitive design[17] is a such a design style. Circuit critics can verify compliance with a particular design style. This type of analysis is called *verification*.

Basic Flow

The flow of a circuit critic can be broken into a several phases, some of which are optional. These phases are:

1. Loading of the description of the technology and design style.

[1]Design styles are also known as design methodologies or structured design.

2. Loading of a description of the circuit to be checked.
3. Extracting the structure.
4. Locating the errors.
5. Presentation of the results.

Loading the Description of the Technology and Design Style

In this phase of the flow, a description of the technology and the design style is loaded into the program, either explicitly or implicitly. An implicit loading means that the knowledge is an integral part of the program. The description includes information about the primitives of the technology, the structures can that can be composed out of the primitives, and the possible errors that might occur when designing using the design style. In addition, to simplify the structure and error descriptions, common constants and subexpressions are separated out. These constants and expressions are usually directly related to the process and design style being checked, such as the threshold voltage of MOSFETs, the maximum number of series connected MOSFETs that is allowed for the design style, or the maximum allowed clock skew.

While this information is needed for a circuit critic, some circuit critics are have this information hardwired for a particular technology and/or design style and thus skip this phase. In these critics the information is implicit in the program. When the description is implicit, using the critic to check a new technology or design style could require rewriting a large portion of the program. If the description is explicitly represented, the amount of work necessary to check a new technology or design style depends on the degree of separation between the program and the description.

Loading the Circuit

Loading the circuit entails reading in the description of the circuit from a text description or a data base, along with processing any extra annotation associated with the circuit. For most critics, the circuit description is represented as a netlist of primitives, where a primitive can range from a single MOSFET to an entire logic block. Along with each primitive are attributes, such as the width and length of a MOSFET, that modify the default attributes of the primitive. Depending on the complexity of the design system used with the critic, the netlist can be in some standard textual format, such as SIM, SPICE[60], and EDIF[2] netlist formats, or can be retrieved from a common design database (*e.g.* OCT[28]).

Many design databases and some textual formats (such as EDIF) allow for the addition of arbitrary annotation. Designers of critics can develop *policies* for storing additional information in the database. These policies describe how the additional information should be stored and interpreted. This additional information can include what regions of the circuit to check, what errors to check or not check on a per element or region basis, and other forms of control. Besides directing the control of the critic, annotation can be used to give the error checks extra information. For example, one critic, QCRITIC[3], uses the results of a SPICE simulation to help in finding errors.

Extracting the Structure

The complexity of large circuit designs makes it difficult, if not impossible, to design at the level of the primitive components. Instead primitive components are grouped together into higher level components and these higher level components are grouped into higher level components, and so on. This is the basis of hierarchical design. Structure extraction is either the process of taking a collection of interconnected primitives and finding the higher level structures that were originally used in the design, or the process of finding new structures not intended or expected (such as unexpected storage elements). By extracting the structures in a design a designer can easily see if the circuit has been implemented correctly. Structure extraction is also known as *gate recognition* and *decompilation*[25, 66, 49]. Another form of structure extraction is that of extracting the symbolic representation from the layout[46].

A major use of structure extraction is to simplify the process of finding errors or verifying the correctness of a design. Most descriptions of errors in a design are not represented in terms of the low level primitives, but in terms of the structures that are composed of the primitives. Rather than have the error check specify how these structures

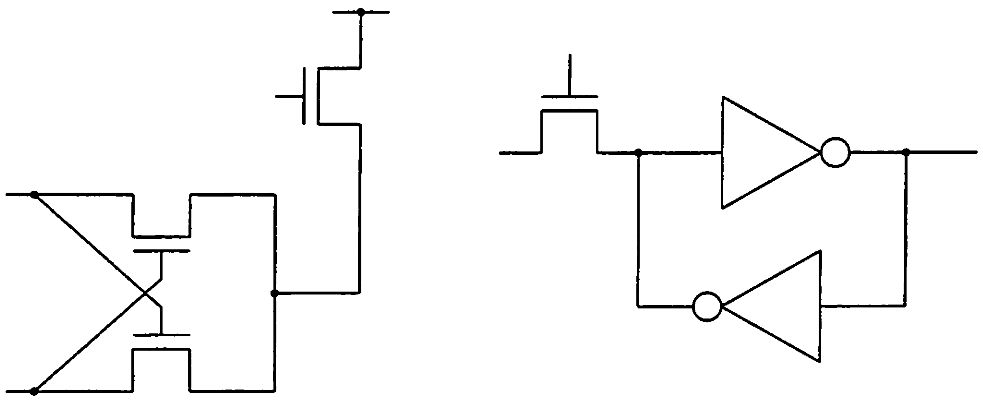


Figure 2.1: XOR and LATCH

are composed (and be responsible for finding them), a structure extraction can be performed before the error checks (or sometimes during the check), allowing the error checks to be specified more simply in terms of the structures.

Finding Errors

The primary purpose of circuit critics is to find errors (or verify that there are no errors) in the circuit design. There are two types of errors that can be flagged: the presence of something that should not exist or a set of attributes that violate one or more constraints. An example of the former is an illegal XOR configuration and an example of the latter is a LATCH that can not change state because of a weak load gate (see Figure 2.1).

Depending on what is allowed in error check specifications, pattern matching could be required. In RUBICC[47], the same rule language used for finding structures is used in the error check and thus the same pattern matching operations can be used (in practice the amount of actual pattern matching was limited since the structure finding phase found most of the structures of interest). Another, QCRITIC[3], has no structure finding phase so errors contain the patterns needed to find the elements needed for the error check.

Presenting the Results

After the structures and errors have been found in the circuit description they are presented to the designer for action. The information can be presented textually or graphically. If the netlist is in the form of a textual description, the output is usually a textual description of the elements in error. If the data is stored in a common design database and there is an editor for that database, then the information can be graphically displayed (where appropriate). This can be further improved if the critic can run from the graphics editor and as errors are found the user can make modifications to the circuit to fix the error. If the critic is integrated with a common database and graphics editor, a whole range of possibilities arise for the presentation of results. The editor can be used to step through the structures and errors found in the design. The editor can highlight the objects in the structure or in error and zoom to the region. If the editor supports dialog facilities, dialog boxes of textual information can be supplied to describe the structure or error (detailed information about why the object is in error).

Previous Work

In the past few years many programs for circuit critiquing have been developed [23, 14, 47, 3, 68, 10]. Most of these have been developed to solve a particular design problem facing the organization which developed the critic. One critic[14] was designed to check compliance with a complex design style. Another was designed to check for common errors made by novices[47].

This previous work can be broken into two distinct classes: work in which developing a critic was the major focus, and work in which a critic (or multiple critics) is a part of a larger task.

RUBICC - Rule Based IC Critiquer

RUBICC[47], developed by Lob and Newton at the University of California at Berkeley, checks two-phase dynamic NMOS circuits. RUBICC is written in HPRL[43, 42, 12], which is an extension to LISP that supports frame-based knowledge and both forward and backward chaining rule systems. Although RUBICC does not have an explicit separation between the knowledge about the technology and design style being checked and the program, the technology and design style constants are broken out. RUBICC has three distinct phases: primitive classification, structure finding, and error checking.

In the primitive classification phase, the MOSFET primitives are classified into the actual function they perform, such as pass gate, driver, and load. The classification is determined by how the primitive is connected to other devices, supplies, and clocks. The major reason for classifying primitives is to simplify the structure finders and error checks. This is analogous to the reason for having structure finding to simplify the error checks.

In the structure finding phase, complex structures are built out of the primitives and other structures in a bottom up fashion. RUBICC does not try to determine an ordering for the structure finding rules, and instead uses the order in which the rules were entered into the system.

In the error finding phase, the same pattern matching that was performed in the structure finding phase is used to find the errors. Again, the error checks are ordered, because of interaction among the rules. The existence of one error may negate the need to check for certain other types of errors. For example, the fact that the widths and lengths of a MOSFET are wildly out of range will negate most other error checks that use the MOSFET widths and lengths in their calculations.

Figure 2.2 shows a RUBICC rule for checking an error associated with a dynamically clocked gate. Each rule is composed of a single form that contains the name of the rule, the type of the rule (forwards or backwards chaining), the types of database elements to be used in the pattern matching portion of the rule, the premise (tests to be performed and patterns to match), and the conclusion (actions to be performed if the premise is true). The **type** form lists the types of objects to be used in the rule and the variables to be assigned to them. In the example rule, two objects, each of a different type (precharger and driver), are being used, assigned to the variables **pre** and **dr** respectively. The premise form describes a set of tests, each of which must evaluate to true. In the example rule there are two forms, one that does pattern matching and another that checks constraints. The pattern matching section can either match an item directly in the database or force a backward chaining rule to fire. The constraints section uses the variables matched in the pattern matching phase. In the example rule the conclusion section evaluates a form that creates an error frame in the database.

Note that the example rule required detailed knowledge of the syntax of the un-

```
(rule dynamic-clocking-rule-1 backward-chain-rule
    (type (precharger ?pre) (driver ?dr))
    (premise (test (and (?pre s-node ?sn)
                        (?pre pre-phase ?pp)
                        (?dr d-node ?sn)
                        (?dr s-node 0)
                        (?dr clk-input ?clkin))
                   (or (equal ?pp ?clkin)
                       (equal ?clkin '*high))))
    (conclusion (funny-node clocking-flag
                            ^(make-node-name ?sn))))
```

Figure 2.2: Example RUBICC Rule

derlying rule system. However since many of the rules have the same basic form, previous rules can be used as templates for new rules.

DIALOG

DIALOG[15, 14], developed by DeMan, *et. al.* at the Catholic University at Leuven, was originally designed to check for design style violations in CMOS NORA[24, 33] circuits. DIALOG uses a special purpose language for describing the errors (LEXTOC). This language can be interpreted in LISP or translated into PASCAL and compiled. LEXTOC is used for both describing the structures to be found during the run and the errors to be checked. The language provides basic data types and constructs for describing structures and errors; where the language does not provide the necessary expressiveness, the designer can escape to LISP. The circuits are described using an *Invariant Network Representation (INR)* and LEXTOC allows the rules to transform the circuit from one form of the INR to another; which another name for structure extraction. Figure 2.3 shows an example of a DIALOG rule for finding a structure (called a *composition* rule) and Figure 2.4 shows an example of a DIALOG error check rule. In [14] the comment "Notice the simplicity of expressing knowledge in this system" is made. While LEXTOC does make it possible to describe a wide range of structures and error checks, it requires the person who creates the ruleset to learn a new language, one different from what a circuit designer is used to.

```
(RULE c2mos-definition
    (IF
        ((PROPERTY fibar-cloctor (clocktransistor ct1))
         (PROPERTY ptype ct1)
         (RELATION inout n1 ct1)
         (PROPERTY fi-clocktor (clocktransistor ct2))
         (RELATION inout n1 ct2)
         (PROPERTY ntype ct2)
         (RELATION input n2 ct1)
         (RELATION input n3 ct2)
         (RELATION inout n4 ct1)
         (RELATION inout n5 ct2)
        )
     THEN
        ((IF (COMPLEMENT (LOGFUN (gnd n5) ((ntype e)))
                         (LOGFUN (vdd n4) ((ptype e))))
             (CLUSTER el1 (n4 vdd) ((ptype e)))
             (CLUSTER el2 (n5 gnd) ((ntype e))))
          THEN
             ((MERGE varc2mos el1 el2 ct1 ct2)
              (ASSIGN fi-c2mos varc2mos)
              (ASSOCIATE output varc2mos n1)
              (CREATE (c2mos varc2mos))))))
```

Figure 2.3: Example DIALOG Structure Finding Rule

```
(RULE input-delay-racefree-1
    (IF
        ((PROPERTY fi-c2mos e1)
         (RELATION output n1 e1)
        (PATH el1 (n1 ((or fi-nblock fi-block) e)) (static e))
        )
     THEN
        ((ASSIGN-ERROR rule-1-violation e1)
         (PRINT " fi-c2mos = " (INSTANCE e1))
         (PRINT " static cmos path = " (INSTANCE el1)))))
```

Figure 2.4: Example DIALOG Error Check Rule

```
If the goal is "finderror" and
   the process is "Super-High Three" and
   there is a PNP substrate transistor and
   the voltage at the emitter is greater than or equal to
     the voltage at the collector + 25
Then record an error
```

Figure 2.5: English Version of QCRITIC Rule

DIALOG was also extended to cover the RUBICC rules described in [47].

QCRITIC - Quick CRITIC

QCRITIC[3], developed by Bergquist and Sparkes at Tektronix, is used for checking bipolar analog circuits. QCRITIC is written in OPS83[21], an AI language that allows simple frame-based knowledge and (the description of) forward chaining rules. QCRITIC loads a set of OPS83 rules that describe the errors to check, but does not do any structure extraction. The error checks work directly at the level of primitives in the circuit (transistors, resistors, capacitors), using pattern matching to find needed collections of elements. QCRITIC uses extra circuit annotation in the form of simulation information from the SPICE[60] circuit simulator. For example, it may use the maximum voltages between nodes to determine if the breakdown voltages for devices have been exceeded (or devices have entered saturation) or use the maximum current into a terminal to see if the current is out of range.

Figure 2.5 shows a 'human-readable' example QCRITIC rule. In this rule a check is made to see if the voltage across the collector and emitter terminals is greater than the breakdown voltage. The check is composed of test and action sections. In the test section, four clauses are used. The first verifies that the current phase of the critic run is the 'finderror' (or error checking) phase. The second makes sure the process of the circuit being checked is 'Super-High Three' (the name of a bipolar process used at Tektronix). The third clause is used to find all PNP substrate transistors. The final clause checks the voltage difference between the collector and emitter of the transistor.

It is important to note (and will be discussed in more detail in the following chapters) that this check contains not only the error check to be performed, but also contextual and rule system semantic information.

Design Advisor

DESIGN ADVISOR[68] is a critic for checking logic gate and latch level designs for NCR's design system. DESIGN ADVISOR employs artificial intelligence techniques to the critiquing problem and uses the Proteus system[59] for representing knowledge and finding errors in a design. DESIGN ADVISOR uses *truth maintenance*[16] techniques to allow the system to support multiple possibilities for why an error might occur. The use of truth maintenance also permitted the retraction of faulty assumptions made during the check.

DESIGN ADVISOR uses the same input representations as other tools in the NCR tool suite and has a graphical interface to display the errors found in the circuit. The program has rules for checking testability, manufacturability, performance, and overall design quality for the NCR gate array design system.

Figure 2.6 shows an example DESIGN ADVISOR rule. This rule checks for phase skew caused by differing numbers of loads on the inverting and non-inverting outputs of a two-phase clock driver. The connection form finds a component of type **'pcl2'** (two-phase clock driver) and binds the name of the driver to **'?pcl2-name'**, the input to **'?pcl2-in'**, and the two outputs to **'?pcl2-ph1'** and **'?pcl2-ph2'**. The fanout-cnt forms return the number of connections to the clocks bindings to **'?ph1-cnt'** and **'?ph2-cnt'**. The **unless** form returns true if the counts are not equal; if the counts are equal, the action is evaluated. The action triggers the storage/presentation of the error.

ESTA

ESTA[10], developed by P. Camurati, *et. al.* at the Politecnico di Torino, is a program for verifying that circuits have been designed following the rules for *Design For Testability*[17, 1]. Designing a circuit for testability means adding extra circuitry (usually latches), and making connections in order to improve the ability of the circuit to be tested. ESTA is written in PROLOG[11]. As an example of testability rules:

```
((connection ?pcl2-name:pcl12 ?pcl2-in ?pcl2-ph1 ?pcl2-ph2)
 (fanout-cnt ?pcl2-ph1 ?ph1-cnt)
 (fanout-cnt ?pcl2-ph2 ?ph2-cnt)
 (unless (= ?ph1-cnt ?ph2-cnt))
 -->
 (phase-skew ?pcl2-name))
```

Figure 2.6: Example DESIGN ADVISOR Rule

A latch X may gate a clock Ci to produce a gated clock Cy which drives another latch Y if, and only if, clocks Cy and Ci are not functionally dependent.

This rule may be violated if and only if:

There are two SRL's and a combinations network connected in such a way that the input clock of the second latch is fed by the output of the combinational network one of whose inputs is the output of the first latch.

This rule is coded in PROLOG for ESTA as follows:

```
lssd22(Dx, Ox) :- latch(Dx, Cx, Ox),
                  network4(Ci, Ox, Cy),
                  latch(_, Cy, _),
                  clock(Dx, Cx, Ci).
network4(C, Co, Cy)
                :- net(T, C, _, Cy),
                  input(Ox, T).
clock(Dx, Cx, []).
clock(Dx, Cx, [H|T])
                :- case(Dx, Cx, H),
                  clock(Dx, Cx, T).
```

CRITTER

Critter[35], developed by Kelly and Steinberg at Rutgers University, is a program for comparing the specification of a design with its behavior. It can symbolically determine the outputs based on the input specification and can take an output specification and determine the inputs required. When give both the inputs and the outputs **Critter** can derive the outputs from the inputs and compare them with the expected outputs. **Critter** works on netlists of instances of *modules*. Each module is described by two parts, a set of operating conditions and a set of input/output mappings. The operating conditions are a set of constraints on the inputs that must be satisfied before the module will work. Example constraints might be the set up time for a latch or the skew of two signals. The input/output mappings describe the effect of the inputs on the outputs.

Palladio

Palladio[8, 9], developed by Brown, *et. al.* at XEROX PARC, is a knowledge-based VLSI design assistant. Palladio directs a designer by helping incrementally refine a design from a high level description. Each step in the refinement is directed by a design assistant that 'knows' how to correctly go from one level of abstraction to another. As the refinement continues, the system follows rules that force each step to produce a legal (correct) design. As examples, as high level blocks are chosen the system makes sure that are no clocking problems or deadlock conditions, as specific devices are chosen the system makes sure there are no charge sharing problems or illegal logic levels, and as the final geometry is generated the system verifies that everything is design rule correct. Palladio can also make sure that components are correctly connected together at the same level of abstraction. For example, here are two of the rules for composing elements at the level of clocked switches and gates (known as the CSG level in Palladio):

> A control input of a steering switch or a steering net can be connected only to an output of a restoring logic gate.
>
> An output of a clocking switch can be connected only to an input of a restoring logic gate.

Palladio is implemented in LOOPS, an exploratory programming language embedded in Interlisp[71] that supports procedural programming, access-based programming, rule-base programming, and object-oriented programming[5, 70].

SCHEMA

SCHEMA[74, 73], developed by Zippel at MIT, is an integrated system for designing circuits. It is a synthesis system with critiquing and analysis for feedback. The basic concepts of SCHEMA are: the internal semantics of the system should match the semantics of the circuit designer (allowing it to better explain what was going on), components should be specified in terms of electrical parameters rather than device sizes, and a single design repository and description language.

As part of the process, SCHEMA, had an analysis phase the would verify the correctness of the design, *i.e.* did the design meet the specified constraints, where there design errors. The critiquing phase would check for errors such as:

"Does this inverter have a trip point of 2.3 volts?"

"Does the bootstrapped node boot?"

Current Work

Current work in the area of checking circuits is heading towards formal verification. DIALOG[7] is moving from a rule-based approach to one based on a formal specification of correct digital circuits. PRIAM[50] verifies that a circuit functions according to the specifications, and tries to modify the circuit such that it meets specifications. Work is continuing in the area of extracting structure from netlists[72, 32].

Problems

There are some short comings that exist in the work on many of the previous and current circuit critics. In particular, many systems are developed for a particular technology and/or design style, the description of the circuit, primitives, structures, and errors is based on the syntax of the implementation, and can contain implementation-specific information, and the critics are not integrated into larger design systems.

Many critics are hardwired with specific knowledge of the primitives, structures, and types of errors for a particular technology or design style, for example, RUBICC can only check two-phase dynamic digital CMOS circuits, DIALOG was developed MOS circuits, and **Critter** only works for digital circuits. This makes it difficult for the system to be modified to check different technologies and design styles[2]. Critics should be independent of the technology and design style. This allows the system to move to new technologies and design styles with minimum effort; a new set of descriptions of the primitives, structures, and errors of the technology and design style, but no changes to the system itself.

Many critics require that the circuits and knowledge about the technology and design style be described using a language that is based on the underlying implementation. Some critics also require the use of implementation-specific functions in the description of the structures and errors. This can be seen in the example structure error check descriptions given in the previous section.

Many systems are stand-alone, they read in the circuit description, find the structures and errors, and then output their results. While they may be part of a larger set of tools for design, they are not integrated into a design system (DESIGN ADVISOR being an exception). This makes it difficult to provide such useful features as storing the results with the circuit, and displaying the structures and errors in the context of the circuit. Integration into a design system simplifies the use of the system by making the input, output, and control look the same as other parts of the system. Depending on the flexibility of the design system, the technology and design style knowledge, along with the circuit, can be described in the database of the design system, the structures and errors can be back-annotated to the database, and the display facilities used for displaying circuits can be used to display the structures and errors found during the run.

[2]Changing to slightly different design styles within a particular technology has been accomplished[7].

Chapter 3

Primitives

3.1 Overview

To critique a circuit, the human or computer program performing the analysis must understand the primitive elements that have been used to represent the circuit. In MOS designs, these are usually MOSFETs of various types and sizes, and capacitors. In higher-level designs, these primitives can be logic gates, registers, or ALUs. To understand the primitives, the descriptions of what they look like, what attributes they might have, and how they might be used in the representation of higher-level structures must be made available. In this chapter, how primitives can be represented in a critic tool (both in terms of their definition and in terms of their representation in the circuit) and how to handle the attributes associated with primitives is presented.

3.2 Representation

Before a circuit critic can process a circuit containing primitives and compositions of primitives in higher-level structures, primitives must be defined. This can either be done by hardwiring the definition into the critic, as has been the case for critics for a specific design style and technology, or the primitives can be described in such a way that new and different types of primitives can ber defined and added easily. A further refinement of the latter would have the primitive definition in a form that allows easy entry, browsing and editing of these definitions as well.

Before exploring the forms of representations for primitives, what actually needs to be represented must first be determined. This information can then be used to guide the implementation choice.

Attributes

Structure finders and error checks use attribute information associated with the primitives for finding structures and errors. The attribute information can be fixed, as in the type of an item, or it can be calculated, as in the W/L of a MOSFET, the beta-ratio of a MOSFET static-gate, or the sink current for a pin. The representation must allow the specification of fixed attributes along with the specification of calculated ones. Since there are calculated attributes, the representation must also specify how to interpret the instructions for calculating the attribute.

Terminals

Information about the terminals of the primitive must be included in the representation. The information that needs to be specified is the names of the terminals, the types of the terminals (input, output, bidirectional, *etc.*), and the permutability of the terminals.

Retrieval from a Database

In order to process a circuit description, the description must be interpreted. Simple text-based descriptions are read from a file and parsed. Integrated CAD systems store circuits in a database and thus the representation must provide enough information for the system to read instances of the primitive from the database. This can be as simple as making one database call or making multiple calls, one for the instance of the primitive, one for each attribute, and one for each terminal.

Display

The representation should provide enough information to allow the CAD system to be able to display the primitive. It is important to be able to display an item in error rather than just output the name of the item in error.

Approaches

There are two representation forms, the external form and the internal form. The external form is used by the user of the system and any tools that are not integral to the system (*i.e.* a browser, documentation generator). Thus the external form must be understandable, readable, and modifyable by the user. The internal form is used by the system and thus must be efficient and fit the algorithms used. For example, an external representation might have indirect pointers via names for representing connectivity (as in textual netlist formats), while a critic, which needs to be able to traverse the connectivity graph quickly, would have direct pointers between elements that are connected.

The representation must also be independent of the details of the underlying system. For example, if the underlying system is a rule-based system, none of the rule system syntax or semantics should be in the external representation. This facilitates changing the underlying system without changing the representation the user sees, and the user is not required to learn the specific syntax of the underlying system.

The definition of a primitive must contain several features, from the simple, such as the name, to the complex, such as permutability information and how to calculate attributes of the primitive. Primitives should have the following:

type: The classification of elements in the database is determined by their **type**.

terminals: Terminals are used to connect the element to other elements via nets. The critiquer must know what terminals exist and if they are logically or electrically equivalent.

attributes: primitives may be further classified based on what the values of their attributes are. For example, the primitive BJT may have the attribute TYPE with values NPN and PNP. The use of attributes are particularly important for use in finding higher-level structures and errors. For example, a CMOS inverter is made up of two MOSFETs, but with different attributes. Besides having simple attributes, like the TYPE example above, there can be complex attributes that can be functions of other attributes on the primitive. The simplest example is that of the attribute for the W/L ratio of a MOSFET. This attribute is composed of two other attributes, the width of the MOSFET and the length of the MOSFET.

The description of a primitive is straightforward and can be easily described by a simple language. Often an existing language can be used, sometimes a language from the domain. Many CAD tools use the SIM[26] or SPICE[60] netlist formats to ease the creation and modification of data and to allow the data to be used by other tools. There is no such language available for describing the information necessary for the primitive. Many critics use a language from an AI system[43, 42, 12, 59, 19]. To make the system independent of the underlying system, a simple language was developed. Examples of using the language are shown below:

```
(defprimitive type
   (terminals list of terminals)
   (permutable list of permutable terminals)
   (attributes list of attributes))

(defprimitive bipolar-transistor
   (terminals emitter base collector)
   (attributes area type))

(defprimitive mosfet
   (terminals gate drain source)
   (permutable drain source)
   (attributes width length (w/l (/ width length)) type))
```

While this language is simple and provides enough expressibility for defining primitives, it does require the person installing or using the critiquing program to define the primitives and enter them into the system. Another approach is to find a representation that matches one already in use in the design system that is already used by the design groups. This allows the definition of the primitives "by example". "By example" in this case means using the description of a primitive that has already be done for other parts of the system. The best case would be using the basic leaf-cell definitions used to design circuits. These leaf-cells have information about the type of the primitive, the terminals on the primitive (and any attributes the terminals might have, such as type and direction), permutability information, and attribute information.

Permutability

Many elements have sets of terminals that are electrically or logically equivalent. These terminals are called permutable. For example, the inputs to a **NAND** gate are logically permutable and possibly electrically permutable if you ignore capacitance and timing concerns[61], and the *source* and *drain* of a MOSFET are electrically permutable. Choosing the best way to represent elements and terminals is important[65, 66]. This determines the number of rules in the system, the complexity of the rules, the size of the database and the overall performance of the system. There are two basic formats: one is simple and allows for easily understood rules, and the other is more complex but allows for more compact rule sets when terminal permutability is needed. The simple case, to be called the non-permutable case from now on, represents each terminal on an element as an attribute of the element, so one item in the data base can describe an element. The second case, to be called the permutable case from now on, represents each terminal on an element as a separate item in the data base with pointers linking the element and its terminals.

Each of the following examples is in the format used by OPS5[19] for representing the rules. OPS5 is a production system that was used in this work for testing out ideas on permutability, representing structures and errors, and finding them (see Chapters 4 and 5). In OPS5, the database is called *working memory* and the rules and patterns used to match the database are called *productions*. A production has the following form[1]:

rule: (p *lhs* --> *rhs*)

lhs: *pattern+*

pattern: (*type* ^*slot-name slot-value*)

type: the name of a data type declared to OPS5.

slot-name: the name of a field in the data type *type*.

slot-value: *constant* — *variable* — *constraint*

constant: a constant value, the slot must have this value.

[1]See [19] for a more detailed description of the format.

variable: <variable-name>; if the variable has not been assigned the variable takes on the value of the slot. If the variable has been assigned, the slot must have this value.

constraint: conditionals (not equal, less than, greater than, *etc.*).

rhs: *action+*

action: operations to perform: add to the data base, modify data, delete data, output, request input, *etc.*

Using the OPS5 notation, the non-permutable representation for a MOSFET is:

```
(mosfet ^gate <gate> ^drain <drain> ^source <source>)
```

and the permutable representation is:

```
(mosfet ^source/drain <term> ^gate <gate>)
(terminal ^parent <term> ^type source/drain ^terminal <drain>)
(terminal ^parent <term> ^type source/drain ^terminal <source>)
```

The following rules show the patterns to match an **inverter** made of a depletion-enhancement MOSFET pair (see Figure 3.1). Since the non-permutable format does not allow permutable terminals, four rules must be used to match all possible terminal permutations (2 allowed permutations per MOSFET and 2 MOSFETs). In general, for the non-permutable format, if M gates with N permutable terminals need to be matched, $(N!)^M$ rules will be needed to match all combinations[2].

```
(p inverter-0
  (dmosfet ^gate <out> ^drain *vdd* ^source <out>)
  (emosfet ^gate <in> ^drain <out> ^source *ground*)
  -->
  (make inverter ^input <in> ^output <out>))

(p inverter-1
  (dmosfet ^gate <out> ^drain *vdd* ^source <out>)
  (emosfet ^gate <in> ^source <out> ^drain *ground*)
  -->
  (make inverter ^input <in> ^output <out>))
```

[2]For each gate with N permutable terminals the number of possible terminal permutations is $N!$ and since the terminals of each gate can permute independently of the others, each gate multiplies the effect, giving $(N!)^M$.

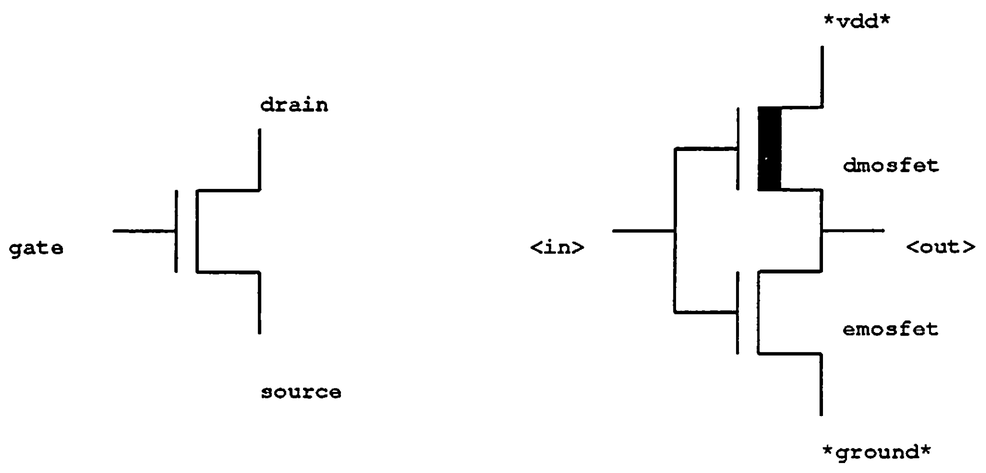


Figure 3.1: Inverter Example

```
(p inverter-2
  (dmosfet ^gate <out> ^source *vdd* ^drain <out>)
  (emosfet ^gate <in> ^drain <out> ^source *ground*)
  -->
  (make inverter ^input <in> ^output <out>))

(p inverter-3
  (dmosfet ^gate <out> ^source *vdd* ^drain <out>)
  (emosfet ^gate <in> ^source <out> ^drain *ground*)
  -->
  (make inverter ^input <in> ^output <out>))
```

The following rule is equivalent to the four **inverter** rules above, but uses the permutable format. Note that the **inverter** created by this rule uses the non-permutable format since it does not have permutable terminals.

```
(p inverter
  (dmosfet ^source/drain <sd1> ^gate <out>)
  (terminal ^parent <sd1> <term> << drain source >> *vdd*)
  (terminal ^parent <sd1> <> <term> << drain source >> <out>)
  (emosfet ^source/drain <sd2> ^gate <in>)
  (terminal ^parent <sd2> <term> << drain source >> <out>)
  (terminal ^parent <sd2> <> <term> << drain source >> *ground*)
  -->
  (make inverter ^input <in> ^output <out>))
```

The non-permutable format requires M working memory elements and $(N!)^M$ rules for M elements with N permutable terminals. The permutable case requires one rule and an extra working memory element for each permutable terminal, for a working memory size of $M \times (N+1)$. The RETE match algorithm used in OPS5[20] has a lower and upper bound of time per rule firing of $O(log_2 R)$ and $O(R)$ for rules, and $O(1)$ to $O(W^{2C-1})$ for working memory elements (where C is number of patterns in a rule, R is the number of rules, W is the number of working memory elements). As can be seen, increasing the size of the working memory has a much worse effect on *worst-case* performance than increasing the number of rules. Tests with OPS5 on circuit examples show this and show that better performance is obtained when more rules are added to the system rather than when more working memory elements are added to the system. Thus the non-permutable format is faster than the permutable format.

The following tables and graph show the results of experimenting with the two permutability formats. The experiment was to find latches that were composed of nand gates which were composed of MOSFETs. OPS5 on a VAX 11/785 was used for the experiments.

Count: the number of MOSFETs in the working memory.

WME count: the number of working memory elements at the start. The same as the MOSFET count for the permutable rules format, three times the MOSFET count for the permutable working memory format (1 for each MOSFET and 1 for each of the 3 terminals).

CPU time: the time to find all the structures.

firings: the number of rules that fired.

firings/sec: the number of rules fired per CPU second.

memory: the maximum amount of memory used by OPS5 while finding the structures.

For M subpatterns with N permutable terminals in each subpattern, there will be $(N!)^M$ pattern sets. Thus a basic CMOS inverter would have 4 $(2!^2)$ pattern sets. Most

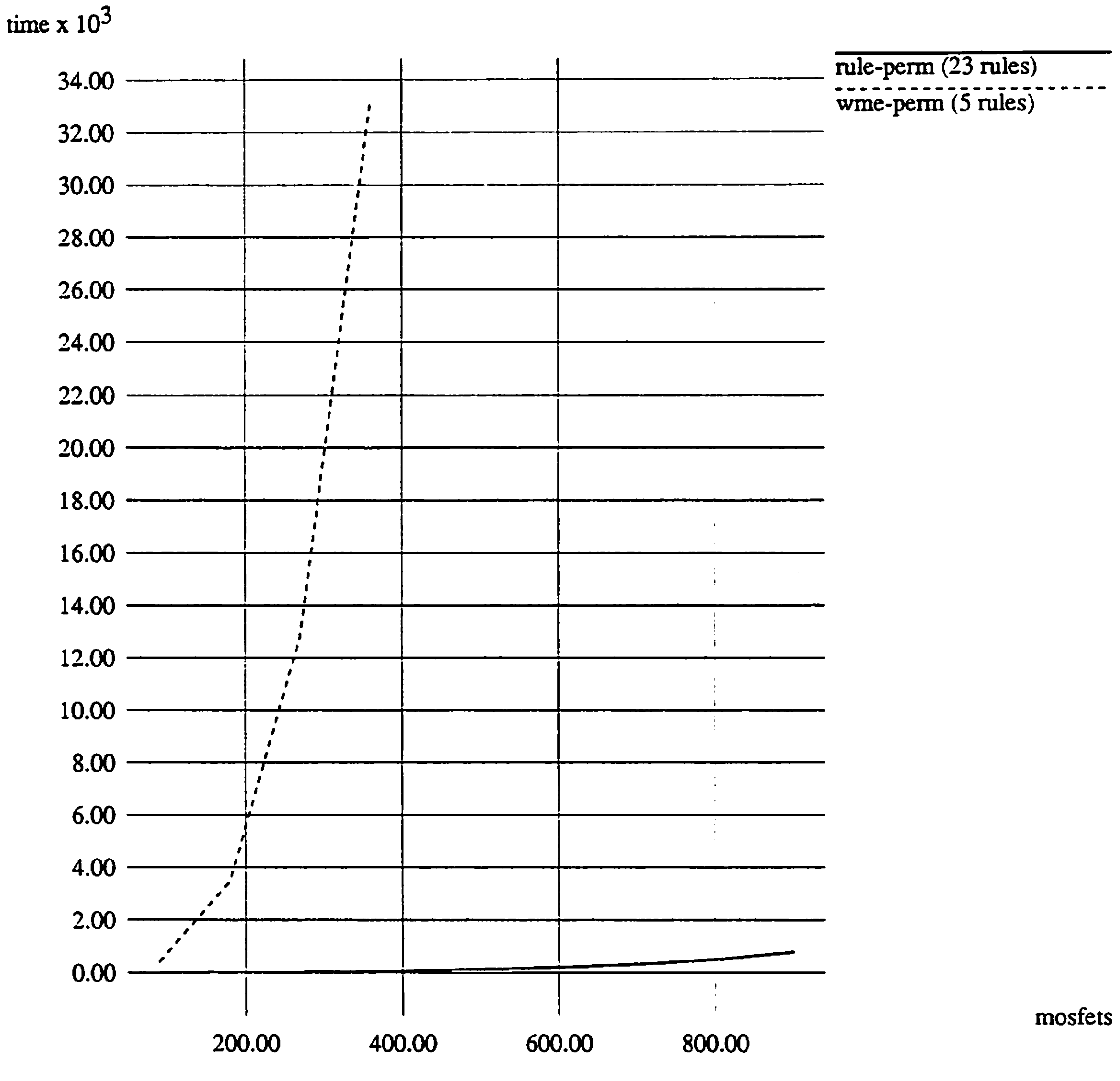


Figure 3.2: Time to Find NAND Gates and Latches

Count	WME count	CPU time (sec)	firings	firings/sec	memory (MBytes)
90	90	4.6	49	10.6	1.3
180	180	14.9	98	6.5	1.3
270	270	32.1	147	4.5	1.3
360	360	58.4	196	3.3	1.3
450	450	95.1	245	2.5	1.4
540	540	150.6	294	1.9	1.4
630	630	231.4	343	1.4	1.5
720	720	347.3	392	1.1	1.6
810	810	519.5	441	0.8	1.7
900	900	772.3	490	0.6	1.8

Table 3.1: Finding NAND Gates and Latches - Non-permutable Format (23 rules)

Count	WME count	CPU time (sec)	firings	firings/sec	memory (MBytes)
90	270	394.6	49	0.1	1.5
180	540	3466.1	98	0.02	2.0
270	810	12790.0	147	0.01	3.0
360	1080	33155.6	196	0.006	4.4

Table 3.2: Finding NAND Gates and Latches - Permutable Format (5 rules)

structure definitions contain two to four subpatterns that contain two permutable terminals, so the number of pattern sets does not grow too big ($2!^4 = 16$).

A point worth noting is that if some of the terminals that permute are not used in the pattern definition, then the number of patterns can be reduced. For example, if a pattern is looking for two MOSFETs that are connected together by their gates and is not concerned with what is connected to the source and drain terminals, then only 1 pattern is needed (see Figure 3.3). If information about how the source and drain were connected, 4 patterns would be needed.

Note that the ideas developed for permutability in this chapter apply equally well to the permutability of terminals in structures.

3.3 Built-in Primitives

While a circuit critic should not have knowledge of the different types of primitives for the various technologies and design styles (*e.g.* MOSFETs, registers), there are a

```
(p interesting-combination
  (emosfet ^gate <mid>)
  (emosfet ^gate <mid>)
  -->
  ...)
```

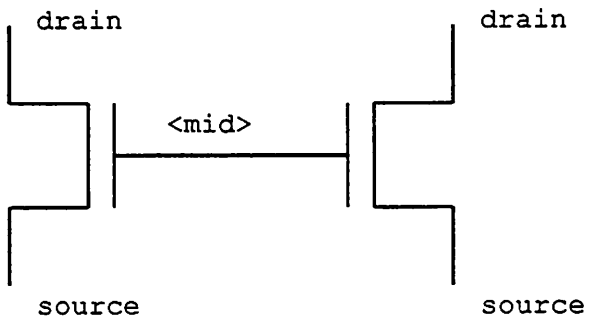


Figure 3.3: Unused Permutable Terminals

number of primitives that are independent of the particular design style or technology. To simplify many of the structure finders and error checks these primitives should be known to the critic. The ones of particular importance are: **Node, Port, Supply, Ground**, and **Clock**.

Nodes are used for tying together the various primitives defined for the technology and design style. The major purpose of the built-in node primitive is as a holder for node-based errors. Each technology or design style has a number of errors that are not associated with structures or primitive elements, but with the node itself. For example, no dc path to ground, multiple static outputs driving the same node, shorted supplies, or a node that is not testable.

Ports are used to connect the primitive elements and structure to the nodes. The purpose of a built-in port primitive is to allow for the special handling of external connections to circuits. These can simply be the formal terminals of a cell, the primary inputs and output of a logic network, or the pads for a chip. Many error checks are modified or do not apply based on whether the primitive element or structure being checked is connected to one of these ports. For example, no dc path to ground does not apply to the inputs of cells. Also, special error checks are required when the inputs/outputs are pads.

Supplies, **Ground**, and **Clocks** are used to represent the power supplies, and time dependent signals input the to circuit. The one problem with supplies and clocks is that they have to be generic enough to handle all possible types of supplies and clocks that might exist for a wide range of technologies and design styles. It could easily be argued that these are design style and technology specific and should be in the definition of the primitives for the particular knowledge base. In the work for this dissertation, they were built-in primitives, but in hindsight, this may have been a mistake. Some technologies have multiple supplies (*e.g.* ECL), and some have complex supplies (*e.g.* ramps, sinusoidal, piece-wise linear). Clocks are even far more complex. Some technologies have no clocks, and some have multiple clocks. Clocks can many different attributes: rise and fall time, duty cycle, pulse width, *etc*. Of the design styles with multiple clocks, there are questions about whether the clocks overlap or not. Trying to fit all these various attributes in a coherent manner into a single supply and a single clock primitive was too ambitious a design goal.

Chapter 4

Structures

4.1 Overview

The complexity of large circuit designs makes it difficult, if not impossible, to design at the level of the primitive components. Instead, primitive components are grouped together into higher level components ('structures') and these higher level components are grouped into still higher level components, and so on. This is the basis of hierarchical design. The goal of hierarchical design is to reduce design complexity by representing structures by a simple abstraction of their detail at the higher level and/or by re-using a particular structure (regularity). Structure extraction is the process of taking a collection of interconnected primitives and finding the higher level structures that were originally used in the design or finding new structures not intended or expected (such as unexpected storage elements). By extracting the structure of a design a designer can easily see if the circuit has been implemented correctly.

A major use of structure extraction is to simplify the process of finding errors or of verifying the correctness of a design. Most descriptions of errors in a design are not represented in terms of the low level primitives, but in terms of the structures that the primitives make up. Rather than have the error check specify how these structures are composed and be responsible for finding them, a structure extraction can be performed before the error checks or sometimes during the check, allowing the error checks to be specified more simply in terms of the structures.

4.2 Representation

There are many approaches to representing structures. However, before these approaches can be explored, the necessary basic concepts to be represented must be determined. These concepts can then be used to explore and compare various representations and implementations. Structures require the specification of the attributes associated with the structure and the attributes of the components of the structure, the terminal information of the structure, the connectivity and composition, how the structure can be stored in the database, and finally how it can be presented to the user.

Attributes

The information needed for the representation and the reasons for needing it are the same as that needed for primitives (see Chapter 3). The only difference is that most of the attributes for structures will be calculated based on the values of the attributes of the components of the structure. Thus a method of referring to attributes on the components is necessary.

Terminals

The information needed for the representation of terminals of structures is the same as that needed for primitives (see Chapter 3).

Composition

The representation must be able to describe the composition of the structure (*i.e.* what items make up the structure) and the constraints on the composition of the structure (*i.e.* connectivity, constraints on attributes).

Database Storage

If the structures are to persist across runs of the critic or if other tools are to be used to present the results of the critic, the structures must be stored in the database.

Display

Once the structures have been found, there must be a way to present this information to the user. If the system is part of a larger integrated system, the display of the structures may be taken care of once a method of making them persist is developed.

Approaches

Two approaches were explored for representing structures: textual and graphical. To explore the textual approach for representing structures, a language was developed for describing the structures. There were three portions to the description: finding the structure, calculating attributes for the structure, and use of the structure in finding higher level structures. The "how to find the structures" portion consisted of what primitives and other structures made up the structure being defined, what constraints each component had, how the components were interrelated by connectivity and attributes, and how the structures were used in other structure definitions.

An important feature of the language was to remove all system internal information and lower-level system syntax and semantics from the language. By removing this information, the language would be independent of a particular implementation. The features of the language are as follows:

name: The name of the structure (*e.g.* INVERTER, LATCH). This is used for documentation and for definitions of structures that contain structures of this type.

comments: Comments about the structure, *e.g.* "This latch operates based on....". This is purely for documentation.

components: Describes the components that make up the structure and how they are interelated. The components are described as a set of one or more patterns.

actions: Describes the actions to be performed when the structure is found in the database. Used mainly for the calculation of attributes of the new structure:

```
(let <w/l> (compute (<width> // <length>)))
```

terminals: A list of the external ports of the structure.

permutable-terminals: A list of the terminals of the structure that can permute.

structure: Describes the structure to be created when the components are found in the database.

The following is an example of a structure defined using the language:

```
(defstructure
  (name inverter)
  (components (depletion-mosfet (name <load>)
                                (drain *vdd*)
                                (gate <out>)
                                (source <out>))
              (enhancement-mosfet (name <driver>)
                                  (drain <out>)
                                  (gate <in>)
                                  (source *ground*)))
  (terminals input output)
  (actions (let <b-r> (compute (<driver:width> // <driver:length>)
                            // (<load:width> // <load:length>))))
  (structure (inverter (input <in>)
                       (output <out>)
                       (beta-ratio <b-r>))))
```

The language described above for structure definitions has the advantage of being human readable, however it is not based on an existing circuit description language or database format. In Heckel's book, *The Elements of Friendly Software Design*, "Speak the Users Language" is listed as one of his important principles if the design of successful user interfaces[30]. In describing structures to the critic, it would be advantageous to be able to use existing cell designs rather than having to enter descriptions of the cells in the structure description language. Structures should be 'described by example'. 'Described by example' means the ability to enter cell designs using standard graphics editors and have them be directly used by the circuit critic for structure definitions.

Processing

To simplify the definition of the structures, the order in which the definitions are stored and read by the critic should not be specified. Since some structure definitions depend on others they must be processed in a specific order, *i.e.* the items used in a definition must be defined before the definition that uses them. A simple levelizing routine is all that is needed to determine an ordering based on dependencies between structure definitions. The algorithm has a best case complexity of $O(n)$[1], where n is the number of structures defined, and a worst case complexity of $O(n^2)$[2]. Most rule sets would have $n < 50$ structures defined, so the time to process the rule set is acceptable[3]. The following pseudo-code shows a simple algorithm for levelizing the structure dependencies:

```
list = list of structure types;
ordered-list = empty;

foreach primitive definition {
    add primitive ordered-list;
}

forever {

   if list is empty return ordered-list;

   item = first element in list;

   if depend(item) subset-of ordered-list {
       /* dependencies have been satisfied */
       add-to-end item ordered-list;
   } else {
       add-to-end item list;
   }
}
```

[1]All structures are defined in the order of use by other structures, *i.e.* all structures are defined before they are used, no forward references.

[2]The structures are defined in the opposite order of use by other structures, *i.e.* structures are used before they are defined.

[3]If the time gets too large, a better algorithm could be used, the structures could be partially ordered, or the results of a levelizing could be saved for future use.

4.3 Finding Structures

In building a system for finding structures in the database, there are several questions to answer: should the search go top-down or bottom-up, should the search be rule-based or algorithmic; if rule-based, should the rules be ordered a priori or change as the database changes; should pattern matching be done; how should the pattern-matching be done; and should finding series-parallel groups of elements be treated the same as finding structures.

To minimize the amount of code that needed to be written to explore some of the ideas in representing and searching for structures, the system was originally built on top of OPS5. This allowed the use of an existing rule system and pattern matcher, along with a language for describing structures. However, after using OPS5, problems were encountered in two areas. First, translating a primitive or structure description to an OPS5 representation required the addition of OPS5 specific information. Not just the expected syntactic changes, but information regarding semantics. For example, OPS5 would allow both components of the following pattern (used for matching two parallel elements) to match the same element in the database:

```
(element ^top <net1> ^bottom <net2>)
(element ^top <net1> ^bottom <net2>)
```

the following additional information would have to be added to force the components to match different elements in the database:

```
(element ^name <name> ^top <net1> ^bottom <net2>)
(element ^name <> <name> ^top <net1> ^bottom <net2>)
```

Each additional component for `element` would require more complex checks:

```
(element ^name <> <name1> <> <name2> .... <> <nameN> .... )
```

Second, the OPS5 rule sequencing (conflict resolution) was different than the sequencing required for searching for the structures, so extra patterns and rules were needed to direct the searching of OPS5. Since there is no way to statically declare rule ordering in OPS5, the ordering was always determined at run-time. In order to get the desired rule order, the ruleset was split into 'rulesets'. Rulesets are usually set up by putting a 'ruleset' pattern in each rule. The ruleset pattern has one attribute which is the name of the ruleset.

In order to allow a ruleset to run, an element is placed in working memory that has the name of the current ruleset. To go to the next ruleset, the current element is removed and a new one with the name of the next ruleset is added. There are a couple of ways to do this sequencing of OPS5 rulesets: one is to have a set of rules that know how to switch from one rule set to another, *i.e.* one rule for each ruleset transition; another is to build a queue of rulesets and have a generic rule that pops a ruleset name off the queue and causes that ruleset to be run. This corresponds to having the control knowledge is in the data or in the ruleset. The following shows an example rule with a ruleset pattern and a rule for switching between rulesets. Both rules match data in the database, but OPS5 will always fire the one that is more specific, *i.e.* has more components. The first rule fires until there are no more inverters to find, then only the second rule matches the database and it is allowed to fire.

```
(p
    (ruleset ^name find-inverters)
    (mosfet ^type nmos ...)
    (mosfet ^type pmos ...)
    ...)


(p
    (ruleset ^name find-inverters)
    ->
    (delete 1)
    (make ruleset ^name find-latches))
```

The following shows the initializing of a stack of rulesets and the generic rule for popping the stack. This scheme works because OPS5 will use the most recently added item when it looks for rules to fire, thus rules from the most recently added ruleset will be considered first. This has the disadvantage that all rules in all rulesets are looked at to determine which one to fire.

```
(make ruleset ^find-latches)
(make ruleset ^find-inverters)

(p
    (ruleset)
    ->
    (delete 1))
```

Top-Down Versus Bottom-Up

The process of extracting structures can be a top-down or bottom-up process. In a bottom-up process the structures that are composed entirely of primitives are found and then structures that use the recently found structures (and possibly other primitives) are found, and so on. The contents of the database drives the process of extraction. In a top-down process, the search begins by looking for the set of top level structures (structures that are not used in any other structure descriptions), for each top level structure the critic searches for its component structures, and so on, until primitives are being searched for. For example, if the structure types to be searched for are latches and inverters, and the primitives are enhancement and depletion mode MOSFETs, then the top-down search would start out trying to find all latches. Since latches are composed of inverters and there are no inverters in the circuit, the inverter structure descriptions would be used to search for inverters. The constraints on the connectivity of the inverters in the latch would be passed down. The inverter descriptions are composed of MOSFETs, which are primitives, so the downward process stops. In a bottom-up search, all structures would be searched for, but only the inverter description would be used since it is composed entirely of primitives. Once inverters are found, the latch description may be used since its components have been found. In the bottom-up process, all structures of a given type can be found before going on to another, or as structures are found other structure descriptions that reference them can be used. For example, all inverters can be found before any latches are searched for, or as soon as an inverter is found, latches can be searched for.

Ordering

There are two ways to manage the rules used for finding structures. The list of rules to be fired and their order can be statically determined at the time that the knowledge base is loaded, or the list of rules and their order can be dynamically determined based on the knowledge base and current state of the database. In dynamic ordering, rules are added to a rule queue based on the ordering criteria along with information based on the current state of the database. Dynamic ordering only adds rules to the queue that stand a chance of firing; thus entire groups of rules may be skipped. A set of rules may depend on a certain type of structure in the database. If instances of the structure type are not in the database, dynamic ordering will never place the rules that use that set in the queue. If the patterns are expensive to evaluate this can be a large savings. In static ordering, all rules are placed in the rule queue based on the ordering criteria. This makes the rule firing algorithm simpler, since it no longer has to determine if new rules should be added to the queue, but it means that entire sets of rules that stand no chance of ever succeeding will be placed in the rule queue and will be rejected at run-time. Both orderings give the same final results. The tradeoff is in the overhead of dynamic ordering versus the extra time to check rules that will never fire in a static ordering.

Some rules can be written in a recursive manner, *i.e.* the rule creates a structure that could be used in the rule itself. This is mainly used for finding series and parallel combinations of elements, as described in the section on Combination finding.

The following pseudo-code shows the basic algorithm for sequencing through a dynamically ordered rule queue:

```
rule-queue <- empty

foreach primitive type (P) instantiated in the circuit {
  foreach structure type (ST) that contains instances of P {
    add the rule corresponding to ST to the end of rule-queue
  }
}

while (rule-queue is not empty) {
  success <- false
  while (first rule fires) {
    create new structure (S)
    success <- true
  }

  if (success) {
    if (combination defined for S) {
      schedule combination finder to be called after all
        recursive rules
    }
    foreach structure type (ST) that contains instances of S {
      add the rule corresponding to ST to rule-queue*
      if (rule is recursive) {
        add the rule to the beginning of rule-queue*
      } else {
        add the rule to the end of rule-queue*
      }
    }
  }
}
```

The scheme used for dynamic ordering allows rules to be added to the end of the rule queue (default) or to the beginning of the queue (for recursive rules). There is also the case where the rule should be inserted into the middle of the queue at some location. This occurs when a rule to be added is more specific than a rule already in the queue. If the new rule were added at the end of the queue, the less specific rule would match the elements that should have been matched by the more specific rule. If the new rule were added to the beginning of the queue it might not match anything since there might be rules later in the queue that create structures needed in the new rule. For example, in Figure 4.1 the group of MOSFETs that represent a 'transfer inverter' are in the database. If the MOSFET combination rules fire before the transfer inverter rule is checked, the transfer inverter rule will not find the structure in the database. Therefore the rule should be inserted into the queue. Instead of placing rules at the end of the queue they could be *insertion sorted*. Each rule would be assigned a value that corresponds to how "specific" it is. The information used to determine how specific a rule is can be the number of patterns in the rule, the number of variables, the number of constraints (shared variable bindings), *etc.*

```
; rule with shared variable bindings
;   five slots used, but only three variables
;   (?name is a variable)
(element1 (slot1 ?var1) (slot2 ?var2))
(element2 (slot3 ?val2) (slot4 ?var3))
(element3 (slot5 ?val3))
```

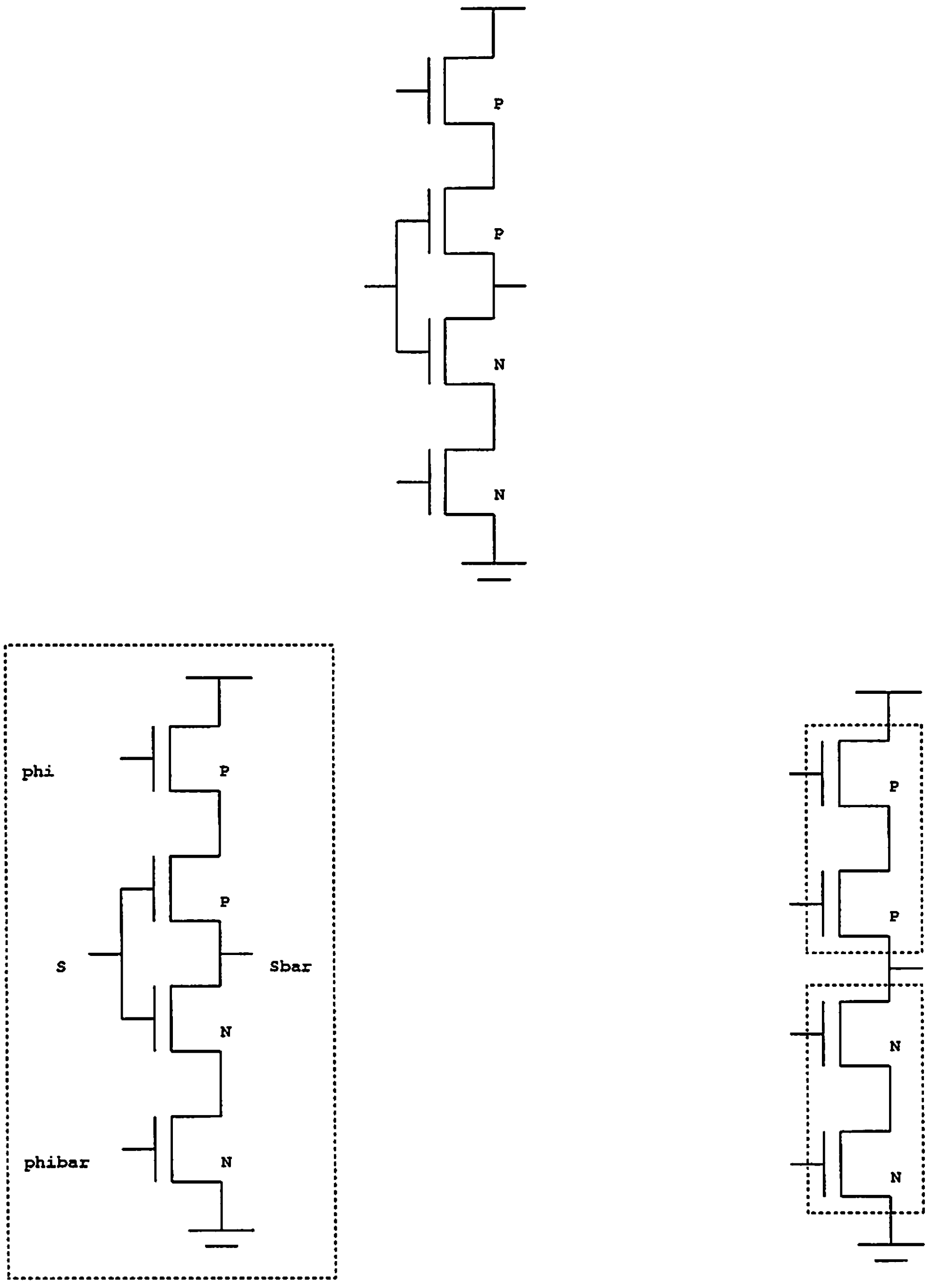


Figure 4.1: Insertion Sort Example

Pattern Matching

The process of determining whether the components necessary to form a structure are in the database can be done either by using patterns to search the database or writing code to find each type of structure. The trade-off is that writing a specific piece of code for each structure is much faster since more information can be encoded and no general purpose pattern matcher needs to be used, but this method requires writing code for each structure. A single piece of code may also cover what would require multiple patterns or multiple invocations of a pattern and thus can maintain information that would be lost between the patterns. Pattern matching is general and makes adding new structures easy.

The problem is to match a set of patterns against a database. A set of patterns is a group of interrelated patterns that are used to find a group of interrelated items in a database. Each pattern is composed of a *class name* and a set of slot descriptions. The *class name* is the type of the item to look for in the database (*i.e.* MOSFET, inverter). The slot descriptions are composed of a name and a value. The value can be either a constant or a variable (represented in the examples by a leading "?"). If the value is a constant, any item matching the pattern must have a slot with the same name and the value of the slot must be equal to the constant. If the value is a variable, any item matching the pattern can have any value and this value is associated (bound) to the variable. All variables with the same name in the set of patterns must be bound to the same value; this is called *unification*. Variables specify subpattern inter-connectivity (nodes) and inter-subpattern constraints (attributes). If the slot name is the name of a terminal, the value used is the node that is connected to the terminal. Note that variables are only used for inter-connectivity and inter-subpattern constraints, not for passing values of attributes to the action portion of a rule (as in OPS5[19], called the right-hand side). For the right-hand side one has full access to the items matched by the pattern. Figure 4.2 describes the syntax of the patterns.

The simplest approach for pattern matching is to match the first pattern against the entire database and for each possible set of variable bindings, match the next pattern against the entire database, taking into account the variable bindings. This method is $O(M^k)$ in complexity, where M is the number of elements in the circuit and k is the maximum number of patterns in a pattern set. The following pseudo-code shows the basic pattern matching algorithm[4]:

```
matcher(patterns, items)
{
    if (no more patterns) {
        /* match ok, perform actions on 'items' */
    }
    matches = match(head(patterns), database));
    if (no matches) {
        unbind all variables;
        throw back to the first call to matcher;
    }
    foreach (match in matches) {
        bind variables;
        matcher(rest(patterns), append(match items));
    }
}
```

There are two straightforward ways to reduce the search space. The first method is to partition the database into groups of like elements (all MOSFETs together, all inverters together, *etc.*). If the database is composed primarily of one class of items (such as in MOSFET circuits) the search space reduction can be small However, once collections of MOSFET have been classified into structures, the size of the structure classes can be small. The worst case pattern matching time is $O(M^k)$. As the circuit is partitioned, the pattern matching time improves: $M = \sum M_i$, and $M^k \gg \sum(M_i^{k_i})$. This method is $O(\sum(M_i^{k_i}))$ in complexity. The following pseudo-code shows the improved algorithm:

```
matcher(patterns, items)
{
    if (no more patterns) {
        /* match ok, perform actions */
    }
   matches = match(head(patterns), class-instances(head(patterns),
                  database));
    if (no matches) {
        unbind all variables;
        throw back to the first call to matcher;
```

[4]For those familiar with LISP (the implementation language for this work), `head` is `car`, `rest` is `cdr`, and `append` is `cons`.

```
    }
    foreach (match in matches) {
        bind variables;
        matcher(rest(patterns), append(match items));
    }
}
```

In the case of finding circuit structures, the interconnectivity of the structures (represented by the variables that match nodes) can be used to reduce the search space. Once the first pattern is matched, the nodes represented by the variables can be used to prune the search space: only the elements connected to those nodes need to be searched. So, in the second and succeeding steps, instead of searching the entire database again, the nodes that have been bound to variables that are in the next pattern are searched for elements of the proper class and the elements connected to the node with the smallest number of elements connected to it are searched. There are usually three elements connected to a single node, as compared to tens, hundreds, or thousands of elements in the entire circuit. It is important to make sure the node with the smallest number of connections is chosen; while most nodes have around three connections, some nodes, such as the supply, clock, and ground nodes, may have hundreds or thousands of connections. A smaller group of elements to be searched could be obtained by intersecting all the node lists, but the intersection cost probably outweighs the small cost of the extra search[5]. This method is $O(\sum(M_i \times min(connections)^{k_i}))$ in complexity. The following pseudo-code shows the algorithm:

```
matcher(patterns, items)
{
    if (no more patterns) {
        /* match ok, perform actions */
    }
    matches = match(head(patterns),
                    items-connected(head(patterns)));
    if (no matches) {
        unbind all variables;
        throw back to the first call to matcher;
    }
```

[5]However, if the smallest element count on the nodes is large and the other node lists are different, intersection could reduce the time considerably.

```
    foreach (match in matches) {
        bind variables;
        matcher(rest(patterns), append(match items));
    }
}

items-connected(pattern)
{
    if (notany(variable-is-a-node, variables(pattern))) {
        return(instances(class-of(pattern)));
    }
    /*
     * build a list of elements of the right class
     * connected to the node with the smallest element
     *.count
     */
}
```

As an example, the following is a simple set of patterns for matching an inverter from MOSFETs:

```
(mosfet (drain ?d) (source ?vdd) (type "pmos"))
(mosfet (drain ?d) (source ?gnd) (type "nmos"))
```

The first step is to match the first pattern against all MOSFETs in the database:

```
(mosfet (drain ?d) (source ?vdd) (type "pmos"))
```

This will find a set of bindings for **?d** and **?vdd**. The next step, for each bound pair of **?d** and **?vdd**, will find all elements of type MOSFET connected to the node bound to **?d**. **?vdd** is not used since it does not appear in the second pattern, and **?gnd** is not used since it has not been bound. This list is then matched against the second pattern (with **?d** replaced by its binding):

```
(mosfet (drain NODE-12) (source ?gnd) (type "nmos"))
```

This list will probably contain less than 4 elements[6], rather than all the MOSFETs in the circuit.

[6]In circuit simulation literature[56], the fanout from an element is usually stated to be in the range of 2.5 to 3. This also can be shown in the number of non-zero elements per row in the connectivity matrix. There is also literature on wiring space requirements[31] that backs up these numbers.

Indexing

Another technique for use in reducing the search space is storing information about the values of slots so that all elements with a particular value for a slot can be quickly accessed. This is known as *indexing*[18]. For finding structures, indexing the nodes makes sense since the components of structures are connected by nodes. This can be taken further by indexing more fields in the elements (such as **type** in MOSFETs). This could considerably cut down the search space if the number of values that the field can take on is large and uniform (*e.g.* in the case of MOSFET type for static CMOS, the number of values is 2 and uniform, which could cut down the search by a factor of 2 for each subpattern in the pattern). There is a trade-off between the time to calculate the indexes (*i.e.* go through all elements and hash them based on the values of the indexed slots) and the time to search the entire set of elements at match time to find elements that match the particular slot value.

Additional Constraints

Besides the constraints associated with interconnectivity and fixed slots values (handled by indexing), there are extra constraints that are not easily handled as patterns and are usually escaped to the underlying system. In the work presented here, the implementation of the matcher is in LISP and thus the constraints can be arbitrary LISP expressions that have access to the variables in the patterns. For example, the following pattern would more precisely define an inverter:

```
(mosfet (drain ?d) (source ?vdd) (type "pmos"))
(mosfet (drain ?d) (source ?gnd) (type "nmos"))
(lisp-eval (supply-node-p ?vdd))
(lisp-eval (ground-node-p ?gnd))
```

Pattern Ordering

Note that ordering the pattern set can speed up the match; the following ordering of the above pattern set would speed it up by immediately throwing out a bad binding of the supply node (this assumes that the **lisp-eval** is faster than pattern matching):

```
(mosfet (drain ?d) (source ?vdd) (type "pmos"))
```

$$\begin{aligned}
\langle \mathit{pattern} \rangle &::= (\langle \mathit{subpattern} \rangle +) \\
\langle \mathit{subpattern} \rangle &::= \langle \mathit{pat} \rangle \mid \langle \mathit{lispEval} \rangle \\
\langle \mathit{pat} \rangle &::= (\langle \mathit{type} \rangle\ \langle \mathit{attribute} \rangle +) \\
\langle \mathit{type} \rangle &::= \mathit{lispSymbol} \\
\langle \mathit{attribute} \rangle &::= (\langle \mathit{aName} \rangle\ \langle \mathit{aValue} \rangle) \\
\langle \mathit{aName} \rangle &::= \mathit{lispSymbol} \\
\langle \mathit{aValue} \rangle &::= \langle \mathit{constant} \rangle \mid (\mathsf{variable}\ \langle \mathit{varName} \rangle) \\
\langle \mathit{constant} \rangle &::= \mathit{lispValue} \\
\langle \mathit{varName} \rangle &::= \mathit{lispSymbol} \\
\langle \mathit{lispEval} \rangle &::= (\mathsf{lisp-eval}\ \mathit{lispExpression})
\end{aligned}$$

Figure 4.2: Internal Pattern Representation

```
(lisp-eval (supply-node-p ?vdd))
(mosfet (drain ?d) (source ?gnd) (type "nmos"))
(lisp-eval (ground-node-p ?gnd))
```

Also, the pattern that matches the class with the least number of instances in the database should be at the beginning; this reduces the search space. If the number of inverters in the database is less than the number of MOSFETs, the following pattern set should have the two patterns switched (*i.e.* fewer bindings to throw out):

```
(mosfet (drain ?in) (source ?mid) (type "nmos"))
(inverter (input ?mid) (output ?out))
```

Figures 4.3 and 4.4 show a CMOS inverter and a pattern that represents it.

Finding Combinations

Combinations are collections of series, parallel, and series-parallel connected elements (see Figure 4.5). Most MOSFET designs are composed of large collections of series,

```
((mosfet (type "PENH")
         (member-of nil)
         (drain (variable G002))
         (gate (variable G034))
         (source (variable G045)))
 (mosfet (type "NENH")
         (member-of nil)
         (drain (variable G045))
         (gate (variable G034))
         (source (variable G010)))
 (lisp-eval (supply-node-p
              (node (variable-value "G002"))))
 (lisp-eval (supply-node-p
              (node (variable-value "G010")))))
```

Figure 4.3: Pattern for a CMOS Inverter

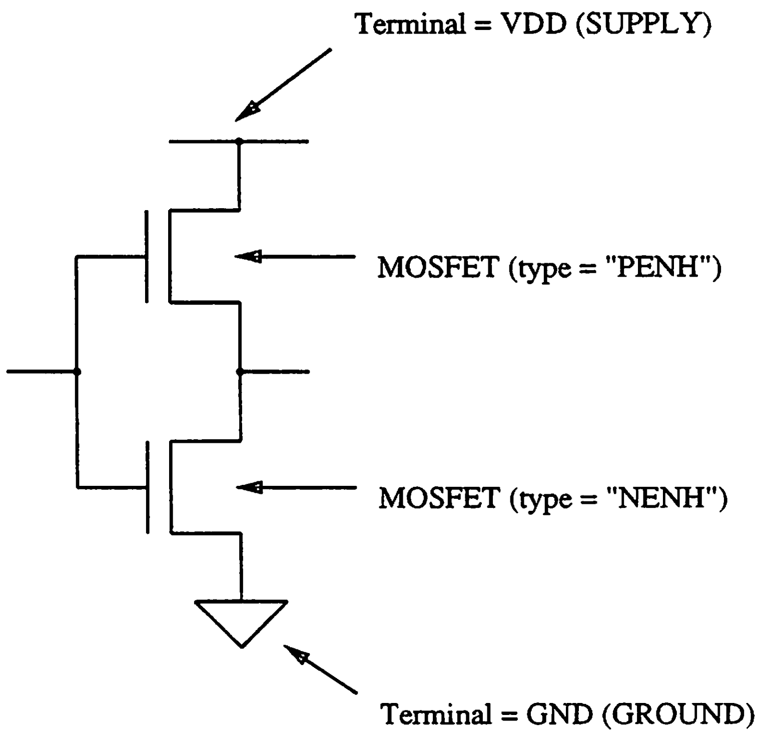


Figure 4.4: Pattern Example

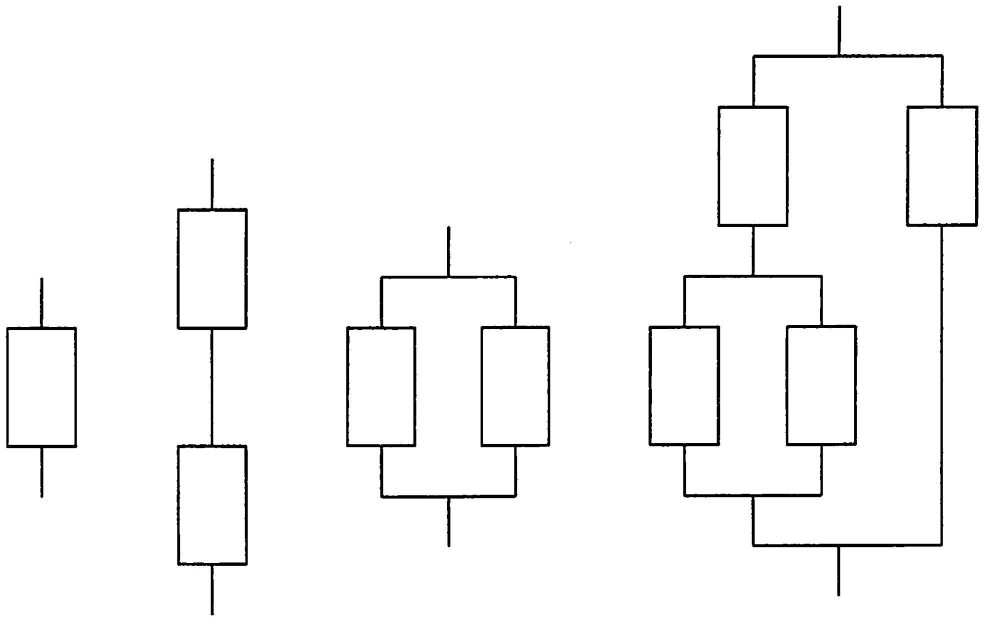


Figure 4.5: Example Combinations

parallel, and series-parallel connected MOSFETs that form logic functions. Two techniques have been explored for finding combinations: pattern-matching and algorithmic.

The following code fragments show how combinations can be found. Pattern-matching is the simplest way to find combinations but is time consuming. Using patterns, the combinations are found two elements at a time and therefore a sequence of pattern matches is needed to build up the entire combination. Using a sequence of pattern matches has the problem that all information (state) gained during the previous match is lost to the next one.

The following patterns can be used for matching parallel combinations:

```
; two elements in parallel
(element (term ?top) (term ?bottom))
(element (term ?top) (term ?bottom))

; adding an element to an existing parallel combination
(element (term ?top) (term ?bottom))
(element-parallel-combination (term ?top) (term ?bottom))
```

The following patterns can be used for matching series combinations:

```
; two elements in series
(element (term ?top) (term ?middle))
(element (term ?middle) (term ?bottom))

; adding an element to an existing series combination
(element (term ?top) (term ?middle))
(element-series-combination (term ?middle) (term ?bottom))
```

With permutability taken into account, these patterns can grow to four patterns for each basic pattern.

Since the patterns can be recursive, that is, a pattern to find a series-combination can have another series-combination as a component, the ordering of the combination finding rules must be different than the ordering used for the structure finders. For structures, the rules are added at the end of the rule queue. In the case of combination finders, they should be continually fired until there are no more matches. As an example, in Figure 4.6 the dynamic-gate rule needs a MOSFET combination, and there are many sub-combinations of the maximal MOSFET combination that fit the constraints put on the MOSFET combination by the dynamic-gate pattern. Therefore the maximal MOSFET combination must be found before the dynamic-gate rule can be fired. One method of doing this is to define a pattern representation that allows for the specification of the maximal pattern. However, the method explored in this work is to have simple two element patterns and have the rule system continually fire the patterns until no more combinations can be found. The rule scheduling system can detect that the rule being scheduled is recursive and place it at the beginning of the rule queue rather than at the end. Note that this implies the use of a dynamic rule ordering system or the partitioning of the ruleset into recursive and non-recursive rules.

The problem is further complicated by the fact that the finders for the two types of combinations, series and parallel, must cycle. After finding all series combinations and parallel combinations, the series and parallel finders must be used again to find the more complicated series-parallel and parallel-series combinations. For example, in Figure 4.7, the first parallel rule will find no parallel combinations, the first series rule will find MOSFETs A and B in series, the next parallel rule will find MOSFET C in parallel with the previously found combination (AB), and finally the last series rule will find MOSFET D in series with

```
; find dynamic gate
(mosfet (type "NMOS") (gate ?clock) (source ?vdd) (drain ?out))
(combination (type "NMOS") (top ?out) (bottom ?gnd))
```

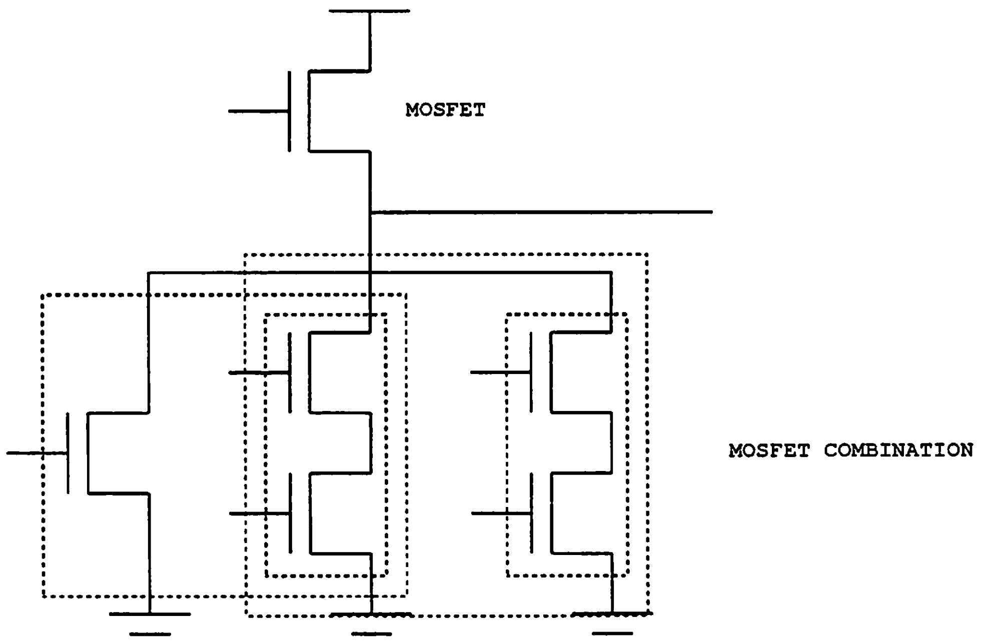


Figure 4.6: Recursive Rule Ordering (Pattern and Example)

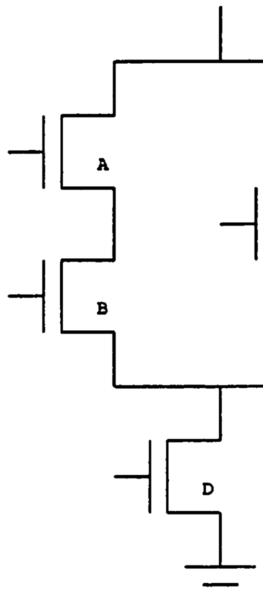


Figure 4.7: Sequencing of Combinations

the ABC combination.

The algorithm used to explore finding combinations is to try to find as many parallel combinations as possible, then try to find as many series combinations as possible and repeat until no more combinations are found.

```
find-combinations()
{
    find-parallel-combinations();
    find-series-combinations();
    if (combinations have been found) {
        find-combinations();
    }
}
```

Parallel combinations are found by picking an item of the type used to build the combination (either the primitive of the combination or another combination) and looking at all other elements of the same type connected to it in parallel. In order to speed up the search, the terminal with the smallest number of connections is used for the search. This process is repeated for each item of the proper type.

```
find-parallel-combinations()
{
    foreach instance of the desired type (element/combination) {
        if not already a member of a combination or structure {
         find the terminal which the fewest connections (top/bottom)
            foreach item connected to that terminal {
                if the item's other terminal connected to the
                    instances other terminal {
                    verify the constraints;
                    make part of the combination;
                }
            }
        }
    }
}
```

Series combinations are found by picking a node with only two elements not already in a combination and making sure they are of the same type, either the primitive of the combination or another combination. This is repeated for each node in the circuit.

```
find-series-combinations()
{
    foreach node in the circuit {
        if the number of free elements connected to the node is 2
            and it is not a dc node {
            get the two items;
            make sure they are not equal;
          make sure elements of the element or combination class fit
                the constraints;
            make a series combination;
        }
    }
}
```

Note that there are other forms of combinations that this scheme does not handle. Some combinations do not fit the basic series-parallel groupings (see Figure 4.8). These are called *ill-formed* combinations. Describing how to find these and how to calculate attributes as they are found was not addressed in this work.

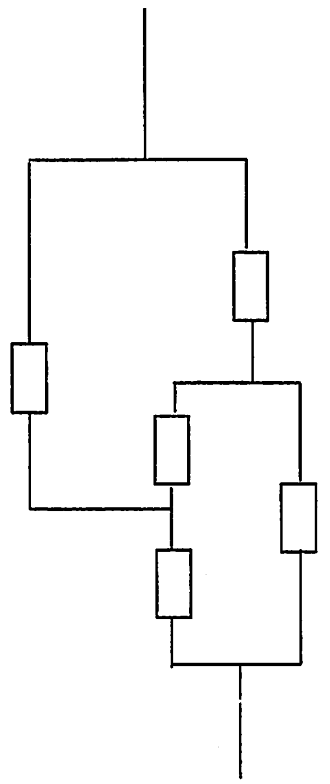


Figure 4.8: Ill-Formed Combination for Series-Parallel Design Styles

Calculated Attributes

For primitives, most attributes are simple and do not depend on other attributes, such as the type of a device or the direction associated with the terminals. For structures, most attributes are computed based on the components of the structure, such as the beta ratio of a static MOSFET inverter or the capacitive loading of a dynamic CMOS gate. The choice to make in handling calculated attributes is when to calculate them and whether to store them. They can be calculated and stored at the time the structure is found. This simplies the process, but means that attributes that will never be used in other structures or error checks may be calculated. Instead of calculating the attributes at the time the structure is found, the attribute can be calculated when it is used. This increases the code complexity[7] but means that only attributes that are really used will be calculated. Once the attribute is calculated it can be saved for later use or it can be recalculated on each use. Saving the value requires a minor increase in code complexity with significant benefits.

If attributes are not calculated at the time the combinations are found, but only as needed (and then optionally cached), the combinations must be traversed to calculate the values. A simple top-down, depth-first algorithm is shown below:

```
traverse-combination()
{
    if primitive get the attribute and return it
    if series combination {
        call traverse-combination on each item in the series
            combination and combine the attributes
        return the composite attribute
    }
    if parallel combination {
        call traverse-combination on each item in the parallel
            combination and combine the attributes
        return the composite attribute
    }
}
```

[7]This can be alleviated somewhat by object-oriented languages, such as CLOS[6].

Explanation-Based Generalization

Currently the use of combinations in structure definitions requires the use of synthetic elements that do not exist in normal cell libraries. These synthetic elements are recognized by the critic as representing combinations. The critic examines the attributes of the synthetic element to determine what type of combination, the components, and how to calculate attributes of the combination. A better way to do this would be to allow the designer of the knowledge base to enter several examples of structures that **Critic** could use to generalize into the structure definition with combinations. A technique that was explored was to use an extension of *Explanation-Based Generalization*[54, 13]. Explanation-Based Generalization (EBG) is a technique for learning what determines sufficient conditions for a concept by generalizing an explanation of why a particular example of the concept meets the definition of the concept.

A key point is that Explanation-Based Generalization uses a single positive example to determine a generalization. The setup consists of a *goal concept*, a *domain theory* (usually a set of horn clauses), an *example* (described in terms of terminals in the clauses of the domain theory), and an *operationality criteria* (usually stating that the resulting generalization must be in terms of the domain theory).

The first step is to *explain* why the example is a member of the concept. This involves back-chaining from the goal concept until the backtracking matches the example. Once this is done the explanation is *generalized* by regressing (following the explanation in reverse and *variablizing*[8]) along the back-chaining path. At the end only the domain and input language predicates used in the explanation will exist with as many parameters variablized as possible. The generalization of the explanation does not change the structure of the explanation (no new branches, no pruning), it only variablizes it.

EBG has a couple of limitations. A single positive example; thus no errors, no negative examples, no multiple examples. A single positive example must be enough to fully describe the concept; if not the generalization only comes out correct via the *clever*

[8]Variablizing is the process of replacing constants or expressions with variables. For example, if you want a pattern that matches both "I program in LISP" and "I program in C", you would variablize and come up with the pattern "I program in ?LANG". The variablization may also include additional constraints, such as "?LANG must be one of (LISP, C)".

selection of the goal concept, domain theory, example and operationality criteria. Another limitation is that the operationality criteria always states that the generalization must be in terms of the example attributes and the domain theory. After thinking about some of the straightforward ways to do the explanation, this is the only way that the operationality criteria can be described and keep the basic ideas intact. The operationality criteria is really implicitly built into the goal, example and domain theory; it is redundant.

An Extension to EBG

EBG can produce over- and undergeneralizations because it forms a generalization based on only one example. With some assumptions, some simple extensions can be made to EBG to allow the use of multiple examples. The EBG algorithm is applied to each example, then similarities (and differences) in the attributes inside of each generalization and between generalizations are found and used to form a generalization over all examples. This is a simple addition of *similarity-based* generalization to EBG. This is discussed by Mitchell in [54], and is implemented by Lebowitz in UNIMEM[45, 44].

The Extension Applied To Static Gates

To explore the concepts of EBG and how it might apply to a circuit critic, an experiment to determine static gate representations from example static gates was performed. To find a static gate generalization, each example is parsed into two *groups*. A *group* is a series-parallel or parallel-series collection of mosfets of the same type (NMOS or PMOS). The standard EBG algorithm is performed on each parsed example, then similarities in each generalization and between generalizations are found and a single generalization is formed. The similarity checks that work well in the static gate problem are *equality* and *inequality*, there are no numeric-valued attributes.

For static gates, a basic EBG setup is (there are many different goal concepts possible, this uses a functional goal and a structural operationality criteria) given:

- *Goal Concept:* Class of objects, such that $staticGate(X)$, where $staticGate(X) \Leftrightarrow pullUp(F, X) \wedge pullDown(FB, X)$

- *Input: (pre-parsed groups)*

  $group(and, vdd, n1, pmos)$

  $group(or, n1, gnd, nmos)$

- *Domain Theory: (mapping from structural to functional)*

  $pullUp(F, X) \Leftrightarrow group(F, vdd, X, pmos)$

  $pullDown(F, X) \Leftrightarrow group(F, X, gnd, nmos)$

- *Operationality Criterion:* Concept definition must be expressed in terms of constructs usable by the pattern matcher (*i.e.*, structural descriptions).

An individual generalization would look like:

$staticGate(x) \Leftrightarrow group(and, vdd, n1, pmos) \ \wedge \ group(or, n1, gnd, nmos)$

The modification begins here; the attributes from each generalization are collected in lists, one list for each generalization (groups within each generalization have been imposed to simplify the noticing of similarities within a single generalization). So, if the three example generalizations are:

$group(and, vdd, n1, pmos) \ \wedge \ group(or, n1, gnd, nmos)$

$group(or, vdd, n2, pmos) \ \wedge \ group(and, n2, gnd, nmos)$

$group(and, vdd, n3, pmos) \ \wedge \ group(or, n3, gnd, nmos)$

The lists are:

$[[and, or], [vdd, n1, gnd], [pmos, nmos]]$

$[[or, and], [vdd, n2, gnd], [pmos, nmos]]$

$[[and, or], [vdd, n3, gnd], [pmos, nmos]]$

If the lists are abstracted to:

$[[v1, v2], [v3, v4, v5], [v6, v7]]$

The similarities (and differences) are:

$v1 \neq v2$

$v3 = vdd$

$v4 is a variable$

$v5 = gnd$

$v6 = pmos$

$v7 = nmos$

so the solution would be:

$group(F1, vdd, X, pmos) \wedge group(F2, X, gnd, nmos) \wedge F1 \neq F2$

Assumptions Made in the Extension

EBG and the extensions to EBG make some assumptions about the examples and the solution:

- Overgeneralizations are due to bad regression, not bad explanation. This is an assumption of the extension.
- There should be no errors in the examples given to the system. This is an assumption of EBG.
- The examples given to the system should all be able to be parsed into series-parallel and parallel-series combinations; they should not be *ill-formed* (this is the *no errors* condition). This is an assumption of the particular problem.
- The examples should not be too similar; if they are the similarity section may notice similarities that are not part of the concept, just artifacts of the examples, see Figure 4.9. This is an assumption of the extension (one that could be removed, see Chapter 8).
- To simplify the extension, groupings of attribute inside of a generalization were imposed to help in noticing similarities. This can easily be removed, but may cause the system to add unnecessary conditions (*i.e.* for the static gate example, the function attribute is never equal to the middle node attribute). This is an assumption of the extension (one that could be removed, see Chapter 8).

To test out these ideas, a simple PROLOG implementation was developed. The MOSFET groups were represented explicitly rather than having a set of PROLOG clauses for parsing the connected MOSFETs. Also, an explicit set of variablizations was coded:

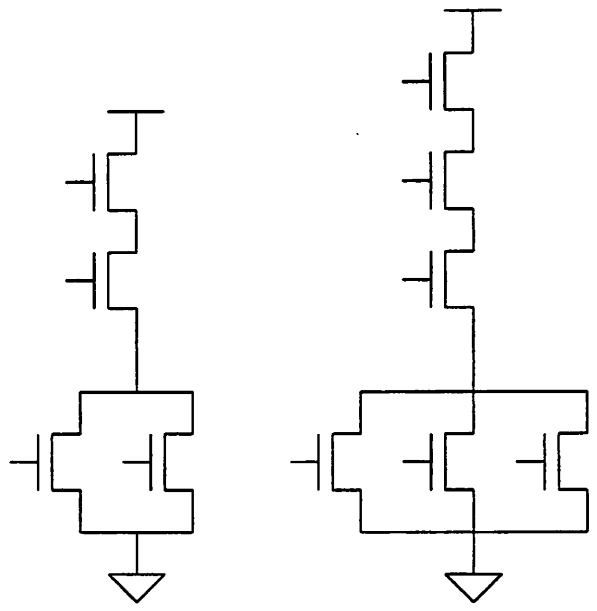


Figure 4.9: Examples that are Too Similar

```
%
% goal concept
%
staticGate :- pullUp(F, X), pullDown(FB, X).

%
% domain theory
%
pullUp(F, X) :- group(F, vdd, X, pmos).
pullDown(F, X) :- group(FB, X, gnd, nmos).

%
% example groups
%
group(and, vdd, n1, pmos).
group(or, n1, gnd, nmos).
```

```
%
% build a list to do similarity checking
%
list([[V1, V2], [V3, V4, V5], [V6, V7]]) :-
    group(V1, V3, V4, V6), group(V2, V4, V5, V7).

%
% example set of similarity and difference
% noticers (variablization)
%
notice(eq, combinationType)  :- list([[V, V], _, _]).
notice(neq, combinationType) :- list([[V, Y], _, _]),
                                not(V = Y).
notice(eq, firstNode)        :- list([_, [V, _, _], _]),
                                list([_, [V, _, _], _]),

%
% find an explanation
%
staticGate?

%
% find the similarities and differences
%
notice(X, Y)?
```

Over- and undergeneralization problems of EBG are best attacked by using more positive examples (not by using negative examples - it also turns out that multiple positive examples are easier to deal with). Using this observation, it was determined that adding a similarity based approach at the end of the EBG algorithm would be a good technique for the problem (this is also one of the recommendations given by Mitchell at the end of [54]).

There are two observations about EBG: (1) the operationality criterion is contained within the other parts of the setup (goal concept, example and domain theory), and (2) that to do anything *useful*, the domain theory must have a lot of information (the simple example in this experiment and the simple examples in [54, 13] do not have much domain theory, but then they are not *useful*). This last point shows that in order to learn, it takes a lot of knowledge.

4.4 Displaying Structures

Most critics display the results of the structure finding phase outside of the context of the circuit being critiqued. Their output is a textual description of the structures found. Since the data is generally not read from a data base that allows arbitrary annotation, this is the only way to display the data. There is the added complication that circuits to be critiqued may have been extracted from mask level designs and the names assigned to nodes and instances are machine generated and relate no information to the designer. Programs could be written to read in the circuit description and the critic output to produce a combined output, but this has not been the case. In the work presented in this dissertation, annotation of the circuit in the data base with the structure information and graphical display of the results were explored. This follows another of Heckel's principles for user interface design, "Communicate Visually"[30].

Chapter 5

Errors

5.1 Overview

The primary purpose of circuit critics is to find errors in the circuit design. There are two types of errors that can be flagged: something that should not exist, *e.g.* an illegal XOR configuration, or a set of attributes that violate a constraint, *e.g.* a latch that can not change state because of a weak load gate (see Figure 5.1).

5.2 Representation

Before performing experiments that test various forms of representing error checks, the components of the error check must be determined. Once the components necessary

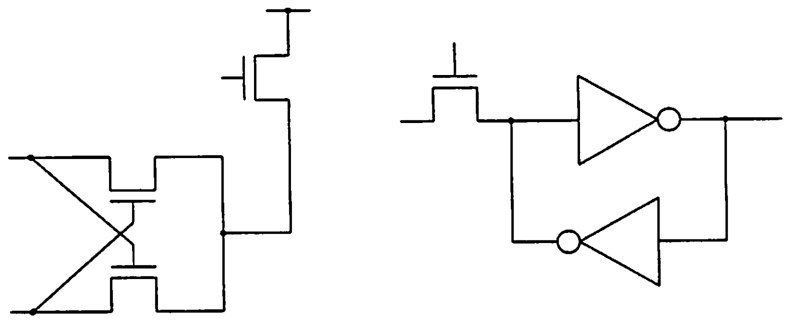


Figure 5.1: XOR and LATCH

for the representation of an error check have been determined, then the various forms of representation can be explored.

Finding

The representation must specify structure(s) the error check should be applied to. Depending on the system, this may also include how to locate the structure(s) that the error check applies to, as in [47, 3].

Check

The main component of the error description is the actual check to be performed. Normally this check is some set of constraints that if satisfied signals an error. Since the error check evaluates some set of constraints, the representation must specify how the constraints are interpreted and evaluated.

Database Storage

If the errors are to persist across runs of the critic or if other tools are to be used to present the results of the critic, the errors must be stored in the database.

Display

The display of an error includes the item that is in error (either a primitive or a structure) and information about the error check itself. The information about the error might be based on the values of various attributes of the item that is in error.

Approaches

In general, since designers do not describe errors, there can be no "described by example" error checks. In previous circuit critics, error checks were monolithic; how to find an error, how to check the error, and how to mark the data base when the error was found was contained in the error description. In addition, information specific to the internals of the implementation found its way into the error descriptions. This combination make the errors difficult to understand.

To make error checks easy to understand, create, and edit, they should be represented as a collection of features ('slots') Instead of the implicit knowledge of other critics, the knowledge should be explicit and internals knowledge should be removed. Each slot now has a single piece of knowledge rather than some clump of knowledge.

The first representation explored was text based, as were the first representations explored for primitives and structures. The definition of an error consisted of the `deferror` construct. Each `deferror` construct contained several fields:

name: The name slot gives the name of the structure, *e.g.* BAD-W-OVER-L, RACE-CONDITION.

comments: The comments slot is used to make comments about the structure, *e.g.* "Check for this and that, assumes this."

structure: This slot is composed of a pattern that describes the structure to be checked.

tests: This slot is composed of one or more tests to be performed to check if the *structure* is in error. Is the structure is inherently bad (it's mere existence is bad), then this slot is not needed.

description: This slot is a string that describes the error in a textual format. The string may contain variables matched in the *structure* slot.

For example:

```
(deferror
    (name name-of-the-error)
    (comments "comments")
    (structure structure-to-look-for)
    (tests test-to-perform-on-the-structure)
    (description "description-of-the-error"))

(deferror
    (name beta-ratio)
    (structure (static-gate (name <gate>)))
    (tests (<gate:beta-ratio> < MIN-BETA-RATIO)))
```

```
(deferror
  (name beta-ratio-check)
  (structure (static-gate (name <item>)))
  (tests (< <item:beta-ratio> $MINIMUM-BETA-RATIO$))
  (description "beta ratio error for <item>"))
```

would be converted to:

```
(defrule
  (name beta-ratio-check)
  (conditions (static-gate (name <item>)))
  (tests (< <item:beta-ratio> $MINIMUM-BETA-RATIO$))
  (actions (make error
              (item <item>)
              (description "beta ratio error for <item>"))))
```

This basic description has evolved to include more information about errors. The representation breaks up the error check into several individual features (or slots):

name

This slot represents the name of the error check The name is used when reporting errors to the user and for documentation purposes. Examples of names are ``Bad N and P Core Connections'', ``No DC Path To Ground'', and ``Charge Sharing Error''.

structure

The value of the **structure** field is the name of a primitive or structure type that the error check applies to. **name** and **structure** are used for uniquely representing an error check when marking items that should not be checked for the error. Examples of structure values are MOSFET, STATIC-GATE, and DOMINO-GATE. To remove the context information from the error, this slot is not directly set, but is inferred from the error's association with a structure.

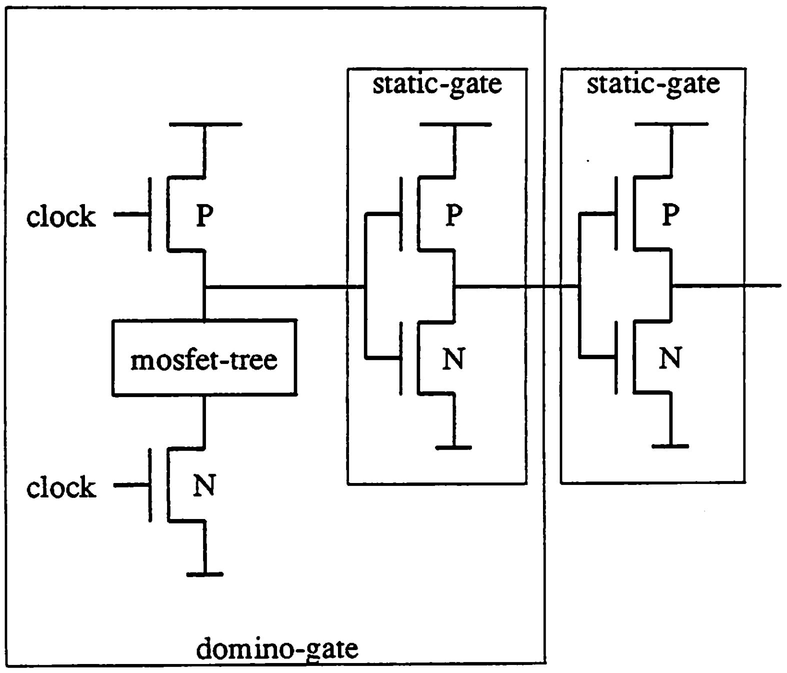


Figure 5.2: Reason for the 'in-context-of' and 'not-in-context-of' Fields

in-context-of

The value of the **in-context-of** field specifies the context in which the item to be checked must be for the error check to apply. For example, a static-gate error check may only apply when the static gate is inside a domino-gate (see Figure 5.2). The value of an **in-context-of** field is the name of a structure type. Zero or more of these fields may exist, implying that the structure may be found anywhere, or may be found in a specific number of contexts.

False errors are error reports that are not really errors. False errors are caused by not having specific enough errors or by purposely designing a section of the circuit to violate the error checks (*e.g.* conservative rules). This field, along with **not-in-context-of** and **role**, are used to make an error check more specific.

role

This is a component of **in-context-of** which specifies the particular role that the item being checked must be in for the error check to apply. For example, if there is an error

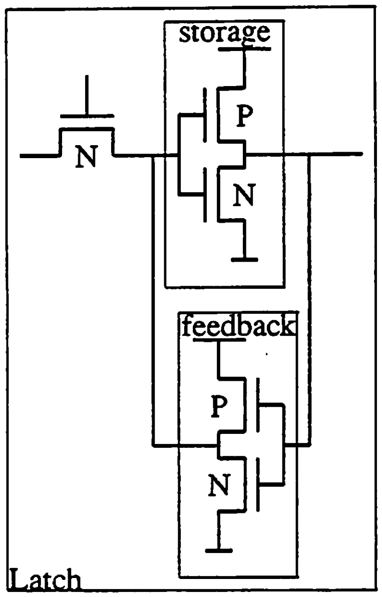


Figure 5.3: Reason for the 'role' Field

check for the weak feedback inverter in a latch, as in the case in Figure 5.3, context and structure is not sufficient. The error check must be able to disambiguate the two inverters in the structure and apply the error check to only one of the structures. The value of a **role** field is the name of the instance in the structure definition. Zero or one role fields may exist for each **in-context-of** field, implying that the structure may play either any role or one specific role.

not-in-context

The **not-in-context** field is like **in-context-of** but specifies a specific context in which the error check does not apply (see Figure 5.2). The value of a **not-in-context** field is the name of the structure type. Zero or more of these fields may exist, implying that the structure may be found anywhere, or may not be found in a specific number of contexts.

check

The value of the **check** field describes the error check independent of the primitive/structure type, and contextual and role information. The check field is usually represented as a set of constraints on the attributes of the primitive/structure being checked, the attributes of components of the primitive/structure, the values of various design style and process constants, and the environment.

An example check is (detects a charge sharing problem):

$$\frac{C_{middle-node}}{C_{middle-node} + C_{dg-internal}} < \frac{V_{th}}{V_{dd}}$$

Structure constraints

Structure constraints are conditions that depend on the values of attributes associated with the structure or primitive that is being checked.

Component constraints

Component constraints are conditions that depend on the values of attributes on the items (primitives and/or structures) that make up the structure being checked for an error. For example, to determine if a latch can change states, the error check must examine the values of the device size of the items that make up the latch.

Design Style constraints

Design style constraints are conditions that depend on the values of various design style, technology, and process constants. For example, a constraint that the number of series connected mosfets in the N (and P cores) of a NMOS or CMOS static or dynamic gate must be less than some design style supplied constant.

Environment constraints

Environment constraints are conditions that depend on the environment of the primitive/structure under check. For example, checking fanout related errors (charge sharing, drive current, *etc.*).

There are several constants, variable, subexpressions, and functions that are common to error checks across design styles and technologies. These should be provided in a library for the error check writer to help simplify the task. Examples are a function to tell whether there is a dc path to ground, a variable that represents the ground node, a function for determining if a node is a supply or clock node, or whether the terminal is a primary input or output of the circuit.

message

The value of the **message** field describes the textual information that should be displayed to the user when the primitive/structure that is in error is displayed. This can range from a simple display of the **check** field to a detailed description of the error and how to fix it. The **message** field should have all of the information available to it that the **check** field had access to. This allows the message to contain specific contextual information. For example, rather than displaying "The width of the mosfet is too small", with contextual information it could be "MOSFET 12 has a width (1.3e-6) that is smaller than the minimum allowed feature size (1.5e-6)". This allows more information to be conveyed to the user; thus following another of Heckel's principles, "Communicate in Specifics, not Generalities"[30].

- Error message without contextual information:

  ```
  This particular type of exclusive-or is not allowed in
  the design style.
  ```

- Error message with contextual information:

  ```
  The load capacitance of the dynamic-gate DG12, 3e-12
  farads, is not large enough to inhibit charge sharing
  problems. Based on the threshold voltage of 0.5 volts, the
  supply voltage of 5 volts, and the dynamic-gates internal
  capacitance of 6e-12 farads, the load capacitance must be
  at least 4e-12 farads.
  ```

One possibility for displaying error messages that was explored was to remove the **message** field and try to translate the **check** field into English, see Section 5.4.

constants

Constants used in the error check. Can be functions of other constants or process constants. For example:

$factor = 5.6$

$mobility - ratio = \frac{\mu_n}{\mu_p}$

$min - ratio = 0.85 \times mobility - ratio$

$max - ratio = 1.15 \times mobility - ratio$

comments

Comments about the error check. This information is not reported to the user, but is placed in the ruleset documentation. To allow the users the ability to use mathematical symbols and the full abilities of a text processor, the string must be in a format that LaTeX can process, for example:

```
As described in Hofmann[Hof85], to eliminate
charge-redistribution problems in domino-gates,
the following constraint must be met:
\[ \frac{C_{middle-node}}{C_{middle-node} +
C_{dg-internal}} < \frac{V_{th}}{V_{dd}} \]
```

Expands into:

As described in Hofmann[Hof85], to eliminate charge-redistribution problems in domino-gates, the following constraint must be met:

$$\frac{C_{middle-node}}{C_{middle-node} + C_{dg-internal}} < \frac{V_{th}}{V_{dd}}$$

Example Rules

Charge Sharing

Name: Charge Sharing

Structure: Domino-Gate

Check:

```
(> (/ VTH-N VDD)
   (/ (CAPACITANCE MID-NODE)
      (+ (CAPACITANCE MID-NODE)
         (INT-CAPACITANCE DYNAMIC-GATE))))
```

Message: The load capacitance of the dynamic-gate is not large enough to inhibit charge sharing problems.

Comment: As described in Hofmann[Hof85], to eliminate charge-redistribution problems in domino-gates, the following constraint must be met:

```
\[ \frac{C_{mid-node}}{C_{mid-node}
+ C_{dg-int}} < \frac{V_{th}}{V_{dd}} \]
```

Equal Rise and Fall Times

Name: Equal Rise and Fall Times

Structure: Static-Gate

Not-In-Context-Of: Domino-Gate

Check:

```
(not (<= min-ratio noverp max-ratio))
```

Constants:

```
min-ratio = (* 0.85 mobility-ratio)
max-ratio = (* 1.15 mobility-ratio)
```

Message: The N and P device W/L's of this static-gate are not ratioed such that the rise and fall times will be approximately equal.

Comments: The rise and fall times of static-gates should be approximately equal. One exception is the static-gate that plays the role of output buffer in a domino-gate; in this case the gate only has to actively pull in one direction.

Weak Latch

Name:	Weak Latch
Structure:	Latch

Check:	`(< (* factor (w-over-l (pull-down fb-inv)))`
	`   (w-over-l load-gate))`

Constants:	`factor = 5.6`
Message:	The load gate is not big enough to cause a state change in the latch.

No DC Path To Ground

Name:	No DC Path To Ground
Structure:	Node
Check:	`(not (path-between self *dc-nodes*))`
Message:	The node has no dc path to ground.

Bad N and P Core Connections

Name:	Bad N and P Core Connections
Structure:	Static-Gate

Check:	`(not (logic-equal (logic-function ncore)`
	`          (logic-dualize`
	`              (logic-function pcore))))`

Message:	The logic functions of the N and P cores are not logical duals.
Comments:	If the logic cores are not logical duals, there will be certain input values that will cause both cores to turn on (shorting the supplies), or cause both cores to turn off (causing the output to float).

5.3 Finding Errors

Depending on what is allowed in error check specifications and whether there is a structure finding phase, finding errors may require pattern matching. In RUBICC the same rule language used for finding structures is used in the error check and thus the same pattern matching operations can be used (in practice the amount of actual pattern matching was limited since the structure finding phase found most of the structures of interest). In QCRITIC, since there is no structure finding phase the error checks must describe how to find the elements (and groups of elements) that are to be checked.

In the work in this dissertation, the removal of pattern matching from the error checks was explored. The removal of pattern matching makes the error check description simpler and makes finding errors simpler and quicker. However, this pushes the finding of groups of items that are grouped exclusively for error checking into the structure finding phase. This means that some structures are defined that have no logical function.

By getting rid of pattern matching, the error checks are limited to the checking of contextual information (`in-context-of`, `not-in-context-of`, and `role`), and the evaluation of expressions (the `check` field). The basic algorithm for locating errors is as follows: each item (either a primitive or a structure) is looked at and all error checks that apply to the item are checked, excluding the error checks that are in the 'do-not-check' list. The 'do-not-check' list contains the names of errors that should not be checked for the particular element type. This information is gained from annotation placed on the circuit by the designer. The complexity of the error check phase is $O(\sum(N_i \times C_i))$, where N_i is the number of elements of type i in the database, and C_i is the number of error checks defined for type i.

```
foreach circuit element type (T) instantiated in the circuit {
  foreach error check of T (E) not in the 'do not check' list {
    foreach instance of T (I) {
      if (E is not a member of the 'do not check' list of I) {
        if (check(E,I) is true) {
          mark I as violating E
        }
      }
    }
  }
}
```

5.4 Displaying Errors

When presenting an error, a description of the error and the objects in error should be presented to the user. One way to present the information is to graphically zoom to the items in error, highlight them, and display the error check. Highlighting gives contextual information about the error and the display of the error check itself gives a description of the error. Structures that are in error don't exist as a graphical entity in the database, so the collection of primitive elements that make up the structure should be highlighted.

In some cases the **message** field would not be sufficient to describe the error (or the **message** field might be missing and one must be synthesized from the **check** field). Most implements of error checks would use LISP as the language for describing the constraints. Since many users would not be fluent in LISP, the **check** should be translated into a form more easily understandable.

It is straightforward to translate LISP of the complexity generally used in the **check** field into simple English. The basic algorithm is to match templates to the various lisp forms. Some templates cover a single minimal s-expression (*e.g.* `(not bob)`), and some handle complex s-expressions (*e.g.* `(and s-expression1 ... s-expressionN)`). The templates can be generic, such as ones for handling the basic lisp functions, or specific to the task of critiquing. The specific ones have information about the functions and variables used in error checks, *e.g.* `(length *dc-nodes*)`. The types of templates used are: critic variables/constants, lisp atoms/numbers, quoted expressions, two and three argument expressions, if-then-else expressions, specific LISP functions (*e.g.* `any`, `every`), functions specific to the application (*i.e.* `count-types`, `(not (path-between ... ...))`, `(not (member-of ..))`), and fields in primitive and structure definitions.

The following are some examples of translating s-expressions to English:

LISP Code:

```
(and (not (member-of (parent gate)))
     (equal (direction gate) sympol::input)
     (= (count-types (node gate) t t) (length *dc-nodes*))))
```

English Translation:

```
     The element containing the terminal GATE is not contained
in a structure and the DIRECTION of GATE is equal to the
direction INPUT and the number of elements of any type connected
to the node connected to the GATE terminal is equal to the
number of elements in the list of nodes connected to a supply
```

or clock

LISP Code:

```
(< (dc-max-fanout self)
   (sum-of 'dc-fanin (free-terminals self))))
```

English Translation:

The DC-MAX-FANOUT of SELF is less than the sum of DC-FANIN over all the terminals of SELF connected to elements not contained in structures

Chapter 6

Critic

6.1 Overview

To experiment with the ideas explored in Chapters 3, 4 and 5, the program **Critic** [64, 67] was developed as a test-bed. **Critic** is a knowledge-based system[1] for critiquing circuit designs that is integrated into a design environment under development at the University of California at Berkeley[28]. Aside from experimenting with the ideas in the previous Chapters, the main goals of **Critic** were: tight integration with a design system, interactive graphical input and feedback, ease of use, and separation of the knowledge base from the program internals.

There are five phases in the execution of **Critic**: load the knowledge base, load the circuit, find structures, find errors, and examine the structures and errors (see Figure 6.1).

The following sections will present the major sections in **Critic** and describe how they implement the ideas and algorithms presented in the previous chapters.

Loading the Knowledge Base

The knowledge base consists of: process and design style constants (such as threshold voltage and the maximum number of series N-type MOSFETs allowed in series), a set of primitive descriptions and the error checks that apply to them, a set of structure

[1]Knowledge-based systems[29] are programs that rely on large amounts of declarative information, *knowledge*, to perform tasks.

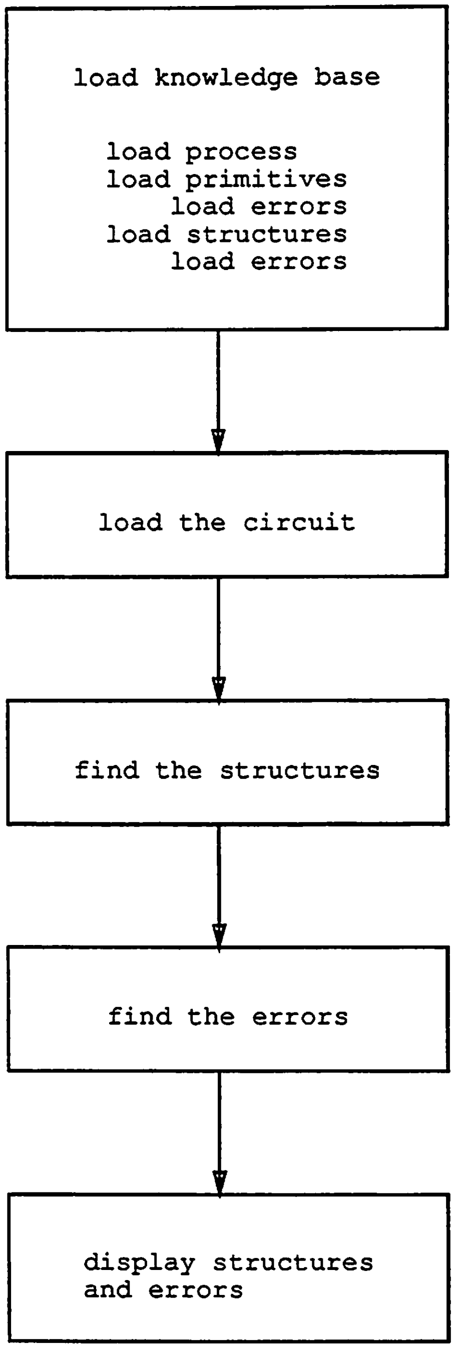


Figure 6.1: Flow of Critic

descriptions (collections of primitives and other structures) and the error checks that apply to them, and error checks that apply to nodes.

A goal of the work on **Critic** was to remove all implementation specific information from the knowledge base. This was partially successful, most of the implementation specific information present in other critics has been removed, however there are two places where the implementation shows up: computed attributes and error checks. The value of a computed attribute and the constraints used in an error check are expressions in the implementation language.

Loading the Circuit Description

Once the knowledge base has been loaded and the information about how to read the data base has been created the circuit can be loaded. Each instance read from the OCT data base is represented internally as an instance of the class of its corresponding primitive. Meta classes are predefined for all primitive classes that deal with keeping lists of primitives of each type and building the inverted netlist representation. Next the information about errors that should not be checked is processed.

Finding Structures and Errors

The process of finding structures and errors follows the algorithms described in Chapters 4 and 5.

Integration into a CAD System

The present version of **Critic** is tightly coupled with the Berkeley OCT CAD design data manager and the VEM user interface using a remote procedure call package[28, 63]. **Critic** has gone through several changes as more was learned about how the program should operate and how it should interact with the user and the environment. The language chosen to implement **Critic** was LISP. There were a number of reasons for this decision: (1) the current trend to use LISP for CAD tools[51, 47, 38], and (2) the majority of rule-based and object-oriented languages available at the time were embedded in LISP[57, 6, 43].

The initial version of **Critic** was not tightly integrated with a design system, but was a stand-alone tool running on a XEROX Dandelion; it was implemented in GLISP[2] and INTERLISP[71]. It took textual input for the circuit description (in SIM[26] format) and the knowledge base.

```
; sim format
; type nodelist ... attributes...
;
; type: n (n-type mosfet), p (p-type mosfet), C (capacitor)
;
; inverter
p in out vdd 4 16
n in out gnd 4 2
```

The textual format of the knowledge base is described in chapters 3, 4, and 5. It had a custom rule system and produced textual descriptions of the structures and errors found in the circuit, for example:

```
``LATCH L12'' is composed of:
    INVERTER 5
    INVERTER 9
    MOSFET 3

Error ``width too small'' was violated by ``MOSFET M231''
```

The next version used OPS5[3] for its rule system but still ran as a stand-alone tool. The textual input was translated into OPS5 commands. The choice of OPS5 was driven by many concerns: (1) the flow of OPS5 (forward chaining pattern matching rule-based system fit the basic requirements of what a critic needed), (2) many other CAD tools were being developed using OPS5[39, 4, 34, 37], and (3) the author was experienced with the usage of the system.

To fit **Critic** into the Berkeley design system, the textual input was replaced with input from the OCT data manager and the program moved to UNIX. By using OCT for circuit and knowledge-base storage **Critic** was then able to directly operate on circuits designed

[2]GLISP[57] is an extension to LISP to allow simple object oriented features, and pseudo-English descriptions of algorithms.

[3]OPS5[19] is a system for building forward chaining (data driven) production systems.

using the Berkeley Design Environment, the knowledge-base could be entered, browsed, and modified using the standard editors, and annotation of the circuit with the structures and errors found during the run became possible. This version of **Critic** was written in GLISP and Franz Lisp.

Next, VEM was modified to accept simple commands from remote applications using UNIX *pipes*. Pipes allow communication between UNIX processes. VEM would send a token that directed the **Critic** to find the structures and errors, identify the next structure, or identify the next error. **Critic** was modified to take advantage of this simple interprocess communication facility. VEM would take the information from the structure/error identification and highlight the objects on the screen. This allowed a simple interface for sequencing through structures and errors found in the design. With the use of pipes, VEM became the primary user interface for **Critic**. However, VEM and **Critic** did not share the same in-memory copy of the OCT data, so either program could modify the data without the other being aware. In order for VEM to display the modifications that **Critic** made and for **Critic** to recheck circuits that had been modified with VEM, the two programs needed to share the same copy of the OCT data. In addition, VEM had several user interface features that could be used by **Critic** to improve its interface. They included dialog boxes for displaying results and requesting user input, selection sets for highlighting regions of the circuit and user input, and menus for **Critic** commands. The need for shared OCT data and the use of the user interface features could not easily be accomplished via the simple pipe mechanism, so a full remote procedure call interface (RPC) was added to VEM and RPC client libraries in C and VAXLISP were written. This allowed **Critic** to be written as if it were a program residing in the same address space as VEM with full access to the OCT data base it was displaying along with all of the VEM user interface facilities.

The current version of **Critic** is written in CLOS[6][4] and VAXLISP[5]. There are 8000 lines of CLOS/VAXLISP with 1000 lines of C[36]. The RPC package written to support **Critic** consists of 4000 lines of C code on the client (**Critic**) side and 5000 lines of C code on the server (VEM) side.

[4]An object-oriented system embedded on top of Common Lisp[69].

[5]Digital Equipment Corporation's version of Common Lisp.

Berkeley CAD Framework

The Berkeley CAD Framework[28] is a framework for CAD tools. The framework provides data representation and storage, user interface, and tool control. The representation and storage portion are provided by OCT, the user interface by VEM, and tool control by RPC.

The OCT data manager

OCT provides the data access and representation facilities for the Framework. OCT is designed to be extensible: it makes few assumptions about the data to be stored. Instead it implements a few basic primitives, a general mechanism for relating primitives to one another, and a small set of generic operations on these primitives. The set of operations insulates the user from the internals of OCT or changes in the data structures and algorithms, and protects the data from accidental corruption by the user. Higher-level layers can implement a particular design style on top of the OCT interface. This allows experimentation with differing representations without requiring modifications to the underlying data base.

The highest level in OCT is the *cell*, which is any portion of the chip that the designer wishes to consider as a single unit. Thus a cell may be a transistor, a contact, a gate, a data path, an entire chip, or even a board with several chips. Each cell can have many *views*, such as a schematic view of the cell, a mask geometry view, or a simulation view. OCT does not address the issues of what views a cell may have, nor what is contained in a view, nor how the views are related. These decisions are left as a design policy decision made by the design team. Views are hierarchical: they contain instances of other views which in turn may contain instances of other cells, *etc.* For various applications, it is advantageous to cut off this hierarchy; instead of continuing to traverse the hierarchy by processing the contents of a view, the view is represented by some simplified abstraction. For a graphics editor, the abstraction might be a bounding box; for routing the abstraction might be the terminals and routing regions of the view; for a schematic view it might be a symbol, and for a design rule-checker it might be just that geometry on the boundary of the view that needs to be checked against neighboring views (thus avoiding rechecking the interior of a view each time it is instantiated). These abstractions are called *facets* of the view. Each view has a facet named *contents* which contains the actual definition of the view, and zero or more interface facets. OCT does not define what interface facets may exist, nor what their relation to the contents facet might be.

The only relation that OCT does enforce is that all interface facets inherit the formal (external) terminals of the contents facet. The facet is the fundamental unit that is edited in OCT. A particular facet of a view of a cell is opened and edited independent of the other facets of that view and the other views of that cell.

OCT has just a few basic objects. There are the standard geometrical objects: boxes, polygons, labels, circles, wires, and layers. Interconnection and hierarchy are represented using terminals, nets, instances, and facets. Free-form annotation is available using properties: which are named objects that take on a variety of values (numeric, string, array, another object) and that can be attached to any OCT object. The general mechanism for relating objects in OCT is *attachment*, which may be thought of as a directed link between two objects. An object may be attached to an arbitrary number of objects and may in turn have an arbitrary number of objects attached to it. For example, in our design policy, geometry is shown to be on a particular layer by attaching it to that layer; a terminal is connected to a net by attaching it to that net. The *bag* object was added to OCT to act as a named attachment point for related objects, such as a collection of permutable pins, a group of objects in the same grid line in compaction, or one of the selected sets used in VEM.

Figure 6.4 shows a fragment of C code that netlists an OCT facet. Two data types are used in the example, the `octGenerator` and the `octObject`. The `octGenerator` is used by the `octGenerate` function to traverse the attachments of an OCT object. The `octObject` is the generic data type used to represent all the OCT objects. The OCT objects `OCT_FACET`, `OCT_INSTANCE`, `OCT_TERM`, and `OCT_NET` are used in the example. The first step in an OCT program (and in this example) is to open the facet to be browsed or edited. This is accomplished with a call to `octOpenFacet`. `octOpenFacet` maps the facet name to a name in the persistent storage system (currently the UNIX file system) and then reads the facet into memory. There is no direct programmatic access to this data; all access is via OCT functions that return copies of the actual OCT data. After the facet has been read into memory, the instances are processed one by one. A generator is created

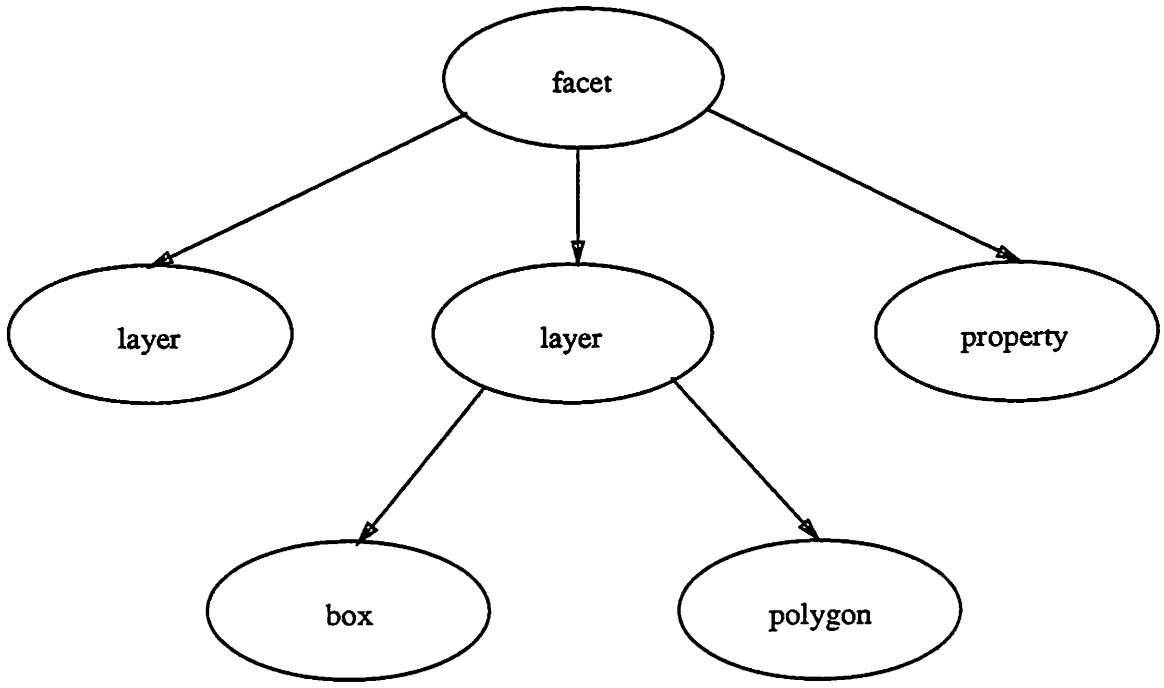


Figure 6.2: Example Attachments

using `octInitGenContents`. Each call to `octGenerate` returns an OCT instance object. `octGenerate` returns `OCT_OK` as long as there are objects to return. When the generation sequence is finished, `OCT_GEN_DONE` is returned. The terminals associated with the instance are attached to the instance, so another generator is started to generate through the terminals. Nets contain the terminals that are logically connected, so the call to `octGenFirstContainer` is used to get the first (and only) net that contains the terminal.

Policies

OCT provides a *mechanism* for representing CAD data but places no meaning of the data. There is no explicit support in OCT for representing various abstract levels of design, such as schematic and logic. *Policy* is used for assigning meaning to the data represented using OCT. For example, OCT has objects that represent layers and geometry, but does not specify how to relate them to give the *meaning* of a geometry implemented on a given layer. The *policy* states that a geometry that is contained by a layer is implemented on

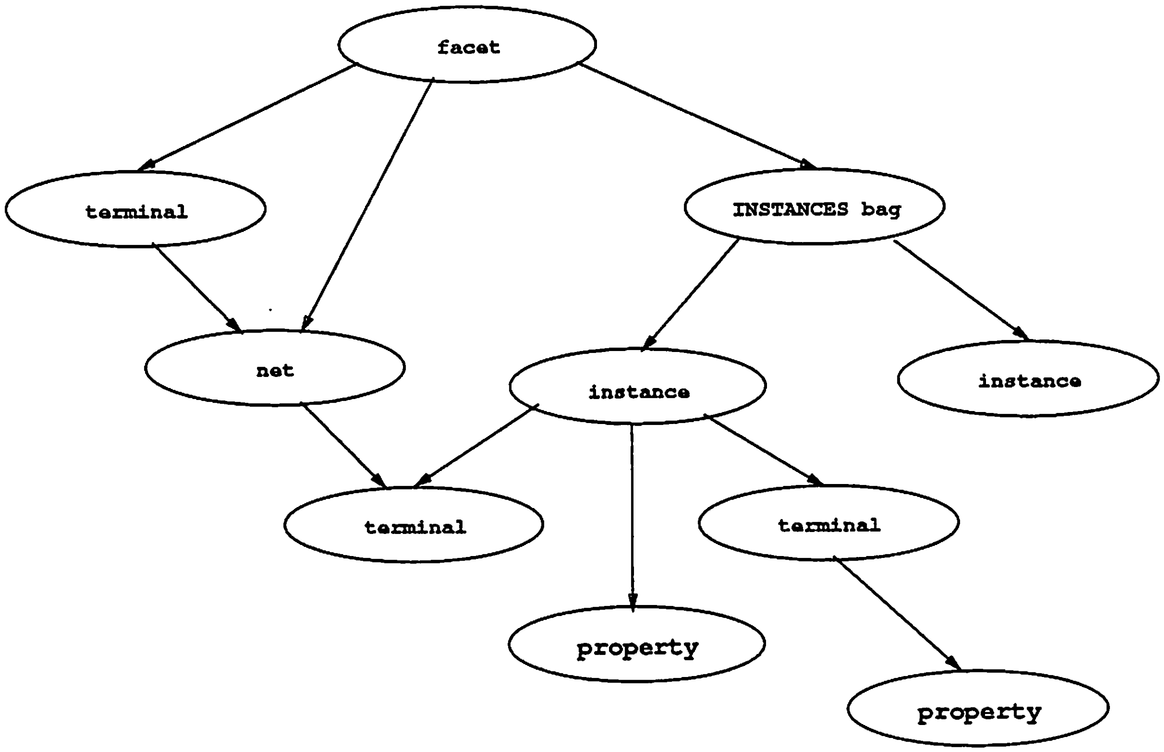


Figure 6.3: Example Netlist Attachments

```
octObject facet, instance, terminal, net;
octGenerator igen, tgen;

/* read a facet into memory */
octOpenFacet(&facet);

/* process each instance */
octInitGenContents(&facet, OCT_INSTANCE_MASK, &igen);

/* generate and process the instances */
while (octGenerate(&igen, &instance) == OCT_OK) {

    /* process each terminal on the instance */
    octInitGenContents(&instance, OCT_INSTANCE_MASK, &tgen);

    /* generate and process the terminals */
    while (octGenerate(&tgen, &terminal) == OCT_OK) {

        /* get the net associated with the terminal */
        octGenFirstContainer(&terminal, OCT_NET_MASK, &net);
    }
}
```

Figure 6.4: Sample OCT Code Fragment - Netlist a Design

that layer. As another example, OCT has objects that represent nets and terminals, but does not specify how connectivity is represented. The *policy* describes how terminals and nets are used to represent connectivity. Policy also describes what annotations are required: to assign a logic function to a cell, the logic function is placed in a string valued property (named `LOGICFUNCTION`) attached to the formal terminal representing the output of the element (the output terminal is in turn denoted by attaching the string-valued property `DIRECTION` to the formal terminal and giving it the value `OUTPUT`).

The following gives an overview of the policies in the Berkeley Design Environment that are used by **Critic**.

generic: The *generic* policy defines the requirements and restrictions that are the same across all other policies. Each view of a cell has two facets: a **contents** facet and an **interface** facet. The **contents** facet contains the *definition* of the view. The definition contains the objects that comprise the view in detail (actual geometries and hierarchy). The **interface** facet contains an *abstraction* of the view. The abstraction is intended to contain the minimal information needed to describe the cell to higher-level cells and tools (*e.g.* terminals, routing regions, annotation). The abstraction should be derivable from the definition.

physical: The *physical* policy defines the requirements and restrictions for representing mask-level designs (see Figure 6.6). All OCT objects with the exception of nets are allowed. Hierarchy is rarely used at this level, with the cells composed primarily of geometric objects, with terminals, and annotation.

symbolic: The *symbolic* policy defines a level slightly higher than mask level where connectivity is explicitly represented and annotation is used to represent *extracted* information (widths of MOSFETs, values of resistors, what terminal of a MOSFET is the gate, *etc.*). All non-geometric objects are allowed. The only geometric object allowed is the path and it is restricted to two points. Most tools in the Berkeley Design Environment work with designs at this level. An important part of the symbolic policy is inheritance. The policy defines various properties that can be attached to facets (masters) and formal terminals; if one of those properties is attached to the instance or the actual terminals of the instance they override the values in the master.

schematic: The schematic representation is similar to symbolic, except that the physical interconnect information is simplified; all interconnect is on a single layer and zero width. All subcells are drawn as abstract symbols rather than detailed physical or symbolic cells (see Figure 6.8).

Critic uses the symbolic policy to analyze designs. The only information **Critic** uses are the netlist and some additional annotation. The netlist is obtained from the contents facet of the design being analyzed. The subcells in the design can be physical, symbolic or schematic. **Critic** does not go down the hierarchy: **Critic** works at a single level in the design. Information about the subcells is found in the interface facet for the subcell. The only information gathered from the interface facets are terminal information (number, names, types, direction), special annotation (properties attached to the facet, such as the width of a MOSFET, or the value of the capacitance). The information that is in the interface is determined by the physical and symbolic polices.

The VEM Graphics Editor

VEM[27] is an interactive graphics environment for viewing and editing OCT data. VEM also provides a facility for invoking various CAD tools on OCT views. VEM uses the *X-Window System*[22] as its primary graphics interface. VEM allows users to open any number of possibly overlapping windows, each showing one OCT facet. More than one window may have the same associated OCT facet, in which case a change in one of the windows causes updates in all the other windows.

VEM supports three editing styles, with each style corresponding to a particular OCT policy:

physical: used for editing *physical* (or *mask-level*) design. VEM provides basic operations for the entry and manipulation of geometric objects (boxes, polygons, *etc.*) and for the creation of terminals (connection points) (see Figure 6.6).

symbolic: used for the entry and manipulation of instances of physical cells or other symbolic cells and for the interconnection of these instances. The symbolic editing mode

```
octObject facet, bag, instance, terminal, net;
octGenerator igen, tgen;

/* read a facet into memory */
octOpenFacet(&facet);

/* process each instance attached to the INSTANCES bag */
bag.type = OCT_BAG; bag.contents.bag.name = "INSTANCES";
octGetByName(&facet, &bag);

/* process each instance */
octInitGenContents(&bag, OCT_INSTANCE_MASK, &igen);

/* generate and process the instances */
while (octGenerate(&igen, &instance) == OCT_OK) {

    /*
     * process each property on the instances master
     * and see if it is overridden on the instance
     * (e.g. MOSFETLENGTH, CAPACITANCE)
     */

    /* process each terminal on the instance */
    octInitGenContents(&instance, OCT_INSTANCE_MASK, &tgen);

    /* generate and process the terminals */
    while (octGenerate(&tgen, &terminal) == OCT_OK) {

        /*
         * process each property on the formal terminal
         * that the actual terminal represents and see if
         * it is overridden on the actual terminal
         * (e.g. TERMTYPE, DIRECTION)
         */

        /* get the net associated with the terminal */
        octGenFirstContainer(&terminal, OCT_NET_MASK, &net);
    }
}
```

Figure 6.5: Netlisting an OCT Symbolic Representation

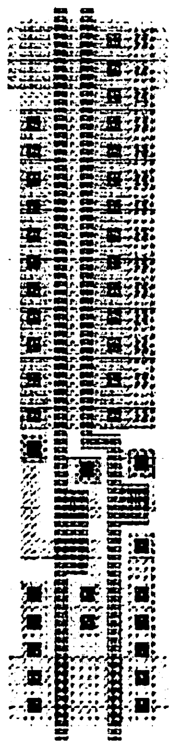


Figure 6.6: Sample OCT Physical Design

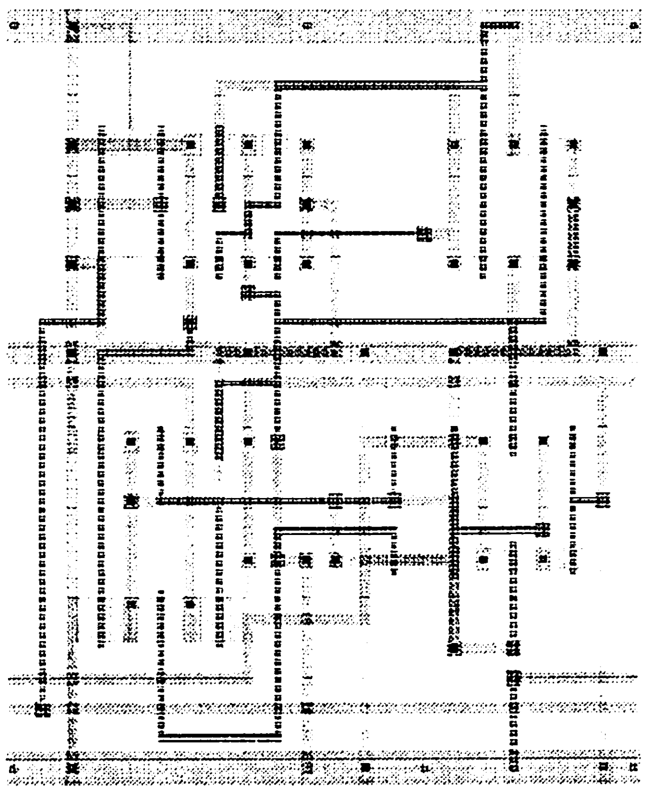


Figure 6.7: Sample OCT Symbolic Design

is responsible for keeping interconnectivity information in the data base up to date (see Figure 6.7).

schematic: is an extension of symbolic where the cells are represented by symbols (see Figure 6.8), wire width is insignificant, and design rule correctness is not an issue. Netlist entry is performed using the schematic editing primitives.

Besides allowing the user to enter and browse circuits and knowledge bases using VEM, **Critic** makes use of several features exported from VEM to RPC applications; these include the ability to collect textual (strings) and graphical (points, boxes, OCT objects) arguments, creation and browsing of annotation (extra information to **Critic** and results of **Critic** runs), highlighting of objects on the screen, menus and key bindings for invoking **Critic** commands (*e.g.* show the next error), and dialog boxes for prompting for extra information.

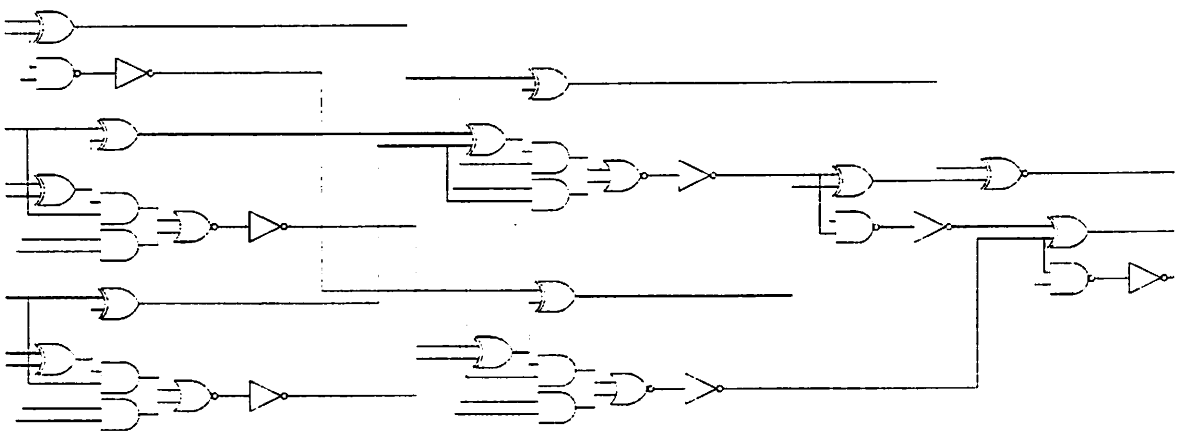


Figure 6.8: Sample OCT Schematic Design

The Remote Procedure Call Package

A goal of the VEM and OCT design system was to take advantage of our distributed environment, one with multiple machines (of different types and operating systems) and multiple languages. OCT works in this distributed environment by using a remote file system to access the data base, thus allowing the data base to be spread out over many machines. VEM works by using a network-transparent windowing system which allows the I/O devices (display, keyboard, mouse) and the program controlling them to be on different machines. VEM and user applications communicate in this environment by using remote procedure calls. Figure 6.9 shows how the system components communicate. The Remote Procedure Call (RPC) package allows user applications to run as separate processes outside of the VEM address space. The applications make subroutine calls to VEM and OCT as if they were in the same process and address space, similar to tightly-bound VEM commands. The RPC client, that is the RPC code linked with the application program, interprets these calls and passes them to VEM. The RPC server linked with VEM calls the appropriate VEM and OCT routines, and returns the results to the RPC client.

See [28, 62] for a detailed description of RPC and a user's manual.

Critic Run-Time Environment

Critic runs as a remote application from the VEM graphics editor, and accesses

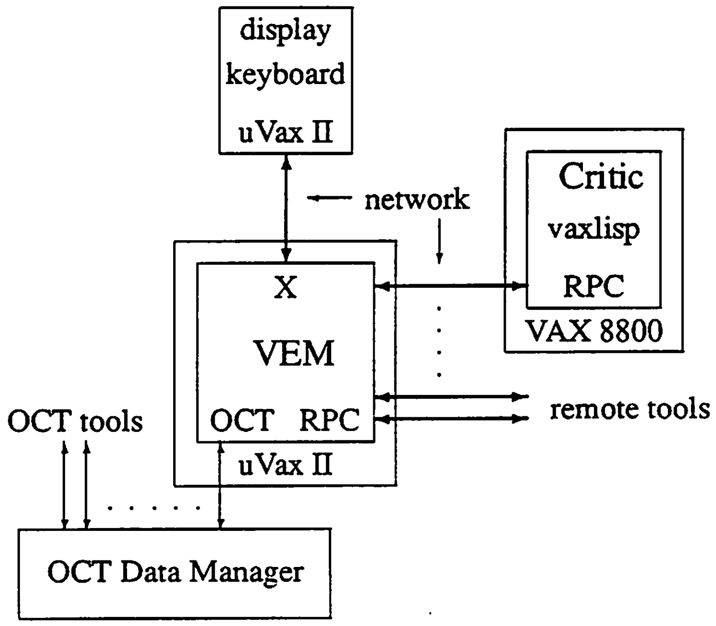


Figure 6.9: **Critic** and the Berkeley Design Environment

the circuit and knowledge-base using the OCT data manager (see Figure 6.11). The user opens one or more VEM windows displaying the circuit to be checked, then requests that VEM start execution **Critic**. VEM is usually run on a workstation and **Critic** is spawned on a mainframe, with all communication handled using remote procedure calls across the network. If a knowledge-base has not been preloaded, a dialog box pops up requesting the name of the knowledge base to be used. At any time before **Critic** starts finding the structures and errors, the user can modify the circuit using any of the standard VEM editing commands. The user can mark regions of the circuit that need special attention: areas where errors are known to occur but the designer has decided that they should be ignored, or to get around a faulty error check. The user is prompted for the types of primitives and structures to be marked and the errors that should be ignored. This information is placed in the OCT data base for use in the current and future (if saved) **Critic** runs. Then **Critic** starts execution and finds the structures and errors in the circuit that correspond to the descriptions in the knowledge-base. As structures and errors are found, this information is placed in the OCT data base for later sequencing and display. **Critic** can use this annotation on a future run to start from where it left off. A dialog box keeps the user informed of the progress of

```
/* define the user-level RPC application commands */
RPCFunction Cmds[] = {{show, "Finder", "show", "S", 0}};

/* initialization routine */
UserMain(display, spot, cmdList, userOptionWord, array)
char *display;
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
RPCFunction **array;
{
    vemMessage("Finder operational", MSG_DISP);
    *array = Cmds;
    return sizeof(Cmds) / sizeof(RPCFunction);
}

/* RPC command - textually display objs in a selected set */
show(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    RPCArg *firstArg;
    octObject theBag, obj;
    octGenerator theGen;

    if (lsLength(cmdList) != 1) {
        vemMessage("format:  objects(n) : show", MSG_DISP);
        return RPC_OK;
    }
    lsFirstItem(cmdList, (lsGeneric *) &firstArg, 0);
    if (firstArg->argType != VEM_OBJ_ARG) {
        vemMessage("format:  objects(n) : show", MSG_DISP);
        return RPC_OK;
    }
    theBag.objectId = firstArg->argData.objArg.theBag;
    octGetById(&theBag);
    octInitGenContents(&theBag, OCT_ALL_MASK, &theGen);
    while (octGenerate(&theGen, &obj) == OCT_OK) disp_obj(&obj);
    return RPC_OK;
}
```

Figure 6.10: Sample RPC Code Fragment

Critic (see Figure 6.11).

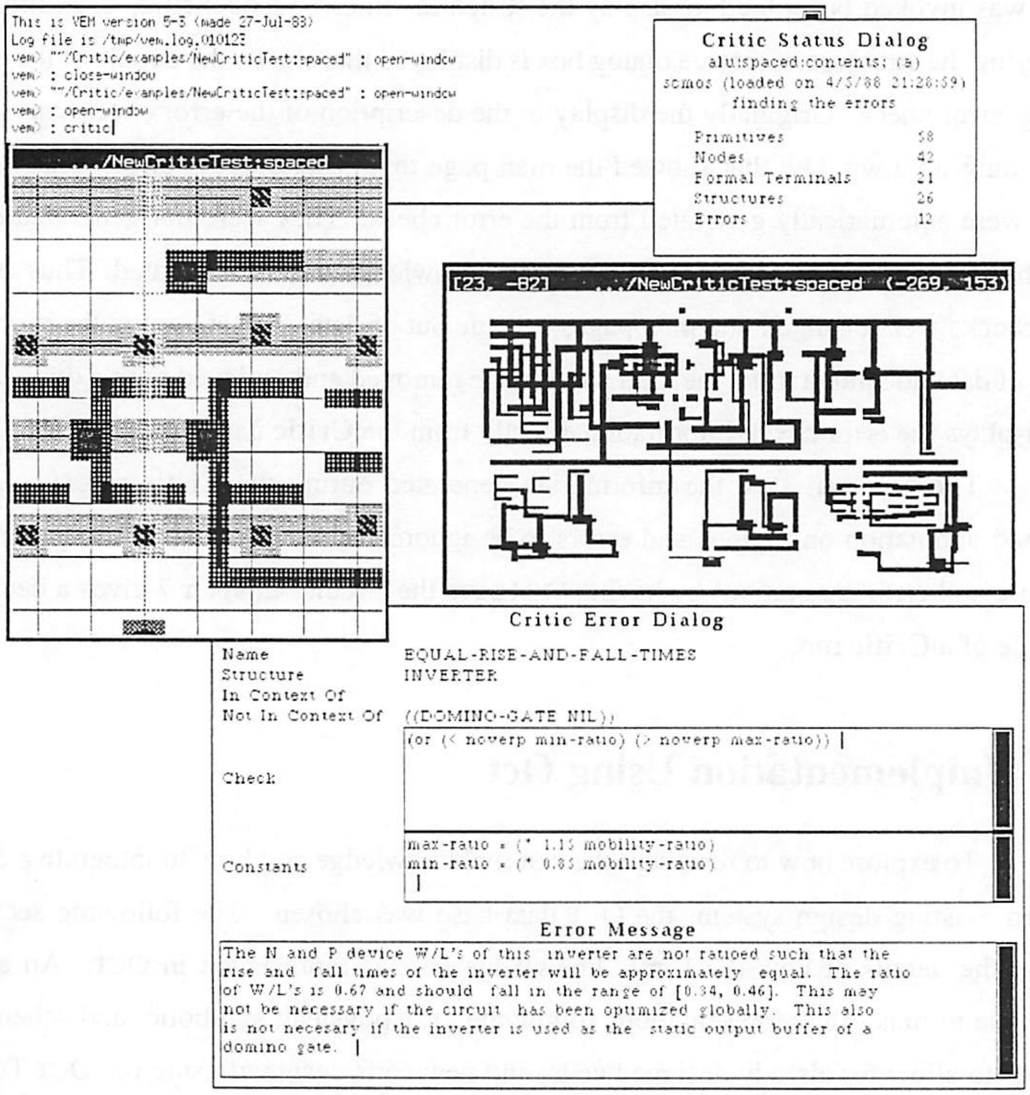
Once **Critic** has finished finding the structures and errors, the user can use **Critic** commands to sequence through the structures and errors. Normally the user would have two windows displaying the circuit being checked: one context window displaying the entire circuit, and the other used for displaying the structures found and the items in error. When sequencing through the structures, the components that make up the structure are highlighted in all windows displaying the circuit. The window in which the sequence command was invoked is zoomed to display the structure. In error sequencing, in addition to displaying the structure in error, a dialog box is displayed that shows the details of the error and the error check. Originally the display of the description of the error check was done by opening up a window that showed the man page that described the error. These man pages were automatically generated from the error check. They were not generated each time the error was displayed, but at the time the knowledge base was created. Thus as the error checks were changed, the man pages became out of date. To get around the problem of out of date documentation, the man pages were removed and replaced with a dialog box that displays the error check information directly from the **Critic** data structures.

The user can save the information generated during the **Critic** run (*i.e.* user-specified annotation on regions and errors to be ignored by **Critic**, and **Critic**-generated structure and error annotation) by having VEM save the circuit. Chapter 7 gives a detailed example of a **Critic** run.

6.2 Implementation Using Oct

To explore how to represent the external knowledge and how to integrate a critic with an existing design system, the OCT data base was chosen. The following sections present the details of how the **Critic** knowledge base is represented in OCT. An effort was made to make the representations follow the OCT physical, symbolic, and schematic policies to allow for already designed cells, and new cells designed using the OCT TOOLS system[6], to be used with zero or minimal additions. The basics of OCT and OCT policy have

[6]The OCT TOOLS system[62] is a set of CAD tools integrated around the OCT data base and the VEM graphics editor.



```
This is VEM version 5-3 (made 27-Jul-88)
Log file is /tmp/vem.log.010123
vem> ""/Critic/examples/NewCriticTest:spaced" : open-window
vem> : close-window
vem> ""/Critic/examples/NewCriticTest:spaced" : open-window
vem> : open-window
vem> : critic
```

```
Critic Status Dialog
alu:spaced:contents; (a)
somos (loaded on 4/5/88 21:26:59)
finding the errors

Primitives            58
Nodes                 42
Formal Terminals      21
Structures            26
Errors                12
```

```
Critic Error Dialog
Name                 EQUAL-RISE-AND-FALL-TIMES
Structure            INVERTER
In Context Of
Not In Context Of    ((DOMINO-GATE NIL))
                     (or (< noverp min-ratio) (> noverp max-ratio))
Check

                     max-ratio = (* 1.15 mobility-ratio)
Constants            min-ratio = (* 0.85 mobility-ratio)

Error Message
The N and P device W/L's of this inverter are not ratioed such that the
rise and fall times of the inverter will be approximately equal. The ratio
of W/L's is 0.67 and should fall in the range of [0.34, 0.46]. This may
not be necessary if the circuit has been optimized globally. This also
is not necesary if the inverter is used as the static output buffer of a
domino gate.
```

Figure 6.11: **Critic** Screen Layout

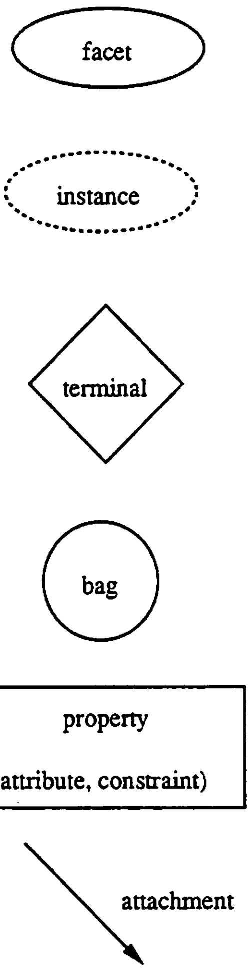


Figure 6.12: Symbols Used for OCT

already been presented in a previous section. Figure 6.12 is a description of the symbols used in the following sections for describing the OCT representation.

Knowledge Base

The knowledge base is represented as an OCT facet with instances for each of the four major parts: *process/design style*, *primitives*, *node*, and *structures* (see Figure 6.13). Each instance references the facet that contains the description of the individual part.

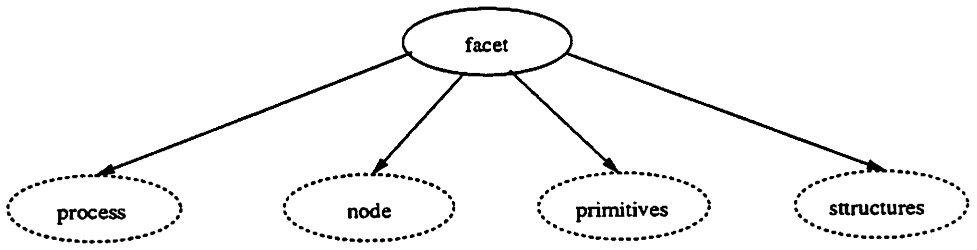


Figure 6.13: Knowledge Base Organization in OCT

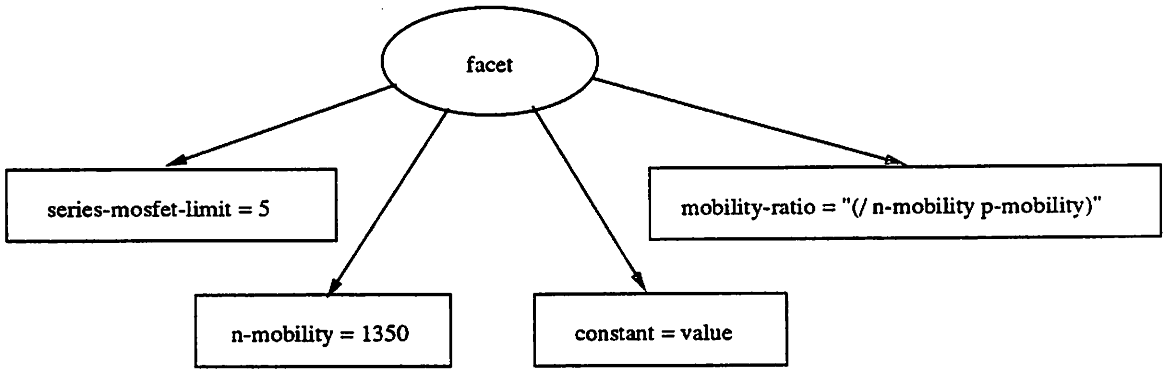


Figure 6.14: Process Definition in OCT

Process/Design Style Description

The *process* facet contains a set of OCT properties that define the constants of interest for the technology and design style being checked (see Figure 6.14). Examples of process constants are threshold voltage and mobility. Design style constants could be the maximum number of series connected MOSFETs in an NMOS static gate, and the maximum fanout for a gate array family.

Node Description

The *node* facet contains descriptions of the error checks to apply to nodes and the when the checks should not be applied based on the type of the node (*e.g.* supply, clock, connected to a formal terminal). See Figure 6.15.

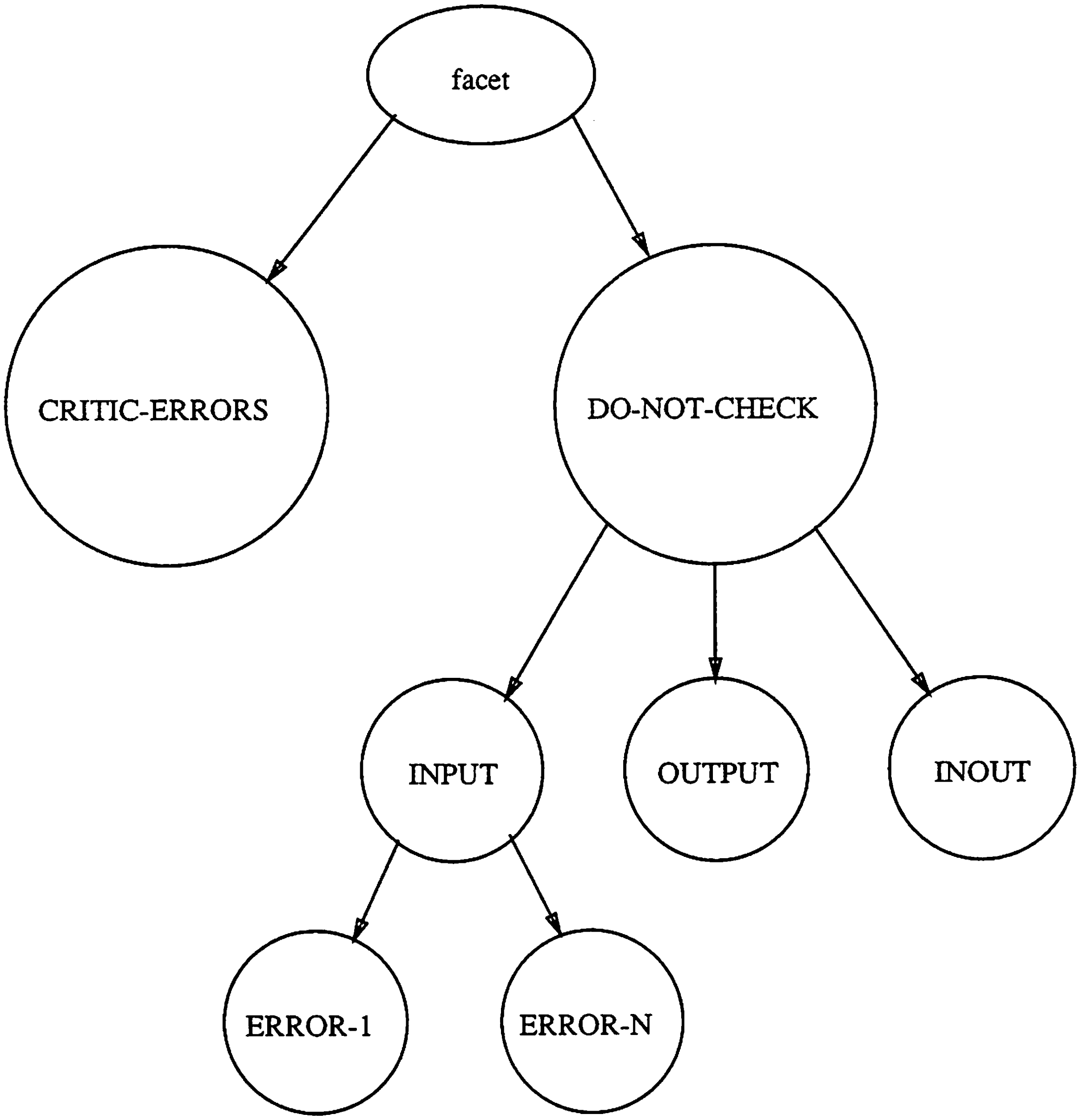


Figure 6.15: Node Definition in OCT

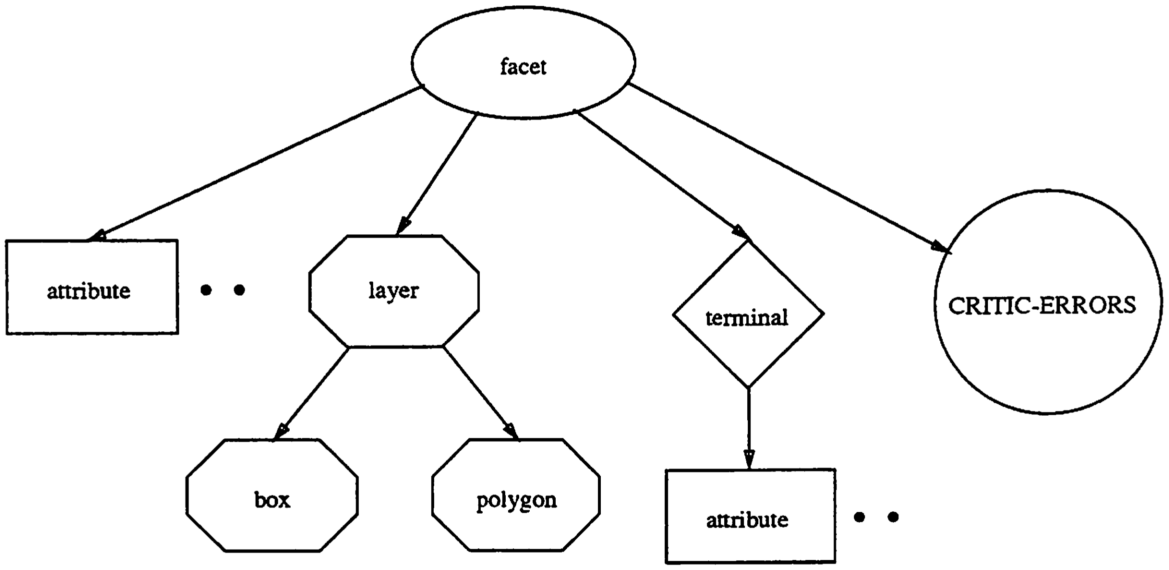


Figure 6.16: Primitive Definition in OCT

Primitive Descriptions

The *primitive* facet contains one instance for each primitive in the knowledge base. The name of each instance is the name of the primitive described by the master of the instance. Each primitive description is represented by an OCT facet (see Figure 6.16). The facet contains an OCT physical policy correct representation of the primitive. This includes the terminals, basic annotation, and optionally, a pictorial representation of the primitive. Additional attributes not defined in the OCT policy can also be added (*i.e.* all calculated attributes). The facet also contains a description of each error that can be applied to instances of the primitive in the circuit, see the section on error descriptions that follows for information on the representation of the errors in OCT.

Structure Descriptions

The *structure* facet contains one instance for each structure in the knowledge base. The name of each instance is the name of the structure described by the master of the instance. Each structure description is represented by an OCT facet (see Figure 6.17) containing the description of the structure (layout, connectivity, terminals, and annotation), and descriptions of the errors associated with the structure being defined. The facet representing

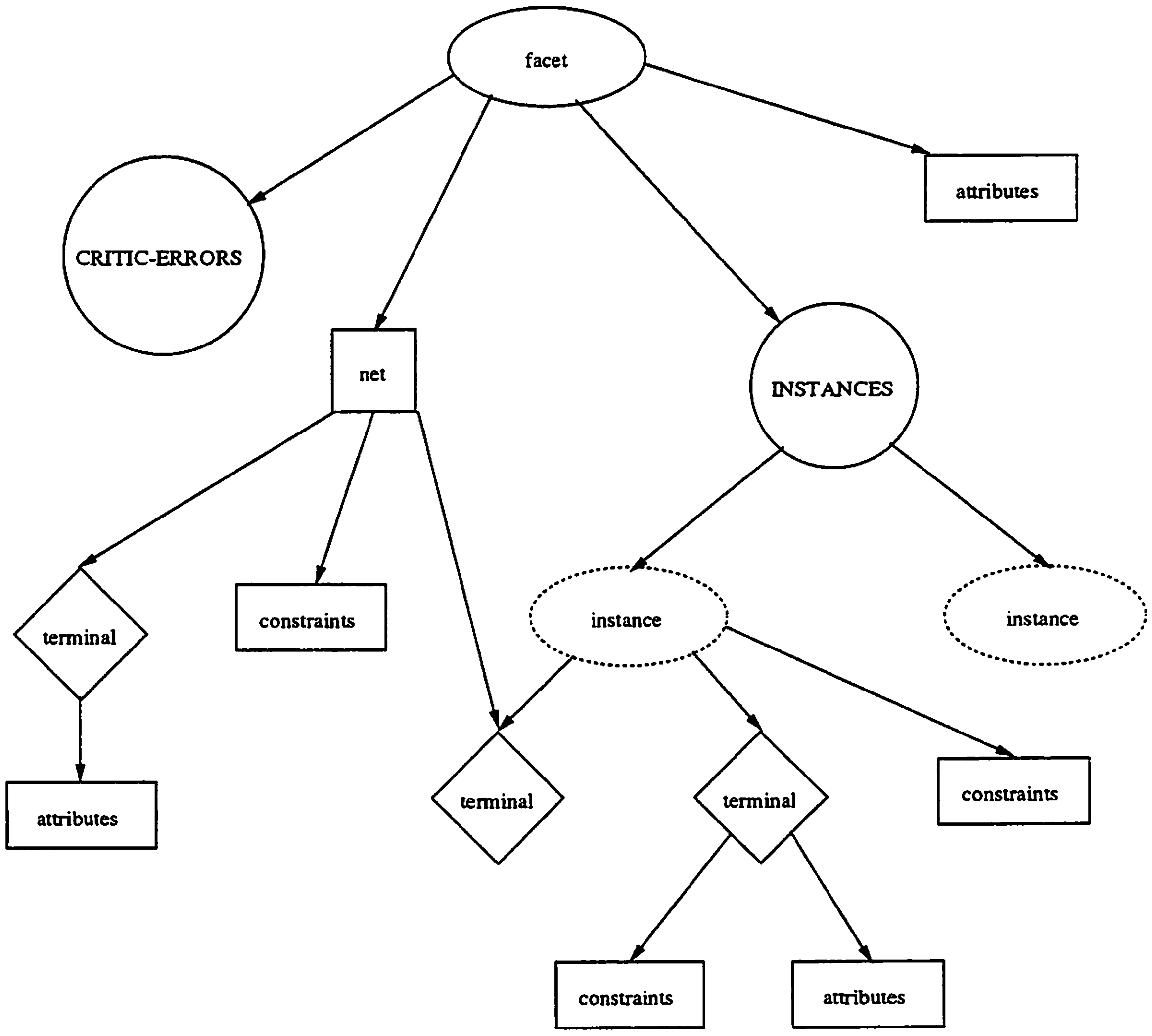


Figure 6.17: Structure Definition in OCT

the structure corresponds to the policies set forth for representing symbolic and schematic designs in OCT. The annotation includes fixed values (*i.e.* the type of the structure, extracted parameters) and calculated values. The calculated annotation describes attributes of the structure that are calculated once the structure is found or when the attribute is needed in another structure attribute or error check. In the implementation of **Critic**, the computed attributes are represented as Common Lisp expressions.

Error Descriptions

The errors associated with a particular node, primitive, or structure are stored with the definition of that item. A bag named CRITIC-ERRORS is attached to the facet that represents the node, primitive. or structure. The CRITIC-ERRORS bag contains zero or more bags, one bag for each error definition. Most of the fields of the error check (*in-context-of*, *non-in-context-of*, *check*, *message*, *comments*) are properties attached to the error check bag (see Figure 6.18). The *name* and *structure* fields are derived from the context and thus are not explicitly stored with the error. The *role* field is a property attached to the IN-CONTEXT-OF or NOT-IN-CONTEXT-OF property. The *constants* field is a bag named CONSTANTS attached to the error check bag and contains zero or more properties; each property represents a constant. Real and integer valued properties are fixed constants and string valued properties beginning with '(' and ending with ')' are Common Lisp expressions that are functions of the other constants in the error check and the constants in the process description.

The value of the *check* field is a Common Lisp expression that evaluates to T for an error or NIL for no error. In the semantics of CLOS, the expression is evaluated in the with context of the process constants, structure attributes, and error constants. The variable self is bound to the structure being checked. The variable error is bound to the error check. Accessing the value of an error check constant, process constant, or an attribute of the structure is by name. Accessing the value of an attribute of a component of the error structure is by:

```
(<attribute-name> <component-name>)
```

For example, if v-th and pb are process constants, e-const is an error check constant, sa and sb are structure attributes, and ca is a component of the structure with attribute attribute:

```
(> (* (/ v-th pb) e-const) (/ (- sa sb) (attribute ca)))
```

The following shows how **check** fields are coded. For example, a charge sharing check that is textually represented as:

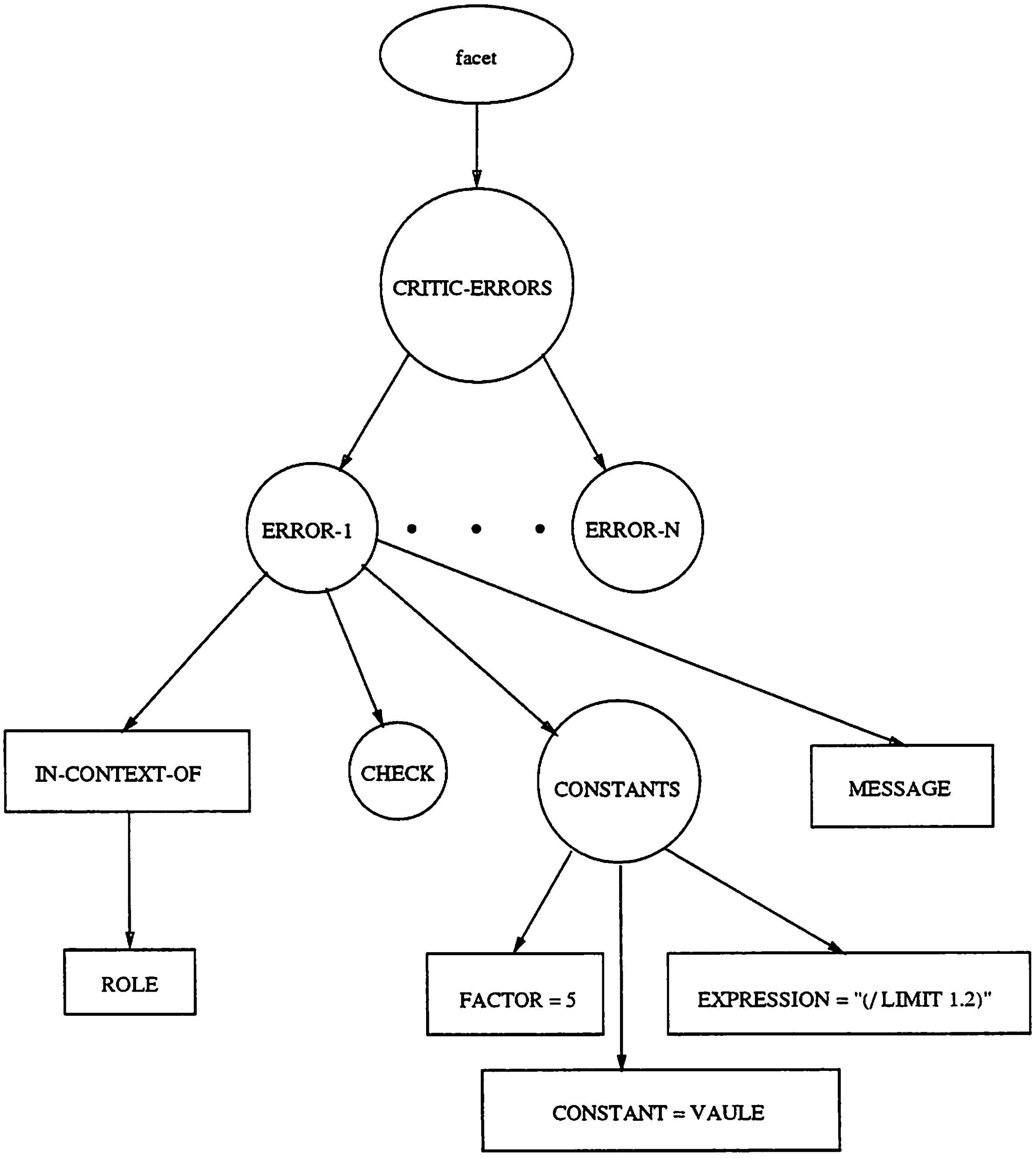


Figure 6.18: Error Definition in Oct

$$\frac{C_{middle-node}}{C_{middle-node} + C_{dg-internal}} < \frac{V_{th}}{V_{dd}}$$

would be expressed as:

```
(> (/ VTH-N VDD)
   (/ (CAPACITANCE MIDDLE-NODE)
      (+ (CAPACITANCE MIDDLE-NODE)
         (INTERNAL-CAPACITANCE DYNAMIC-GATE))))
```

As error checks were created, common functions and subexpressions were found. There common functions and subexpressions were abstracted out and provided as a standard set of functions for the creators of error checks. There were also several variables that could be conveniently defined.

Error message fields and constants, along with computed attributes in structures and primitives use the same syntax and semantics as described above.

Circuit Annotation

The **in-context-of, not-in-context-of,** and **role** fields of errors are used for limiting the number of false errors by making the context (under which the error check should be applied) more specific. However, there are times when the checks cannot be made specific enough or when the designers purposely violate rules. In this case the circuit must be annotated with information to direct the critic. The information can apply to the circuit as a whole, to regions of the circuit, or to individual elements. The method used in this work is annotation of individual elements with information about what error checks not to perform on them and on all structures that include them. This information is represented in OCT as a bag named DO-NOT-CHECK which is attached to the item that requires special handling (a primitive or node). The bag contains a property named ERROR-NAME which has as its value the name of the error not to check. The ERROR-NAME property contains another property named ERROR-STRUCTURE which has as its value the name of the type of structure or primitive that the special handling applies to (see Figure 6.19). Because the designer adds annotation to the circuit before the structure information exists in the data

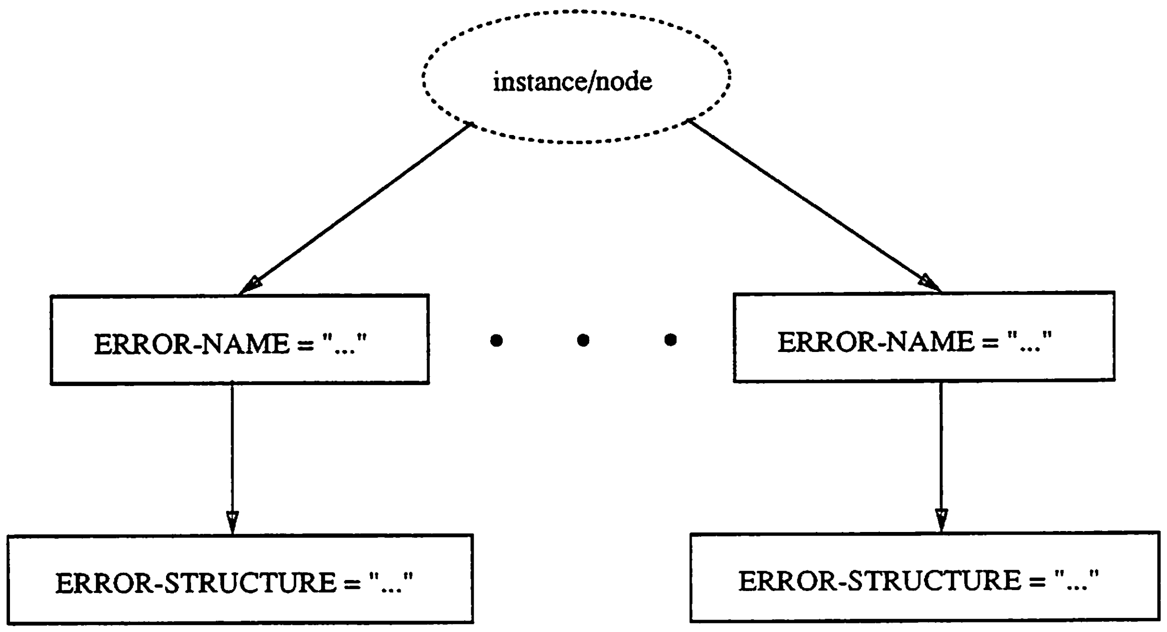


Figure 6.19: User Annotation in OCT

base, the extra annotation must be added to the primitives that will make up the structures. For example, a MOSFET might be annotated with information that directs the critic to not check the `too-small-width` rule for MOSFETs and the `weak-load-gate` rule for latches.

Structure/Error Persistence

Making the information found during a **Critic** run persist has a three major advantages: (1) subsequent **Critic** runs can use the information to reduce the amount of checking needed, (2) browsers and other tools can be developed for displaying and documenting the results of **Critic** runs, and (3) the display portion of **Critic** can be detached from the internal representation of the structures and errors. The following two sections describe how the structures and errors found during a **Critic** run are stored in the OCT database.

Structures

Each time a structure is found in the circuit, a bag with the name of the structure (and a property specifying the type) is created and the elements that make up the structure are grouped in the bag. Structures made up of other structures are bags made up of other bags and thus a hierarchy of bags that corresponds to the structure hierarchy being extracted is created. Properties are added to the bags to specify the attributes associated with the structures. As an example, Figure 6.20 shows a set of structure bags associated with a latch. The latch is composed of two inverters and a MOSFET. Each inverter is in turn composed of two MOSFETs.

Errors

The information necessary to display the error at some later time is the node, primitive, or structure in error (*item-in-error*), the name of the error check, and the message (since the message can be dependent on the context of the current session). Each error is represented as an OCT bag whose name is the name of the error check with a property named `ERROR-MESSAGE` attached to the bag. The item-in-error information is represented by attaching a bag named `CRITIC-ERRORS` to the item-in-error and attaching the error bags to the `CRITIC-ERRORS` bag (see Figure 6.21).

6.3 Browsing the Critic Knowledge Base

It is important that users who want to add to, delete from, or change the knowledge base be able to browse it. This should also be true for programs. One approach that can be used for browsing the knowledge base is to use the generic browsers that exist for the data base. There are two OCT browsers: **attache**[62] and **vem**[27]. **attache** is a text-based OCT data base browser that allows you to follow attachments in a facet and move between instances and their masters. **attache** can be used for following the attachments that were presented in Section 6.2. **vem** is the graphics editor/browser for OCT and allows the user to graphically traverse the data base and design hierarchy. While both of these programs can be used to browse and edit the knowledge base, some of the information in the knowledge

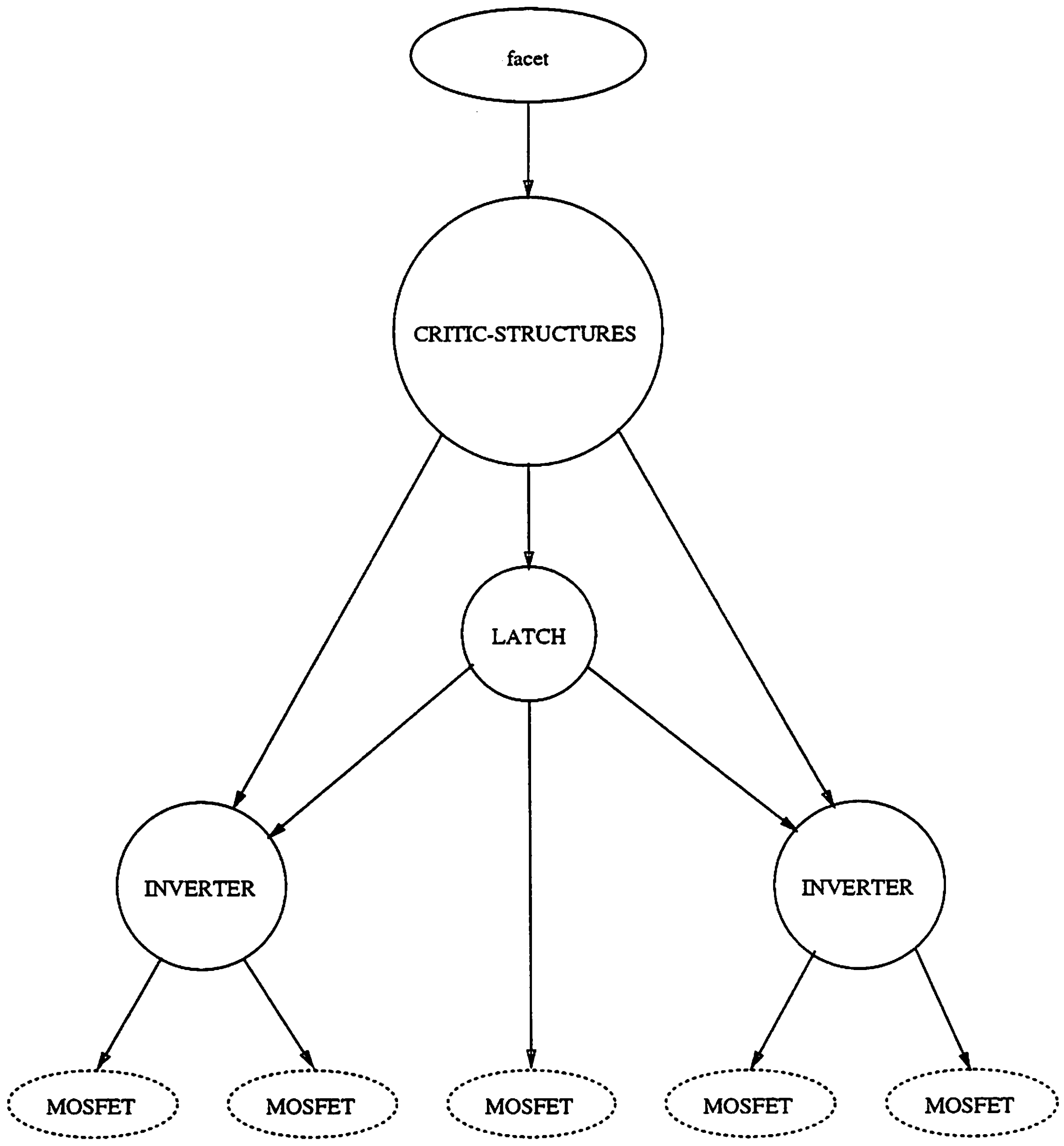


Figure 6.20: Structures Found During the **Critic** Execution in OCT

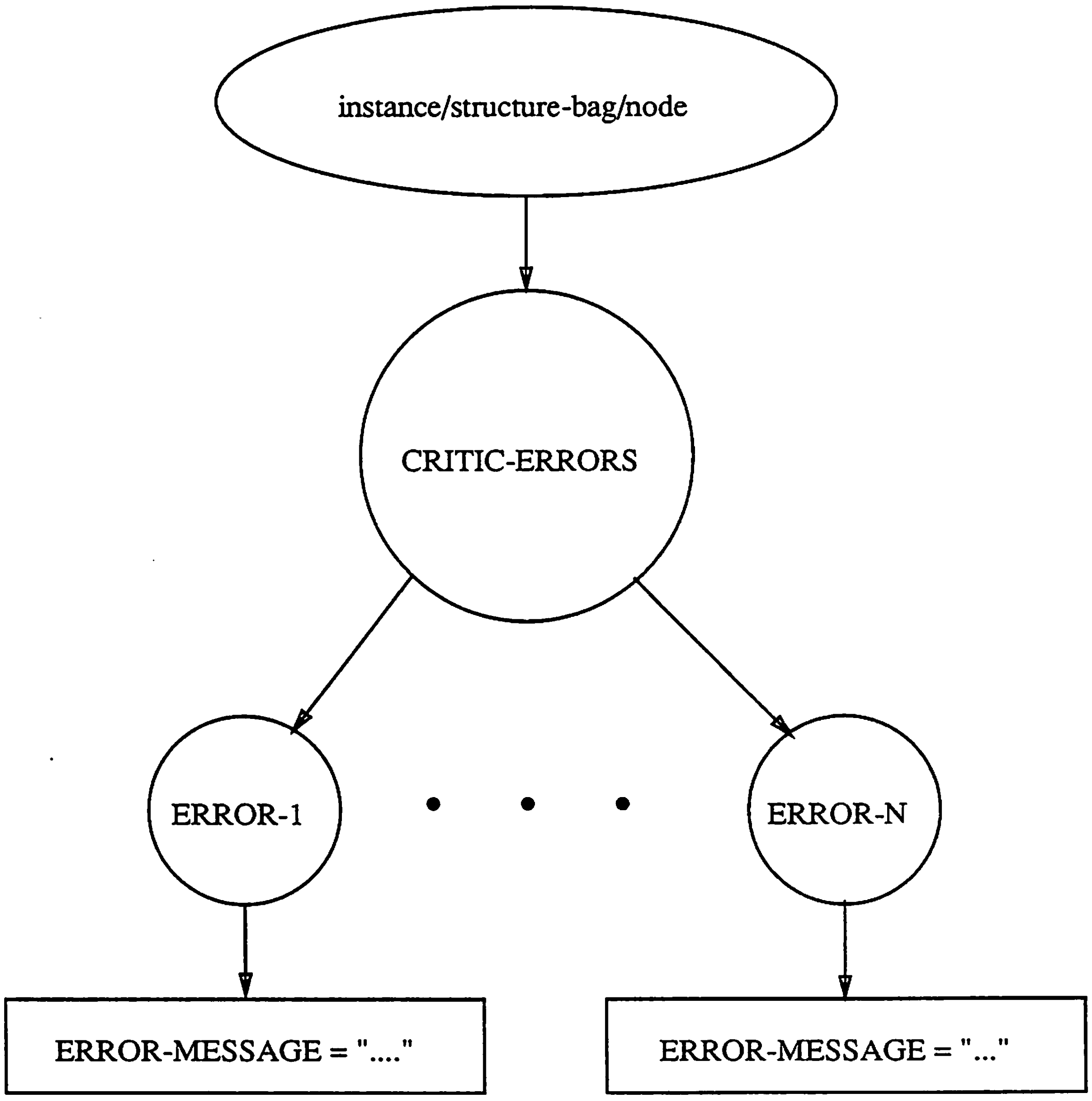


Figure 6.21: Errors Found During the **Critic** Execution in OCT

base is specific to the function of **Critic**, and generic browsing and display tools cannot use information about the structure and use of the data to display the much of the data in a meaningful form for the **Critic** user.

Because of the **Critic** specific information in the knowledge base, a browsing tool specific to **Critic**, **critic-browser**, was developed. **critic-browser** allows the user to traverse a knowledge base without having to know the structure of the data stored in OCT. The interface to **critic-browser** is X-windows based. The user starts up the program and a list of primitives and structures in the knowledge base are shown. Clicking on one of the primitives or structures causes a list of error checks for that particular item to be shown. Clicking on the error check shows a description of the error check.

6.4 Knowledge Base Documentation

It is common for documentation on a knowledge base or process or circuit design to get out of date with respect to the actual item being documented. Therefore the ability to automatically generate the documentation from the knowledge base is important. Since the knowledge base for **Critic** is stored in OCT and there is no direct way to generate a meaningful text hardcopy of the information in OCT, it is necessary to write specific routines to gather the information and put it in a form that can be used for documentation purposes. As **Critic** loads the knowledge base it can be told to generate a LaTeX document that describes the knowledge base. See Appendix A for the documentation of a simple ruleset.

Chapter 7

Example and Results

Critic has been run on a variety of example circuits: circuits specifically built to test certain features of **Critic**, circuits produced by module generators (**Critic** was used to test the quality of the designs), and student designs using a mixture of CAD tools and hand-design.

7.1 Example Critic Run

This section goes through an example **Critic** run showing how the user interacts with **Critic** and how **Critic** interacts with the design framework. It is assumed that the circuit has already been created (either by CAD tools or by hand in VEM) and that the **Critic** knowledge-base already exists.

The first step is to start the VEM graphics editor and open up a window on the circuit to be checked. In this example assume, the circuit is stored in the OCT facet *circuit:symbolic:contents*. To run VEM, type (make sure your `DISPLAY` variable is set to the name of your display, *e.g.* shambhala:0):

```
% vem &
```

You can override the `DISPLAY` environment variable by running VEM with an argument that is the display name:

```
% vem dent:0 &
```

VEM will prompt for a size and location for the *console* window. The console window is where all textual input and feedback is displayed by VEM. As you move the mouse cursor the window outline follows; move the mouse cursor to the top left corner of the screen and click the left button to get the default size.

To open a window on the circuit to be checked, issue the `open-window` command:

```
vem> "circuit:symbolic" : open-window   <carriage return>
```

All commands in VEM have three ways of being invoked, either by typing in the full name (prepended with a colon) and entering a carriage return, typing a single character, or selecting a menu entry. All commands in this example will be invoked by typing the full name.

After the `open-window` command is invoked VEM will prompt for the window size and location.

The best way to run **Critic** is with two windows displaying the circuit being checked, one displaying the entire circuit, called the *context* window, and the other displaying the items in error, called the *highlight* window. In this example, the first window opened will be the *context* window. To open the *highlight* window, move the mouse cursor into the context window and issue the **open-window** command.

```
vem> : open-window
```

With the mouse cursor in the *highlight* window, issue the **critic** command (either by type-in or by selection on the **Applications** menu pane):

```
vem> : critic
```

Now **Critic** has been started and is waiting for the user to tell it how to proceed, you can check the available **Critic** commands by holding down the shift key and pressing down and releasing the middle mouse button. Before checking the circuit, certain areas of the circuit should be marked to stop **Critic** from reporting some types of errors in those regions. This can be done by drawing a box around each region and then issuing the **add-to-not-check** command:

```
    vem> boxes(1) : add-do-not-check
```

The **add-do-not-check** command will create a dialog box that has a list of all primitive and structure types in the knowledge-base. The user selects the ones to be marked. For each one selected another dialog box is created that lists all errors associated with the primitive or element, and for each one of these selected, the primitives in the region will be marked so that **Critic** will not check for any of those errors (on the primitive or any structure the primitive is found to be contained in).

To specify what error checks will not be performed for a particular instance or net (and the structures that might contain the particular instance) select the object and issue the `show-do-not-check` command:

```
    vem> : select
    vem> objects(1) : show-do-not-check
```

Now that the initial state has been set, the check can be started by using the **run** command:

```
    vem> : run
```

Critic will create a window that displays the current status. The status window contains the following information: current phase (loading circuit, finding structures, finding errors), number of primitive items in the database, number of structures found, and number of errors found.

As **Critic** finds structures and errors in the circuit being checked, it annotates the database so that the next phase of **Critic**, displaying the results, can be done directly off the database rather than from internal data structures. **Critic** will print *Finished* in the VEM console window when it has completed finding the structures and errors in the circuit. Now that the structures and errors have been found, the use can sequence them.

The **next-structure** command will sequence through each structure found by **Critic**, zooming to the structure (in the window where the **next-structure** command was issued) and highlighting the components that make up the structure in all VEM windows that are displaying the components (in this example, the context window and the highlight window).

```
vem> : next-structure
```

The first structure found on this run is a **STATIC-GATE**.

The **next-error** command will sequence through each error found by **Critic**, zooming to the error (in the window where the **next-error** command was issued) and highlighting the structure in error in all VEM windows displaying the structure.

```
vem> : next-error
```

As the errors are sequenced through, the errors can be corrected by the user.

The first error found in this example is 'weak load gate'. This error is triggered when the load MOSFET of a latch is not big enough to force a state change. The simple fix is to replace the load device with a bigger one. This first step is to open a palette of standard sized MOSFETs. To make room for the new MOSFET, select items around the MOSFET and use the **drag-instance** command to move them (**drag-instance** stretches the interconnection paths and maintains interconnectivity).

```
vem> "xmosfet-palette" p
vem> boxes(1) : select-objects
vem> objects(N) : drag-instance
vem> : select-objects
vem> objects(N) : replace-instance
```

After each error is scanned (and possibly corrected) the user can go to the next error and the process repeats itself.

To end the **Critic** run:

```
vem> : exit
```

Critic leaves data in the circuit so that you can view it outside of **Critic** or restart **Critic** and go directly to **next-structure** and **next-error**. In particular there are two bags, `CRITIC-ERRORS` and `CRITIC-STRUCTURES`, that contain the errors and structures found during the **Critic** run. If you issue the **save-window** command with the mouse cursor in one of the windows displaying the circuit, this information will be made permanent:

```
    vem> : save-window
```

To view the contents of the CRITIC-ERRORS bag, issue the ***sel-bag-contents*** command, and select the item labeled CRITIC-ERRORS:

```
    vem> : sel-bag-contents
```

This will highlight all items that are in error. VEM will popup a dialog asking in what direction you want to follow the bag hierarchy. After selecting the direction to traverse (down), another dialog will popup up with a list of bags (*i.e.*, errors) that were found during the previous **Critic** run. Selecting one of these will highlight the items that are associated with that particular error.

To exit VEM, issue the *close-window* command from the console window:

```
    vem> : close-window
```

The *close-window* command will pop up a dialog box asking to confirm the exit.

7.2 Statistics

To check the ease of use of and cpu usage of **Critic** several example circuits were run. Some of the circuits were contrived to test various features of the program and to excite selected errors, others were from the Berkeley VLSI Design Course (CS250). For each of the examples shown in Table 7.1, the cpu time for each of the major cpu intensive phases of the **Critic** execution is shown (circuit load, structure finding, and error checking)[1]. Along with the cpu times is information about each circuit (number of components, number of nodes, number of structure found, and number of errors found). The results show that for circuit designs on the order of a few hundred components, the cpu is acceptable for an interactive application. As the circuit grows, structure finding (which can be an exponential problem) grows from a third of the total time to the majority of the time for execution.

[1]All CPU times are for a VAX 8800 Running ULTRIX 2.2.

Name	Circuit Load	Structure Finding	Error Checking	Total
dec	4	5	6	18
28 mosfets, 21 nodes, 20 structures, 9 errors				
CriticTest	6	4	6	18
35 mosfets, 27 nodes, 14 structures, 8 errors				
alu_odd	4	4	6	15
36 mosfets, 28 nodes, 10 structures, 12 errors				
critic-test	7	5	7	20
39 mosfets, 27 nodes, 17 structures, 11 errors				
alu	7	9	7	26
58 mosfets, 42 nodes, 26 structures, 12 errors				
NewCriticTest	8	10	9	30
68 mosfets, 56 nodes, 26 structures, 23 errors				
alu3	20	41	17	81
212 mosfets, 135 nodes, 58 structures, 32 errors				
IRMARPC	38	563	26	636
524 mosfets, 219 nodes, 130 structures, 201 errors				
big-test	106	914	103	1139
1248 mosfets, 771 nodes, 544 structures, 480 errors				

Table 7.1: Statistics for **Critic** Runs

7.3 Knowledge Collection

A major problem in building any knowledge-based system is collecting the knowledge [29, 55]. Many techniques have been developed for gathering the knowledge for VLSI design assistants [39, 47, 68]. The main purpose of **Critic** was not to collect knowledge of circuit design, but to explore representation and build a circuit critiquer that was integrated into an interactive design system. As such, the knowledge collection for **Critic** was more *ad hoc* than in other critics. The knowledge in **Critic** builds on the work of previous circuit critics [47], rules collected from design-styles [40, 24], from the author's own knowledge of design, and from industrial visitors. Many error checks were developed after spending time discussing **Critic** with industrial visitors. Rather than collecting knowledge by interviewing designers, the methods used in this and other circuits critics, work is being done on developing ways of discovering rules by watching designers[52, 53].

Chapter 8

Conclusions

In this dissertation, several ideas and algorithms for use in circuit critiquing were explored. These ideas and algorithms have been implemented in **Critic**, a test-bed for exploring ideas in circuit critiquing. **Critic** has been used to critique a number of student circuit designs from a VLSI design course at the University of California at Berkeley. A summary of the major results and possible future directions follows:

Major Results

Tight Integration

If the critiquer is tightly integrated with an existing CAD system, the users of the CAD system will be able to start using the critiquer with a minimum amount of effort. **Critic** was developed as a remote application for the VEM graphics editor and used the OCT data base for the storage of the knowledge base, circuits to be checked, and the results of the execution of **Critic** (*e.g.* the structures and errors found). By being a remote application for the VEM graphics editor, the user is able to quickly learn how to use the system since the control, entry and presentation of data is the same as other VEM remote applications.

Technology and Design Style Independence

Maintaining independence from a particular technology and design style allows the basic critiquer to be used for new technologies and design styles. **Critic** maintains independence by having no technology or design style specific information. **Critic** has knowledge about elements, terminals, nets, supplies, and clocks, but has no concept of the type of elements that can exist or the attributes that can be associated with them.

Representation of Primitives, Structures, and Errors

The specification of primitives, structures, and error checks, the method of describing them should be independent of the internals of the system. If the method of description matches a method already used in the design system, that is "by example", then that will further simplify the specification. Primitive and structure description by example was appropriate, but inappropriate for errors. **Critic** maintained a high level of system independence. There are three places where the system did not fully succeed: the specification of computed attributes, the specification of the check and message fields in error descriptions, and the specification and use of combinations. The first two of these use the implementation language and some information about how things are represented internally. The check and message fields and the computed attributes are written in Common Lisp and CLOS. The use of combinations introduces synthetic elements that are not part of the underlying design system. Explanation-Based Generalization was used to try to replace the synthetic elements with examples of combinations that the system could generalize from.

Future Work

There are several interesting areas that could be further explored. In particular, the areas of rechecking and incremental checking, the use of classes to minimize the amount of duplication, making the system specific to a small set of design styles or technologies, and further research in explanation-based generalization to simplify the specification of structures.

Rechecking and Incremental Checking

After fixing the errors found by a circuit critic, the user may want to recheck the circuit to make sure the errors were actually corrected and that no new errors were introduced; in certain cases the recheck can be sped up. If no changes have occurred that would change the structures found on the previous run, the structure finding phase can be skipped. This can be done when the following is true:

- No topological changes have been made to the circuit. If no nets have been modified, deleted, or created, the topology of the circuit has not been changed.
- No attributes that are used in structure finders have been changed. Some structure finding rules may put constraints on instances and nets (*e.g.* the type of a MOSFET must be a certain value or a net must be a supply net). If any of the attributes used in structure finders has been changed, the structures found on the previous run may be invalid. The critic could invalidate all structures of that type and all structures that use structures of that type (recursively up) and start the structure finding from that point.

In the error checking phase only the error checks that apply to new structures and those that use attributes that have been changed must be checked. Since attributes can depend on other attributes, a dependency graph must be made for each attribute, and if any attributes in the graph have changed, the calculated attribute must be assumed to have changed.

As has been shown in the area of design rule checking[58, 26], interactive, incremental checking for errors greatly helps in the design of circuits at the mask level. However, the same degree of checking that is done in incremental design rule checking is inappropriate for circuit critiquing. This is for two reasons: (1) as opposed to entering geometries, that are always design rule correct, as more complex structures are entered, at various points in time before the structure is complete, it may possibly be in violation of various error checks; and (2) running checks that a critiquer would use takes more time than the types of checks used by a design rule checker. For these reasons, the incremental checker should be invoked by the designer. After the designer has entered one or more complete structures, the critiquer can be told to check and report any errors in the previously entered structures.

The ability to do rechecks and incremental checking depends on the ability of the underlying database to determine what changes have occurred. The OCT data manager supports change detection. A program can register with OCT that certain operations (such as delete, modify, or create) on certain object types (such as net, instance, or layer) should be recorded. Each change is represented by the OCT change-record object. The change-record object contains information about the object that was changed (both before and after the change) and what operation was performed on the object. Change-objects are grouped by the change-list object. The change-list object is like other OCT objects and thus change-records are retrieved by creating a generator on the change-list. By using the change-list feature in OCT, both rechecking and incremental checking are possible.

Using User-Defined Structure Class Hierarchy

The current version of **Critic** has no hierarchy associated with the structures defined in the knowledge base. Thus information about structures and their error checks are duplicated. For example, a knowledge base might have definitions for a inverter, a more generic static-gate, and a dynamic-gate. There are information and error checks that are generic to gates that would cover all three types, and information and error checks that would cover the static-gates (which includes the inverter). Ways to represent the generic information should be explored. By deciding to use existing OCT representations for the structures in the knowledge base, use of generic information was not feasible. OCT symbolic/schematic policy does not have the ability to represent the generic or parameterized structures that would be necessary for this ability.

Limited Domains

One of the purposes of this research was to make a general system. However, the more general a system is, the more difficult it is to use. A general system can not have functions specific to a particular technology or design style, and there are few functions that are general enough to be included (count the number of items that are in series or parallel, *etc.*). Most users of this system will have only a small number of technologies and design styles available to them, so a less general system would suit their needs and be able to provide a set of primitives tailored to those technologies and design styles. By 'knowing' what the primitives are, functions that are specific to those primitives and the technology used can be provided for use in error checks.

By limiting the critic to a set of primitives and structures that are 'known' to the system, simulation of the circuit becomes possible (as with the use of SPICE in QCRITIC[3]). Simulation can be used for two different purposes: (1) as an input the to critiquer[3], and (2) as a way to verify that an error really exists. In the case of input to the critic, the simulation file can be used to determine if values that should never occur appear in the output of the simulator. For analog circuit simulations this could be voltages across devices that exceed the breakdown voltage for the device. For digital circuit simulations this could be undefined or high impedance values at locations where only low and high values are legal.

Many errors are too complicated to express in terms of a set of attribute constraints. In many cases what needs to be done is to take the structure being checked, add some context and inputs, build a simulation control file and send it to a simulator. This is good for error checks that point out possible errors, since simulation could weed out possible errors that are not really errors.

Explanation-Based Generalization

There are several additions/changes that could be done to this particular extension of Explanation-Based Generalization to make it more generally useful.

- Get rid of the fixed set of attributes. This is not a limit imposed by the concepts, just by the implementation. This can be removed by pre-processing the domain theory and examples to find all attributes and generating the necessary clauses for building the similarity lists and similarity tests. However, one could claim that this sort of information should actually be an explicit part of the domain theory.

- Get rid of the fixed groupings of attributes. This was introduced to limit the amount of bogus similarities and differences found by the system. One could remove the groupings and let the system look at all possible combinations of attributes inside and between generalizations. Doing this the system would generate large numbers of $V_k \neq V_j$[1], where V_j and V_k have nothing in common and probably are not even of the same type. For example: $combinationType \neq componentType, topNode \neq internalCapacitance$. Using type information would help limit the number.

- Better Similarity Checks. The similarity checks should be tuned to the type of the items being compared; for example, the function equality check should be used for function equality, inequality and dualism. Only simple literal equality/inequality checks were explored. This is something that can easily be added to the domain theory.

- Eliminate the Pre-Parsing Constraint. To simplify the experiment, the examples of static-gates were described in with the series-parallel combinations already determined. The system must be able to take examples directly from cell libraries and thus must be able to recognize and parse the series-parallel combinations.

[1] V_k is the kth attribute of the combination.

Appendix A

Ruleset Documentation

The following section contains an automatically generated ruleset document for a simple CMOS ruleset. One feature missing from the automatic documentation generator is the use of the LISP-to-English translator used when displaying the error messages in VEM.

CMOS

Introduction

This document describes the **scmos** ruleset for **Critic**. A ruleset describes the primitives that make up the circuits that will be checked with the ruleset, the structures (collections of primitives and other structures) that are to be found in the circuit, the error checks that are to be performed on the primitives and structures, and process and design style specific constants that can be used in the error checks.

Ruleset Introduction

This ruleset is used for testing out the various features of **Critic** and experimenting with various methods of representing the knowledge base. The ruleset is not meant to be a complete set of rules for finding errors in CMOS circuit designs.

Process

The **process** description contains constants and functions of constants that can be used by the **error checks** in the ruleset. Examples of constants are *threshold voltage*, *mobility*, and *breakdown voltage*. An example of a function of constants is the *mobility ratio*; the ratio of the p-type and n-type mobilities in a CMOS process.

Process constants and functions are stored as OCT properties on the process facet. The process facet is found via opening the ruleset facet and doing an instance get by name on “process”. Process constants are stored as OCT integer and real valued properties. Process functions are stored as OCT string valued properties that are converted to LISP s-expressions and are evaluated in the dynamic scope of the error being checked and the lexical scope of the process description. Because of this, process functions should only be functions of process constants and other process functions.

Since process descriptions are stored as OCT properties; process descriptions can be created and edited with VEM or ATTACHE, or created programmatically.

MOBILITY-N	0.475d0
MOBILITY-P	0.19d0
VTH-N	0.5d0
VTH-P	-0.5d0
MOBILITY-RATIO	(/ MOBILITY-P MOBILITY-N)
VDD	5.0d0

Node

The **node** is a basic object in **Critic**. Nodes are used for connecting together terminals. The **node** description contains descriptions of the **error checks** that apply to nodes, and the names of the node error checks which should not be checked if the node is connected to a formal terminal of the circuit.

Some types of errors do not apply for the nodes that are connected to the formal terminals of a circuit. For example, checking for a DC path to ground is probably incorrect for an input terminal of a circuit (in MOS technology, many cell inputs are only connected to gates of MOSFETs.

Errors

INCOMPATIBLE-FORMAL-TERMINALS

CHECK: -

```
(INCOMPATIBLE-FORMAL-TERMINALS-P SELF)
```

MESSAGE: - "The formal terminals on this node are incompatible. For example, both INPUT and OUTPUT formal terminals of the circuit being checked may be connected to this node"

CLOCK-AND-SUPPLY-ON-NODE

CHECK: -

```
(CLOCK-AND-SUPPLY-ON-NODE SELF)
```

MESSAGE: - "This node has CLOCK and SUPPLY terminals on it"

SIGNAL-INPUT-ON-SUPPLY-NODE

CHECK: -

```
(SIGNAL-INPUT-ON-SUPPLY SELF)
```

MESSAGE: - "This supply node has a SIGNAL/INPUT terminal on it. This may be ok, but I thought I would warn you"

SIGNAL-OUTPUT-ON-SUPPLY-NODE

CHECK: -

```
(SIGNAL-OUTPUT-ON-SUPPLY SELF)
```

MESSAGE: - "This supply node has a SIGNAL/OUTPUT terminal on it. This could cause a short if the OUTPUT tries to pull to the opposite supply"

OUTPUT-SHORT

CHECK: -

```
(OUTPUT-SHORT SELF)
```

MESSAGE: - "This node is connected to more than one terminal of type SIGNAL and direction OUTPUT. If the signals drive in opposite directions, a short will occur"

DC-PATH-TO-GROUND

CHECK: -

```
(NOT (PATH-BETWEEN SELF *DC-NODES*))
```

MESSAGE: - "The node does not have a DC path to GROUND, thus it can not be driven. If the node is connected to a formal INPUT terminal of the circuit being checked, CRITIC will not list it as in error"

FLOATING-TERMINAL

CHECK: -

```
(FLOATING-TERMINAL SELF)
```

MESSAGE: - "This node is connected to only one element"

ONLY-INPUTS

CHECK: -

```
(ONLY-INPUTS SELF)
```

MESSAGE: - "This node is only connected to INPUT terminals, thus the node is floating. If the node is connected to a formal INPUT terminal of the circuit being checked, CRITIC will not list it as in error"

Error Checks to Skip on Input Terminals

```
"DC-PATH-TO-GROUND"
"FLOATING-TERMINAL"
"ONLY-INPUTS"
```

Error Checks to Skip on Output Terminals

```
"FLOATING-TERMINAL"
```

Primitives

A **primitive** is the lowest level item that **Critic** deals with. Primitives describe the cells that are read from a circuit description. Primitives can range from mosfets, capacitors, bipolar transistors, and resistors, to macro cells of any size and complexity

A primitive description contains the names of the attributes that should be read from the database when processing an instance of the primitive, attributes that are functions of other attributes of the instance (and process parameters), the terminals of the primitive, and any error checks associated with the primitive.

MOSFET

Attributes

`WOVERL: (/ MOSFETWIDTH MOSFETLENGTH)`

`MOSFETTYPE:` read from the database

`MOSFETLENGTH:` read from the database

`MOSFETWIDTH:` read from the database

Terminals

`GATE` with direction `INPUT` of type `SIGNAL`

`SOURCE` with direction `INOUT` of type `SIGNAL`

`DRAIN` with direction `INOUT` of type `SIGNAL`

The following terminals are permutable: `(SOURCE DRAIN)`

The following terminals are DC equivalent: `(SOURCE DRAIN)`

Errors

TOO-SMALL-LENGTH

CHECK: -

```
(< MOSFETLENGTH MINLENGTH)
```

MESSAGE: -

```
(FORMAT NIL "The mosfet has a channel length
of ~,2G meters which is below the minimum
allowed length of ~,2G meters~%"
        MOSFETLENGTH MINLENGTH)
```

CONSTANTS: -

```
MINLENGTH: 1.2d-6
```

TOO-SMALL-WIDTH

CHECK: -

```
(< MOSFETWIDTH MINWIDTH)
```

MESSAGE: -

```
(FORMAT NIL "The mosfet has a channel width
of ~,2G meters which is below the minimum
allowed width of ~,2G meters~%"
        MOSFETWIDTH MINWIDTH)
```

CONSTANTS: -

```
MINWIDTH: 1.4d-6
```

CAPACITOR

Attributes

`CAPACITANCE:` read from the database

Terminals

`BOTTOM` with direction `INOUT` of type `SIGNAL`

`TOP` with direction `INOUT` of type `SIGNAL`

The following terminals are permutable: `(BOTTOM TOP)`

Errors

TOO-LARGE

CHECK: -

```
(> VALUE MAXIMUM-CAPACITANCE)
```

MESSAGE: - `"The value of the capacitor is too large"`

CONSTANTS: -

```
MAXIMUM-CAPACITANCE: 1.0d-6
```

Combinations

Combinations are used for describing series/parallel collections of other *circuit* objects in **Critic**.

A combination description contains information on what item to combine, what terminals of the item to use, what attributes to compute for the new combination, and a set of constraints on the items that make up the combination. Also, error checks....

MOSFET-COMBINATION

Element to combine: MOSFET using terminals DRAIN and SOURCE

Attributes

`WOVERL-MAX` with components:

```
ACCESSOR WOVERL

SERIES (LAMBDA (LST) (/ 1.0 (REDUCE (FUNCTION +) (MAPCAR
    (FUNCTION (LAMBDA (X) (/ 1.0 X))) LST))))

PARALLEL (LAMBDA (LST) (REDUCE (FUNCTION MAX) LST))
```

`WOVERL-MIN` with components:

```
ACCESSOR WOVERL

SERIES (LAMBDA (LST) (/ 1.0 (REDUCE (FUNCTION +) (MAPCAR
    (FUNCTION (LAMBDA (X) (/ 1.0 X))) LST))))

PARALLEL (LAMBDA (LST) (REDUCE (FUNCTION +) LST))
```

`WOVERL` with components:

```
ACCESSOR WOVERL

SERIES (LAMBDA (LST) (/ 1.0 (REDUCE (FUNCTION +) (MAPCAR
    (FUNCTION (LAMBDA (X) (/ 1.0 X))) LST))))
    PARALLEL (LAMBDA (LST) (REDUCE (FUNCTION +) LST))
```

`LOGIC-FUNCTION` with components:

```
    ACCESSOR (OCTID (NODE GATE))

    SERIES (LAMBDA (LST) (CONS (QUOTE AND) LST))

    PARALLEL (LAMBDA (LST) (CONS (QUOTE OR) LST))
```

`MOSFETTYPE` with components:

```
    ACCESSOR MOSFETTYPE

    SERIES CAR

    PARALLEL CAR
```

The following terminals are to be collected: `(GATE)` The following attributes are constraints: `(MOSFETTYPE)`

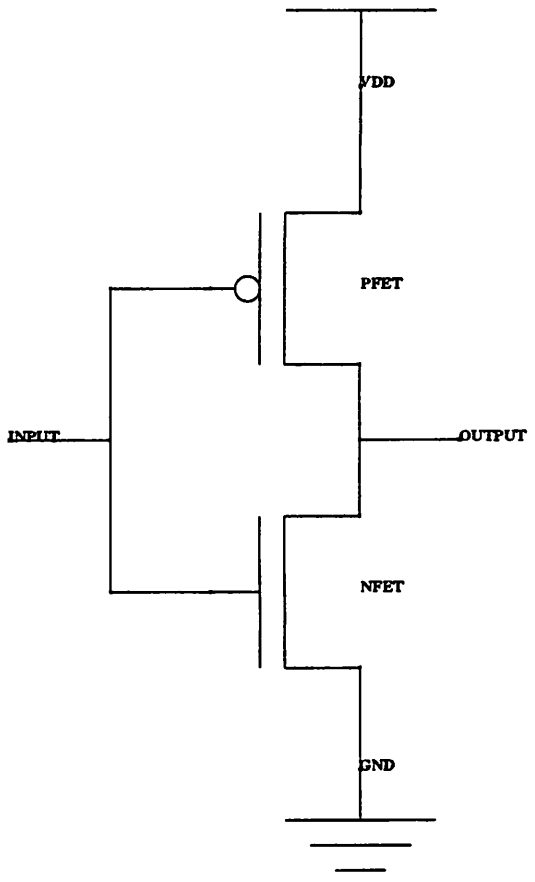


Figure A.1: Schematic of INVERTER

Structures

Structures are used for classifying collections of items in a circuit. A **structure** is an interconnection of **primitives** and other **structures**.

A structure description contains information on what items make up the structure, how they are connected, constraints upon the items and connections, attributes that are functions of attributes associated with the items that make up the structure, and any error checks associated with the structure.

INVERTER

Attributes

```
NOVERP: (/ (WOVERL NFET) (WOVERL PFET))
```

Patterns

NFET

```
(MOSFET (DRAIN (VARIABLE T8)) (GATE (VARIABLE T2))
 (SOURCE (VARIABLE T4)) (MOSFETTYPE "NENH"))
```

PFET

```
(MOSFET (SOURCE (VARIABLE T6)) (DRAIN (VARIABLE T4))
 (GATE (VARIABLE T2)) (MOSFETTYPE "PENH"))

(LISP-EVAL (GROUND-NODE-P (VARIABLE-VALUE 'T8)))

(LISP-EVAL (SUPPLY-NODE-P (VARIABLE-VALUE 'T6)))
```

Terminals

GND with direction INOUT of type GROUND tied to the node matched by variable T8

VDD with direction INOUT of type SUPPLY tied to the node matched by variable T6

OUTPUT with direction OUTPUT of type SIGNAL tied to the node matched by variable T4

INPUT with direction INPUT of type SIGNAL tied to the node matched by variable T2

Errors

EQUAL-RISE-AND-FALL-TIMES

NOT-IN-CONTEXT-OF: (DOMINO-GATE NIL)

CHECK: -

```
    (OR (< NOVERP MIN-RATIO) (> NOVERP MAX-RATIO))
```

MESSAGE: -

```
(FORMAT NIL "The N and P device W/L's of this
inverter are not ratioed such that the rise and
fall times of the inverter will be approximately
equal.  The ratio of W/L's is ~,2F and should
fall in the range of [~,2F, ~,2F].
This may not be necessary if the circuit
has been optimized globally.  This also is
not necessary if the inverter is used as the
static output buffer of a domino gate.~%"
        NOVERP MIN-RATIO MAX-RATIO)
```

CONSTANTS: -

```
MAX-RATIO: (* 1.15 MOBILITY-RATIO)
MIN-RATIO: (* 0.85 MOBILITY-RATIO)
```

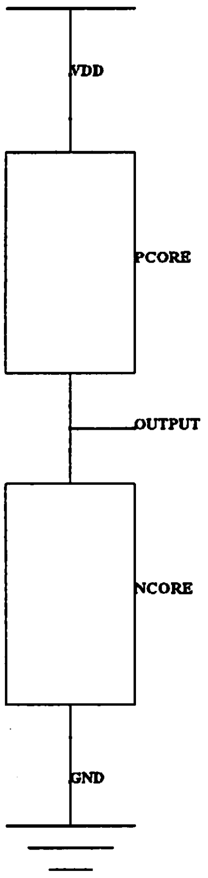


Figure A.2: Schematic of STATIC-GATE

STATIC-GATE

Attributes

```
NOVERP: (/ (WOVERL NCORE) (WOVERL PCORE))
```

Patterns

NCORE

```
(MOSFET-COMBINATION (BOTTOM (VARIABLE T14))
 (TOP (VARIABLE T10)) (MOSFETTYPE "NENH"))
```

PCORE

```
(MOSFET-COMBINATION (BOTTOM (VARIABLE T10))
 (TOP (VARIABLE T12)) (MOSFETTYPE "PENH"))
(LISP-EVAL (GROUND-NODE-P (VARIABLE-VALUE 'T14)))
(LISP-EVAL (SUPPLY-NODE-P (VARIABLE-VALUE 'T12)))
```

Terminals

`INPUT` with direction `INOUT` of type `SIGNAL` tied to the node matched by variable `T16`

`GROUND` with direction `INPUT` of type `GROUND` tied to the node matched by variable `T14`

`VDD` with direction `INPUT` of type `SUPPLY` tied to the node matched by variable `T12`

`OUTPUT` with direction `OUTPUT` of type `SIGNAL` tied to the node matched by variable `T10`

Errors

P-SERIES-LENGTH

CHECK: -

```
(> (SERIES-LENGTH PCORE) P-LIMIT)
```

MESSAGE: -

```
(FORMAT NIL
        "Too many mosfets in the P-type mosfet block.
There are ~A items in series and there may only be ~A.
Too many series connected mosfets can increase the time
it takes to charge the capacitance in the gate~%"
        (SERIES-LENGTH PCORE) P-LIMIT)
```

CONSTANTS: -

```
P-LIMIT: 3
```

EQUAL-RISE-AND-FALL-TIMES

NOT-IN-CONTEXT-OF: (DOMINO-GATE NIL)

CHECK: -

```
(OR (> NOVERP MAX-RATIO) (< NOVERP MIN-RATIO))
```

MESSAGE: -

```
(FORMAT NIL
        "The rise and fall times of the static gate are
not equal.  The N/P ratio is ~,2F and should be
within the range [~,2F, ~,2F]"
        NOVERP MIN-RATIO MAX-RATIO)
```

CONSTANTS: -

```
MAX-RATIO: (* 1.15 MOBILITY-RATIO)
MIN-RATIO: (* 0.85 MOBILITY-RATIO)
```

LOGIC-DUALITY

CHECK: -

```
(NOT (LOGIC-EQUAL (LOGIC-FUNCTION NCORE)
                  (LOGIC-DUALIZE (LOGIC-FUNCTION PCORE))))
```

MESSAGE: -

```
(FORMAT NIL
        "The logic functions of the N and P mosfet blocks
are not logical duals of each other.  There are
certain input conditions that will cause the output
the float or be pulled towards both supplies (i.e. a
short)~%~%LF1=~A~%~%LF2=~A~%"
         (LOGIC-FUNCTION NCORE) (LOGIC-FUNCTION PCORE))
```

N-SERIES-LENGTH

CHECK: -

```
(> (SERIES-LENGTH NCORE) N-LIMIT)
```

MESSAGE: -

```
(FORMAT NIL
       "Too many mosfets in the N-type mosfet block.
There are ~A items in series and there may only be ~A.
Too many series connected mosfets can increase the
time it takes to charge the capacitance in the gate~%."
       (SERIES-LENGTH NCORE) N-LIMIT)
```

CONSTANTS: -

```
N-LIMIT: 3
```

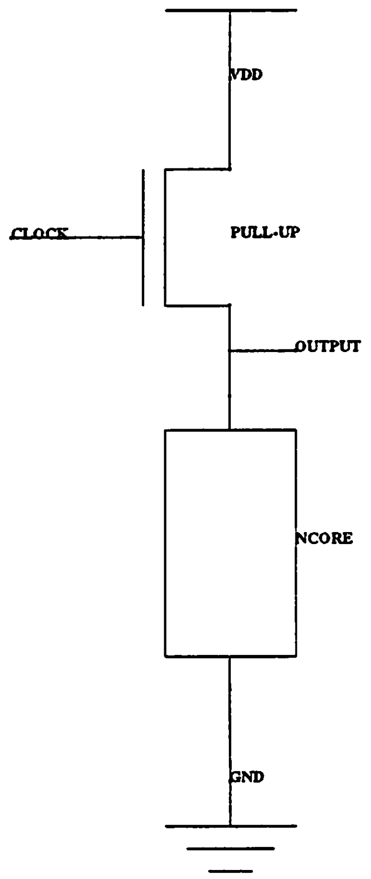


Figure A.3: Schematic of DYNAMIC-GATE

DYNAMIC-GATE

Attributes

```
INTERNAL-CAPACITANCE: (CAPACITANCE NCORE)
```

Patterns

PULL-UP

```
(MOSFET (SOURCE (VARIABLE T20)) (DRAIN (VARIABLE T22))
 (GATE (VARIABLE T24)))
```

NCORE

```
(MOSFET-COMBINATION (BOTTOM (VARIABLE T18))
 (TOP (VARIABLE T22)) (BOTTOM-CLOCKED "YES")
 (MOSFETTYPE "NENH"))

(LISP-EVAL (CLOCK-NODE-P (VARIABLE-VALUE 'T24)))

(LISP-EVAL (SUPPLY-NODE-P (VARIABLE-VALUE 'T20)))

(LISP-EVAL (GROUND-NODE-P (VARIABLE-VALUE 'T18)))
```

Terminals

`CLOCK` with direction `INPUT` of type `CLOCK` tied to the node matched by variable `T24`

`OUTPUT` with direction `OUTPUT` of type `SIGNAL` tied to the node matched by variable `T22`

`VDD` with direction `INPUT` of type `SUPPLY` tied to the node matched by variable `T20`

`GROUND` with direction `INPUT` of type `GROUND` tied to the node matched by variable `T18`

Errors

SERIES-LENGTH

CHECK: -

```
    (> (SERIES-LENGTH NCORE) N-LIMIT)
```

MESSAGE: - `"Too many mosfets in series"`

CONSTANTS: -

```
    N-LIMIT: 5
```

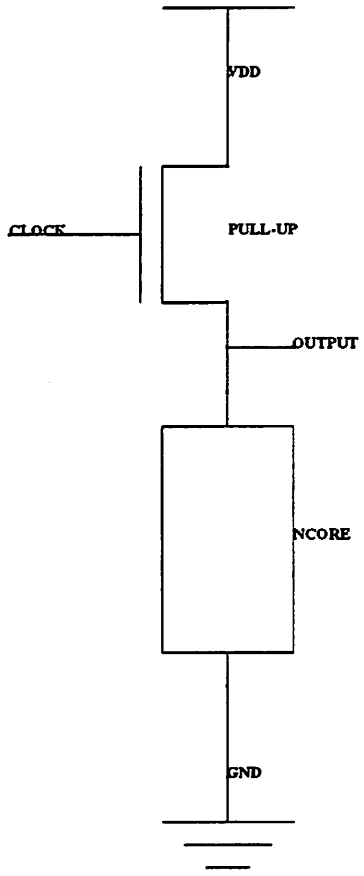


Figure A.4: Schematic of PRECHARGE-GATE

PRECHARGE-GATE

Patterns

PULL-UP

```
(MOSFET (SOURCE (VARIABLE T28)) (DRAIN (VARIABLE T30))
 (GATE (VARIABLE T32)))
```

NCORE

```
(MOSFET-COMBINATION (BOTTOM (VARIABLE T26))
 (TOP (VARIABLE T30)) (BOTTOM-CLOCKED "NO")
 (MOSFETTYPE "NENH"))

(LISP-EVAL (CLOCK-NODE-P (VARIABLE-VALUE 'T32)))
```

```
(LISP-EVAL (SUPPLY-NODE-P (VARIABLE-VALUE 'T28)))

(LISP-EVAL (GROUND-NODE-P (VARIABLE-VALUE 'T26)))
```

Terminals

CLOCK with direction INPUT of type CLOCK tied to the node matched by variable T32

OUTPUT with direction OUTPUT of type SIGNAL tied to the node matched by variable T30

VDD with direction INPUT of type SUPPLY tied to the node matched by variable T28

GROUND with direction INPUT of type GROUND tied to the node matched by variable T26

Errors

SERIES-LENGTH

CHECK: -

```
    (> (SERIES-LENGTH NCORE) N-LIMIT)
```

MESSAGE: - "Too many mosfets in series"

CONSTANTS: -

```
    N-LIMIT: 5
```

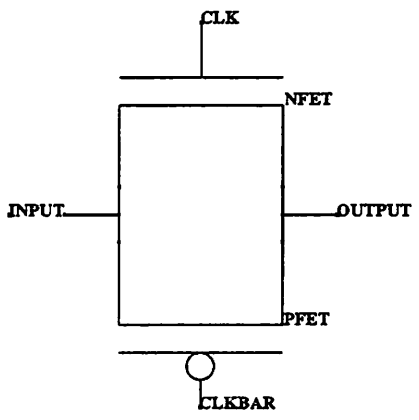


Figure A.5: Schematic of TRANSFER-GATE

TRANSFER-GATE

Patterns

PFET

```
(MOSFET (SOURCE (VARIABLE T34)) (DRAIN (VARIABLE T36))
 (GATE (VARIABLE T40)) (MOSFETTYPE "PENH"))
```

NFET

```
(MOSFET (DRAIN (VARIABLE T36)) (GATE (VARIABLE T38))
 (SOURCE (VARIABLE T34)) (MOSFETTYPE "NENH"))

(LISP-EVAL (CLOCK-NODE-P (VARIABLE-VALUE 'T40)))

(LISP-EVAL (CLOCK-NODE-P (VARIABLE-VALUE 'T38)))
```

Terminals

`CLKBAR` with direction `INPUT` of type `CLOCK` tied to the node matched by variable `T40`

`CLK` with direction `INPUT` of type `CLOCK` tied to the node matched by variable `T38`

`OUTPUT` with direction `INOUT` of type `SIGNAL` tied to the node matched by variable `T36`

`INPUT` with direction `INOUT` of type `SIGNAL` tied to the node matched by variable `T34`

The following terminals are permutable: `(OUTPUT INPUT)`

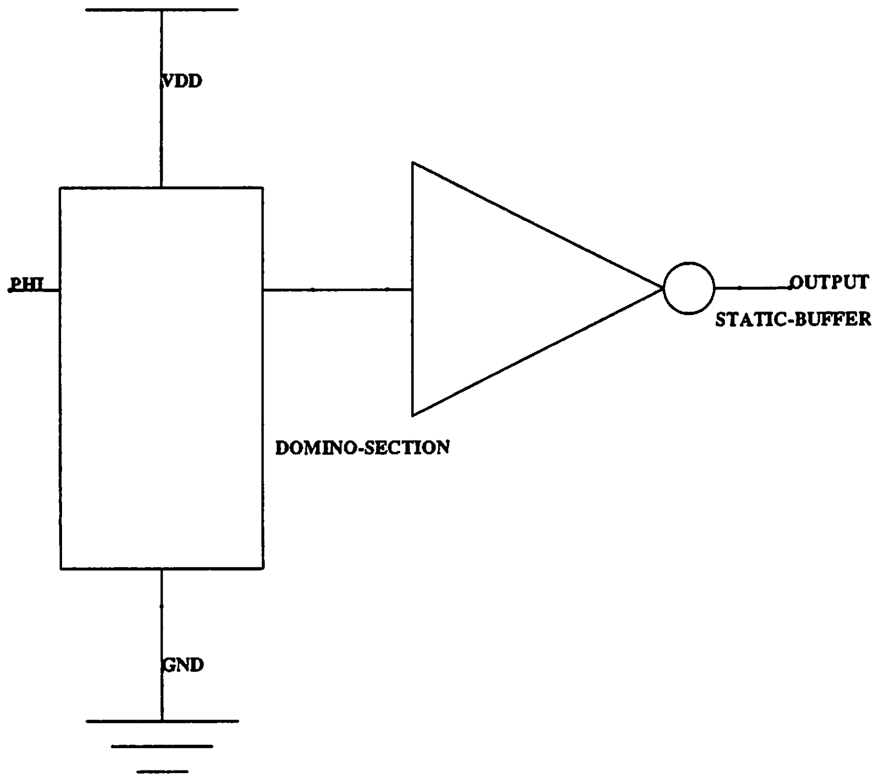


Figure A.6: Schematic of DOMINO-GATE

DOMINO-GATE

Patterns

STATIC-BUFFER

```
(INVERTER (GND (VARIABLE T42)) (VDD (VARIABLE T44))
 (OUTPUT (VARIABLE T46)) (INPUT (VARIABLE T49)))
```

DOMINO-SECTION

```
(DYNAMIC-GATE (CLOCK (VARIABLE T48))
 (OUTPUT (VARIABLE T49)) (VDD (VARIABLE T44))
 (GROUND (VARIABLE T42)))

(LISP-EVAL (CLOCK-NODE-P (VARIABLE-VALUE 'T48)))

(LISP-EVAL (SUPPLY-NODE-P (VARIABLE-VALUE 'T44)))

(LISP-EVAL (GROUND-NODE-P (VARIABLE-VALUE 'T42)))
```

Terminals

`PHI` with direction `INPUT` of type `CLOCK` tied to the node matched by variable `T48`

`OUTPUT` with direction `OUTPUT` of type `SIGNAL` tied to the node matched by variable `T46`

VDD with direction `INPUT` of type `SUPPLY` tied to the node matched by variable `T44`

GROUND with direction `INPUT` of type GROUND tied to the node matched by variable `T42`

Errors

CHARGE-REDISTRIBUTION

CHECK: -

```
(< (/ (CAPACITANCE (NODE (INPUT STATIC-BUFFER)))
      (+ (CAPACITANCE (NODE (INPUT STATIC-BUFFER)))
         (INTERNAL-CAPACITANCE DOMINO-SECTION)))
   (/ VTH-N VDD))
```

MESSAGE: -

```
(FORMAT NIL
        "The load capacitance of the dynamic-gate (~8,3G)
is not large enought to inhibit charge sharing problems.
The internal capacitance of the dynamic-gate is ~8,3G"
        (CAPACITANCE (NODE (INPUT STATIC-BUFFER)))
        (INTERNAL-CAPACITANCE DOMINO-SECTION))
```

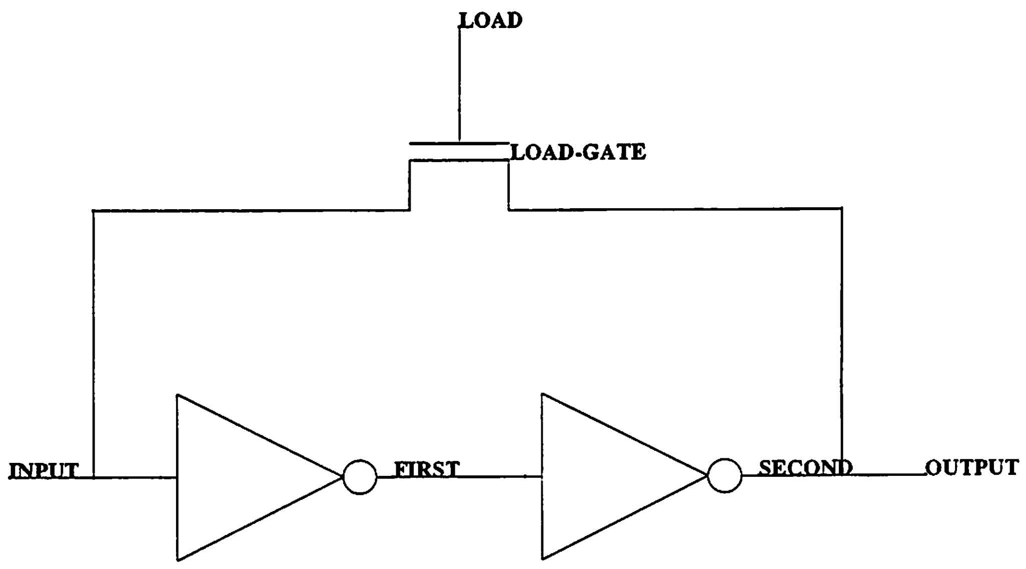


Figure A.7: Schematic of LATCH

LATCH

Patterns

LOAD-GATE

```
(MOSFET (SOURCE (VARIABLE T55)) (DRAIN (VARIABLE T57))
 (GATE (VARIABLE T59)) (MOSFETTYPE "PENH"))
```

SECOND

```
(INVERTER (GND (VARIABLE T53)) (VDD (VARIABLE T51))
 (OUTPUT (VARIABLE T57)) (INPUT (VARIABLE T60)))
```

FIRST

```
(INVERTER (GND (VARIABLE T53)) (VDD (VARIABLE T51))
 (OUTPUT (VARIABLE T60)) (INPUT (VARIABLE T55)))

(LISP-EVAL (GROUND-NODE-P (VARIABLE-VALUE 'T53)))

(LISP-EVAL (SUPPLY-NODE-P (VARIABLE-VALUE 'T51)))
```

Terminals

LOAD with direction INPUT of type SIGNAL tied to the node matched by variable T59

OUTPUT with direction OUTPUT of type SIGNAL tied to the node matched by variable T57

INPUT with direction INPUT of type SIGNAL tied to the node matched by variable T55

GND with direction INPUT of type GROUND tied to the node matched by variable T53

VDD with direction INPUT of type SUPPLY tied to the node matched by variable T51

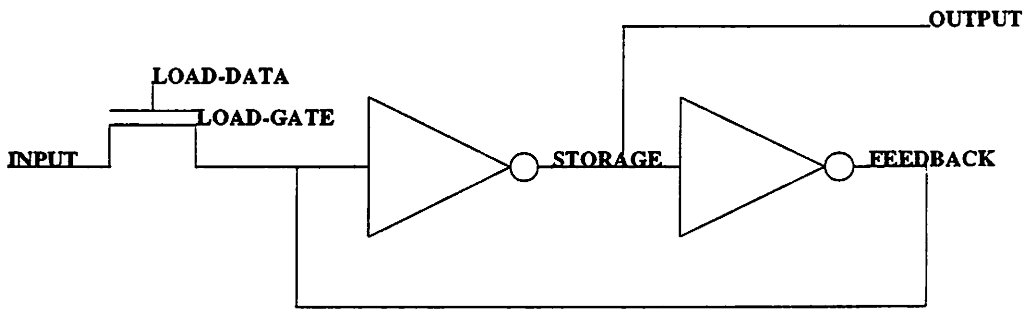


Figure A.8: Schematic of INVERTING-LATCH

INVERTING-LATCH

Patterns

LOAD-GATE

```
(MOSFET (DRAIN (VARIABLE T64)) (GATE (VARIABLE T66))
 (SOURCE (VARIABLE T71)) (MOSFETTYPE "NENH"))
```

FEEDBACK

```
(INVERTER (GND (VARIABLE T62)) (VDD (VARIABLE T70))
 (OUTPUT (VARIABLE T71)) (INPUT (VARIABLE T68)))
```

STORAGE

```
(INVERTER (GND (VARIABLE T62)) (VDD (VARIABLE T70))
 (OUTPUT (VARIABLE T68)) (INPUT (VARIABLE T71)))
```

```
(LISP-EVAL (SUPPLY-NODE-P (VARIABLE-VALUE 'T70)))
```

```
(LISP-EVAL (GROUND-NODE-P (VARIABLE-VALUE 'T62)))
```

Terminals

VDD with direction `INOUT` of type `SUPPLY` tied to the node matched by variable `T70`

`OUTPUT` with direction `OUTPUT` of type `SIGNAL` tied to the node matched by variable `T68`

LOAD-DATA with direction INPUT of type SIGNAL tied to the node matched by variable T66

INPUT with direction INPUT of type SIGNAL tied to the node matched by variable T64

GND with direction INOUT of type GROUND tied to the node matched by variable T62

Errors

WEAK-LOAD-GATE

CHECK: -

```
(OR (< (WOVERL LOAD-GATE) (* 6 (WOVERL (NFET FEEDBACK))))
    (< (WOVERL LOAD-GATE) (* 1 (WOVERL (PFET FEEDBACK)))))
```

MESSAGE: -

```
(FORMAT NIL "The load gate is too small to
change the state of the latch.  The W/L of
the load gate is ~,2F, the gate must have a
W/L of at least ~,2F to be able to change the
state."
        (WOVERL LOAD-GATE)
        (MAX (* 6 (WOVERL (NFET FEEDBACK)))
             (* 1 (WOVERL (PFET FEEDBACK)))))
```

COMMENTS: - assumptions: mobility ratio is 2, VT is small compared to VDD, LOAD-GATE is a single N-type mosfet, not a full transfer gate.

Bibliography

[1] V. D. Agarwal, S. K. Jain, and D. M. Singer. Automation in Design for Testability. In *The Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1984.

[2] Electronic Industries Association. EDIF - Electronic Design Interchange Format Version 2.0.0. Technical report, Electronic Industries Association, 1987.

[3] Sue Bergquist and Robert Sparkes. QCritic: A Rule-Based Analyzer for Bipolar Analog Circuit Designs. In *The Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 617–620, May 1986.

[4] W. P. Birmingham, Anurag P. Gupta, and D. P. Siewiorek. The MICON System for Computer Design. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 135–140, Las Vegas, Nevada, June 1989.

[5] D. Bobrow and M. Stefik. The LOOPS Manual. Technical Report KB-VLSI-81-13, XEROX Palo Alto Research Center, 1981.

[6] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. Technical Report X3J13 Document 88-002R, American National Standards Institute, June 1988.

[7] I. Bolsens, W. De Rammelaere, L. Clausen, and H. DeMan. Electrical Debugging of Synchronous MOS VLSI Circuits Exploiting Analysis of the Intended Logic Behaviour. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 513–518, Las Vegas, Nevada, June 1989.

[8] Harold Brown and Mark Stefik. Palladio: An Expert Assistant for Integrated Circuit Design. Technical Report HPP-82-5, Heuristic Programming Project, Stanford University, Palo Alto, California, April 1982.

[9] Harold Brown, Christopher Tong, and Gordon Foyster. Palladio: An Exploratory Environment for Circuit Design. *Computer*, 16(12):41–56, December 1983.

[10] P. Camurati, P. Gianoglio, R. Gianoglio, and P. Prinetto. ESTA: An Expert System for DFT Rule Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-7(11):1172–1180, November 1988.

[11] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.

[12] Mike Creech, Dan Flickinger, Pierre Huyn, Mike Lemon, Reed Letsinger, Derek Proudian, Steven Rosenberg, and Steve Weyer. Guide to the Heuristic Programming and Representation Language, Part 3: Environment and I/O. Technical Report AT-MEMO-83-5, Hewlett-Packard Computer Research Center, July 1984.

[13] G. DeJong and R. Mooney. Explanation-Based Learning: An Alternative View. *Machine Learning*, 1(2):145–226, 1986.

[14] H. DeMan, I. Bolsens, E. van der Meersch, and J. van Cleynenbreugel. DIALOG: An Expert Debugging System for MOS VLSI Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-4(3):303–311, July 1985.

[15] H. DeMan, I. Bolsens, and E. van der Meersh. An Expert System for Logical and Electrical Debugging of MOSVLSI Networks. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 203–205, Santa Clara, California, November 1984.

[16] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:495–516, 1979.

[17] E. B. Eichelberger and T. W. Williams. A Logic Design Structure for LSI Testability. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 462–468, New Orleans, Louisiana, June 1977.

[18] Joseph Faletti and Robert Wilenski. The Implementation of PEARL. Technical report, UC Berkeley Department of Electrical Engineering and Computer Sciences, Computer Science Division, Berkeley, California, March 1982.

[19] Charles L. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, July 1981.

[20] Charles L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[21] Charles L. Forgy. *The OPS83 Report System Version 2*. Production Systems Technologies, Pittsburgh, Pennsylvania, 1984.

[22] Jim Gettys. X Version 10 Protocol Guide. Technical report, Massachusetts Institute of Technology, Cambridge, Mass., 1986.

[23] H. C. Godoy, G. B. Franklin, and P. S. Bottorff. Automatic Checking of Logic Design Structures for Compliance with Testability Ground Rules. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 469–478, New Orleans, Louisiana, June 1977.

[24] N. Goncalves and H. DeMan. NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures. *IEEE Journal of Solid-State Circuits*, SC-18(3):261–266, June 1983.

[25] Steven Greenberg and Mahmud Buazza. Logic Recognition in the SAVVY Timing Verification System. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 269–271, Santa Clara, California, November 1984.

[26] Gordon Hamachi, Robert Mayo, John Ousterhout, Walter Scott, and George Taylor. 1985 VLSI Tools: More Works by the Original Artists. Technical Report UCB/CSD 85/225, UC Berkeley Department of Electrical Engineering and Computer Sciences, Computer Science Division, Berkeley, California, 1985.

[27] David S. Harrison. VEM: Interactive Graphics for Oct. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 1989.

[28] David S. Harrison, Peter Moore, Rick L. Spickelmier, and A. Richard Newton. Data Management and Graphics Editing in the Berkeley Design Environment. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 20–24, Santa Clara, California, November 1986.

[29] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building Expert Systems*. Addison-Wesley, Reading, Massachusetts, 1983.

[30] Paul Heckel. *The Elements of Friendly Software Design*. Warner Books, New York, New York, 1984.

[31] W. R. Heller, W. F. Mikhail, and W. E. Donath. *Prediction of Wiring Space Requirements for LSI*, pages 117–144. Computer Science Press, 1978.

[32] Mark Hirsch and Daniel Siewiorek. Automatically Extracting Structure from a Logical Design. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 456–459, Santa Clara, California, November 1988.

[33] Mark Hofmann. Automated Synthesis of Multi-level Combinational Logic in CMOS Technology. Technical Report UCB/ERL M85/53, UC Berkeley Electronics Research Laboratory, Berkeley, California, July 1985.

[34] Rostam Joobbani. WEAVER: An Application of Knowledge-Based Expert Systems to Detailed Routing of VLSI Circuits. Technical Report CMUCAD-85-56, SRC-CMU Center for Computer-Aided Design, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1985.

[35] V. Kelly and L. Steinberg. The CRITTER System: Analyzing Digital Circuits by Propagation of Behavior and Specifications. In *The Proceedings of AAAI*, pages 284–289, Pittsburgh, Pennsylvania, August 1982.

[36] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[37] Jin H. Kim. Use of Domain Knowledge in Computer Aid for IC Cell Layout Design. Technical Report CMUCAD-85-57, SRC-CMU Center for Computer-Aided Design, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1985.

[38] P.W. Kollaritsch and N.H.E. Weste. A Rule-Based Symbolic Layout Expert. *VLSI Design*, pages 62–66, August 1984.

[39] T. Kowalski and D. Thomas. The VLSI Design Automation Assistant: Prototype System. In *The Proceedings of the ACM/IEEE Design Automation Conference*, Miami Beach, Florida, June 1983.

[40] R. Krambeck, C. Lee, and H. Law. High-Speed Compact Circuits with CMOS. *IEEE Journal of Solid-State Circuits*, SC-17(3):614–619, June 1982.

[41] Sandip Kundu. Design of Multioutput CMOS Combinational Logic Circuits for Robust Testability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-8(11):1222–1226, November 1989.

[42] Douglas Lanam, Pierre Huyn, Mike Lemon, Steven Rosenberg, and Reed Letsinger. Guide to the Heuristic Programming and Representation Language, Part 2: Rules. Technical Report AT-MEMO-84.1, Hewlett-Packard Computer Research Center, June 1984.

[43] Douglas Lanam, Reed Letsinger, Steven Rosenberg, Pierre Huyn, and Mike Lemon. Guide to the Heuristic Programming and Representation Language, Part 1: Frames. Technical Report AT-MEMO-83-3, Hewlett-Packard Computer Research Center, June 1984.

[44] M. Lebowitz. Experiments with Incremental Concept Formation: UNIMEM. *Machine Learning*, 2(2):103–138, 1987.

[45] Michael Lebowitz. Concept Learning in a Rich Input Domain: Generalization-Based Memory. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors,

Machine Learning - An Artificial Intelligence Approach, volume 2, chapter 8. Morgan Kaufmann, Los Altos, California, 1986.

[46] B. Lin and A. R. Newton. KAHLUA: A Hierarchical Circuit Disassembler. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 311–317, Miami Beach, Florida, June 1987.

[47] C. Lob. RUBICC: A Rule-Based Expert System for VLSI Integrated Circuit Critique. Technical Report UCB/ERL M84/80, UC Berkeley Electronics Research Laboratory, Berkeley, California, September 1984.

[48] C. Lob, Rick L. Spickelmier, and A. Richard Newton. Circuit Verification Using Rule-Based Expert Systems. In *The Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 881–884, Kyoto, Japan, June 1985.

[49] Wen-Jeng Lue and L. P. McNamee. Extracting Schematic-Like Information from CMOS Circuit Net-Lists. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 690–693, Las Vegas, Nevada, June 1989.

[50] Jean Christophe Marie, Olivier Coudert, and Jean Paul Billon. Automating the Diagnosis and the Rectification of Design Errors with PRIAM. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 30–33, Santa Clara, California, November 1989.

[51] K. Mayaram and D.O. Pederson. Circuit Simulation in LISP. Technical Report UCB/ERL M84/60, UC Berkeley Electronics Research Laboratory, Berkeley, California, August 1984.

[52] T. M. Mitchell, S. Mahadevan, and L. I. Steinberg. A Learning Apprentice for VLSI Design. Technical Report LCSR-TR-64, Laboratory for Computer Science Research, Rutgers University, January 1985.

[53] T. M. Mitchell, L. I. Steinberg, and J. S. Shulman. A Knowledge-Based Approach To Design. Technical Report LCSR-TR-65, Laboratory for Computer Science Research, Rutgers University, January 1985.

[54] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1(1):47–80, 1986.

[55] Sanjay Mittal and Clive L. Dym. Knowledge Acquisition from Multiple Experts. *AI Magazine*, 6(2), Summer 1985.

[56] A. Richard Newton and Alberto Sangiovanni-Vincentelli. Relaxation-Based Electrical Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-3(4):308–331, October 1984.

[57] Gordon S. Novak Jr,. GLISP User's Manual. Technical report, Stanford University Computer Science Department, Palo Alto, California, 1982.

[58] John Ousterhout. Magic: A VLSI Layout System. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 152–159, Albuquerque, New Mexico, 1984.

[59] C. J. Petrie, D. M. Russinoff, and D. D. Steiner. Proteus: A Default Reasoning Perspective. In *Proceedings 5th Generation Conference, National Institute for Software*, October 1986.

[60] Thomas L. Quarles. SPICE3 Version 3C1 Users Guide. Technical Report UCB/ERL M89/46, UC Berkeley Electronics Research Laboratory, Berkeley, California, April 1989.

[61] M. Shoji. FET Scaling in Domino CMOS Gates. In *The Proceedings of the IEEE International Symposium on Circuits and Systems*, Kyoto, Japan, June 1985.

[62] Rick Spickelmier, editor. *Oct Tools Distribution 3.0*. UC Berkeley Electronics Research Laboratory, Berkeley, California, March 1989.

[63] Rick L. Spickelmier. The Oct 2.0 Users Guide. Internal Report, UC Berkeley Electronics Research Laboratory, 1989.

[64] Rick L. Spickelmier and A. Richard Newton. A General Knowledge-Based Circuit Critic. *SIGART Newsletter*, (92):78–79, April 1985.

[65] Rick L. Spickelmier and A. Richard Newton. A Rule-Based Connectivity Verifier. *SIGART Newsletter*, (92):79–80, April 1985.

[66] Rick L. Spickelmier and A. Richard Newton. Connectivity Verification Using a Rule-Based Approach. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 190–192, Santa Clara, California, November 1985.

[67] Rick L. Spickelmier and A. Richard Newton. Critic: A Knowledge-Based Program for Critiquing Circuit Designs. In *The Proceedings of the IEEE International Conference on Computer Design*, pages 324–327, Rye Brook, New York, 1988.

[68] Robin Steele. An Expert System Application in Semicustom VLSI Design. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 679–686, Miami Beach, Florida, 1987.

[69] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, Massachusetts, 1984.

[70] Mark Stefik, Daniel G. Bobrow, Sanjay Mittal, and Lynn Conway. Knowledge Programming in LOOPS. *AI Magazine*, 4(3):3–13, Fall 1983.

[71] Warren Teitelman. Interlisp Reference Manual. Technical report, XEROX Palo Alto Research Center, October 1983.

[72] Ching-Farn E. Wu, Lionel M. Ni, and Anthony S. Wojcik. Functional Recognition of Static CMOS Circuits. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 306–309, Santa Clara, California, November 1987.

[73] R. Zippel and C. Clark. Schema: An Architecture for Knowledge Based CAD. Technical Report VLSI Memo 85-271, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, October 1985.

[74] Richard Zippel. An Expert System for VLSI Design. In *The Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 191–193, Newport Beach, California, May 1983.