ON THE INTERACTION OF FUNCTIONAL AND TIMING BEHAVIOUR OF COMBINATIONAL LOGIC CIRCUITS

by

Patrick C. McGeer

Memorandum No. UCB/ERL M89/137

11 December 1989

ON THE INTERACTION OF FUNCTIONAL AND TIMING BEHAVIOUR OF COMBINATIONAL LOGIC CIRCUITS

by

Patrick C. McGeer

Memorandum No. UCB/ERL M89/137

11 December 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

On the Interaction of Functional and Timing Behaviour of Combinational Logic Circuits

Patrick C. McGeer

Ph.D. Computer Science Division

Abstract

We consider the elimination of false paths in combinational circuits. We demonstrate that static sensitization, the classic condition used to eliminate false paths, can incorrectly eliminate some true paths, leading to dangerous underestimates of the delay of a circuit. We introduce the property of *monotone speedup*, and argue that any correct false path procedure must not only accept all true paths but also satisfy this property. We then introduce the concept of *viable paths* and show that every true path is viable and that viability satisfies monotone speedup on symmetric networks. We show that any network may be transformed into a symmetric network while retaining the set of viable paths, and that this therefore gives us a correct false-path elimination procedure. We demonstate that determining whether a path is viable is equivalent to computing whether a logic function is satisfiable. We describe briefly a dynamic programming procedure to compute the longest viable path. We give a general approximation procedure and show that an algorithm due to Brand and Iyengar [10] is an approximation to the viability procedure. We give the single generic algorithm that is used by all authors to solve this problem, and demonstrate that it is parameterized by a boolean function called the *sensitization condition*. We give two criteria which we argue that a valid sensitization condition must meet, and introduce four conditions that have appeared in the recent literature, of which two meet the criteria and two do not. We then introduce a dynamic programming procedure for the tightest of these conditions, the *viability condition*, and discuss the integration of all four sensitization conditions in the LLAMA timing environment. We give results on the IWLS and ISCAS benchmark examples and on carry-bypass adders.

We then consider the special properties of precharge-unate circuits. We demonstrate that the only circuits which are hazard-free are those constructed using this technology. We then demonstrate that the *dynamic sensitization* criterion, a tighter criterion than viability, satisfies the monotone speedup property on such networks.

R.K. Brayton

Prof. Robert K. Brayton
Thesis Committee Chairman

Acknowledgements

For some obscure reason, acknowledgements in a Ph. D. thesis always tend to be bland lists of names, starting with the advisor, careening through the other members of the committee, continuing briskly with faculty-members-who-have-had-an-influence-on-my-career, finally tailing off with a sort of grab bag of colleagues, friends, landladies, lovers, and pets, and generally coming to a grinding halt with a mushy tribute to Mom.

Now, I don't object to any of this. Those names *should* all be there, and (in general) more, besides. But a mere recital hardly does justice to those being acknowledged. This section is an opportunity to leave a rollicking goodbye and valentine to those who have made one's graduate career a pleasant time. Further, after 200 pages of bland technical writing, I feel a need to let the rhetorical juices flow. It will take a couple of pages more, but what the dickens. My thesis is done, school's out for summer and I'm in a good mood. Finally, after eight years of graduate school at two Universities with – my goodness – five different research advisers – I have a great many thank-yous to give.

My first thanks go to my friend, mentor, and research advisor, Robert K. Brayton. The work reported herein is very much our joint effort, and it seems to me – as I look down the road to the research I'll be doing over the next few years – that the seeds of all of it were planted in conversations with him. Our collaboration has been enormously fruitful, and, more than that, tremendously pleasant. As we walked down this trail of discovery together, Bob was always eagerly looking ahead, shining a flashlight down interesting side paths, occasionally exploring some – and was always delighted to look at any rock I'd dug up, checking to see if some nugget of knowledge was contained therein. More than any mathematics or logic synthesis (though I picked up plenty of both from Bob), Bob gave me two things which I shall always treasure: the courage to follow that trail, wherever it leads, and to proclaim boldly when it takes an unexpected twist, rather than simply hacking a path in the expected direction; and, further, to revel in the joy of discovery and the thrill of science. Too often all of us get caught in the importance of career and fame, and we

forget that what's really important is that unexpected vista that science affords to anyone with the time to look. In sum, these past two years – has it really been so long? – have been whacking great fun. I shall miss them, and shall always treasure the memory.

Alberto L. Sangiovanni-Vincentelli is much more than the second reader of this thesis. It was Alberto who kept me going in the dark days before Bob arrived at Berkeley, persuading me, on the many, many occasions when I felt like trotting off to the Valley, that I was able to do good research and that I should continue at Berkeley (Believe me, when you've been a doctoral student for four years with no smell of a thesis, the Valley starts to look awfully good). Further, Alberto recruited and led the really remarkable team I've been privileged to work with over the past two years (more on that below): if it weren't for him, there are a great many of us who would be doing other, duller, and less productive things. I am grateful to Alberto for many things, but one more deserves special mention. When I started in cadgroup, I gave the world's worst seminars. Through many long, patient hours Alberto shaped me into an acceptable speaker. I shall never hear the word "turkey" again without thinking fondly of him. Intensely loyal, terribly generous, scrupulously honest: I am deeply honoured to count him my friend.

It had been a number of years between math classes when I took graduate topology from Professor Henry Helson, my third reader. I couldn't have picked a better time, or a better professor. Computer scientists don't get enough practice on proof techniques in general, and I was terribly rusty: I needed practice, especially inasmuch as I was about to embark on some research that required some fairly nontrivial constructions. That the results came out is due in no small part to the techniques that Prof. Helson taught me that year.

Though not on my thesis committee, Professors Donald O. Pederson and A. Richard Newton have been a constant source of help and inspiration. Prof. Pederson kindly served on my qualifying exam committee. As for Prof. Newton, it seems to me that some of the most illuminating conversations I've had in graduate school have come at midnight in the lab with Richard. On those too-rare occasions when he'd pop by to talk about intelligent televisions, or the world as a global network, or

engineering design environments, I'd remember that engineering is not fundamentally about mathematics or electronics, but rather about creating miracles so that we may all live longer, healthier, more fulfilling lives. I shall miss those conversations, too. Richard always showed keen interest in this research, and encouraged Bob and I to publish this material.

Here I should say something about the Computer-Aided Design research group at UC-Berkeley, one of the Western world's finer institutions (the group, not the University – oh, well, maybe both of them make it...). In the time I've been here, cadgroup has been home to over 60 graduate students and industrial visitors, working on everything from behavioural synthesis to compaction to device simulation to analog CAD, and a great deal (including me) in between. Of course, my closest colleagues were in the logic synthesis group, and I'd like to say a particular thanks to Rick Rudell, Albert Wang, Sharad Malik, Alex Saldanha, KJ Singh, Tiziano Villa, Ellen Sentovich, Hamid Savoj, Cho Moon, Luciano Lavagno, Pranav Ashar, Herve Touati, Yosi Watanabe, and Abdul Malik, with whom I shared that special elan that comes to mark an active research team. At times it seemed that every day brought a new discovery; each of you contributed to making these swift years a remarkable and exciting time. Chuck Kring, Tom Laidig, Wendell Baker, Brian Lee, Peter Moore, Bill Bush, Antony Ng and Brian O'Krafka were always ready to talk about any number of interesting and diverting topics, and broadened my education beyond mathematics. And thanks to Ruth Brayton for being a friend to all of us.

Cadgroup is largely a student-run group; almost all the software we use is unsupported, kept going by one or more of us – this includes the document processing software that I'm using to write this thesis. There aren't any rewards for doing this, aside from thanks in a thesis acknowledgement, and it's a great deal of effort. So a big thanks to Rick Spickelmeier, Tom Quarles, Tom Laidig, Dave Harrison, Rick Rudell, Beorn Johnson, Brian Lee, Chuck Kring and Don Webber for pitching in and maintaining the software.

Cadgroup isn't entirely student-supported. Brad Krebs (somehow) keeps an installation of three mainframe DEC products (two Vaxen and one MIPS-based mini) and over fifty workstations up and running. It's a nontrivial task, and he handles it with good humour, grace and a couple of part-time student assistants. So thanks, Brad, to you and your staff – Mike Kiernan, Valerie Walker, and Kurt Pires.

The grants we're on have a never-ending stream of paperwork, which keeps our clerical and administrative staff busy – but never too busy to help a graduate student out who's absolutely, positively gotta get a paper onto the Fed X truck today – so thanks to Shelly Sprandel, Flora Oviedo, Irena Stanczyk-Ng, Maria Delgado-Braun, Erika Buky, Susie Reynolds, Sherry Parrish and Deirdre McAuliffe-Bauer.

The other cadgroup students have been a constant source of inspiration and friendship. Ciao, aloha, and see you soon to Bill Lin, Giorgio Casinovi, Dave Reilly, Carl Sechen, Hyunchul Shin, Seung Ho Hwang, Lorraine Layer, Wayne Christopher, Tony Ma, Srinivas Devadas, Young Kim, Jeff Burns, Tammy Huang, Ken Kundert, Karti Mayaram, Theo Kelessoglou, Alan Kramer, Greg Sorkin, Linda Milor, Umakanta Choudhury, Jaijeet Roychowdhury, Andrea Casotto, Mark Beardslee, Mitch Igusa, Peter Simanyi, Arlindo Olivieria, Narendra Shenoy, Rajeev Murgai, Abhijit Ghosh, Chris Lennard, Eric Tomacruz, Charmine Tung, Gregg Whitcomb, Fabio Romeo, and Yoshi Nishizaki.

We've had industrial visitors, too, and their interaction has helped all of us enormously. I am especially indebted to Kurt Keutzer of AT & T Bell Labs, Gary Gannot of Intel and Ewald Detjens of Exemplar Logic.

Berkeley is the world's worst, most unforgiving bureaucracy – absolutely rigid, dreadfully incompetent, missing even the leavening influence of corruption. And I'm terrible with my paperwork. The only reason I made it through was due to various Saints in Human Form, the principal one being Kat' Crabtree. Thanks, Kat'. Inadequate, that, but you know what I mean. While I'm on the subject of saints in human form, let's not forget Genevieve Thiebault and professors Mike Lieberman and Dave Patterson.

I got into CAD on Prof. Al Despain's ASP project: Al got me started in this business, and was kind enough to let me go elsewhere when it became clear that my research had diverged from his area of expertise. The very, very best, Al. I shall always treasure my association with you and with the other members of the ASP team: Bill Bush, Jon Pincus, and Gino Cheng.

I came to Berkeley to work on a symbolic computation project with Prof. Richard Fateman. If it weren't for him I wouldn't be here, and I learned a great deal from him and from the other students on that project: Andy Lazarus, Clifton Williamson, Carolyn Smith, Dave Barton, John Foderaro and Neil Soiffer; and, lest we forget, Phil Colella, for reminding me that making a computer sit up and speak isn't nearly as important as making it say something interesting.

Then there were the friends who didn't fit into any project that I did here, but provided help, guidance, and a great deal of friendship, and continue to do so. Gaetano Borriello, Bill Bush and David Wood have superb research judgement and a thorough knowledge of how academia and systems research works – and were always ready to lend an ear and a word of advice. Carl Ponder was always ready to listen to an idea and offer constructive criticism. And as for Antony Ng: well, I probably learned more over innumerable lunches and coffees talking to Antony than in the rest of my career put together. And Antony has a mean overhead – I learned, painfully, never to float a lob near the forehand wall. Also, let me not forget Scott Baden, Ben Zorn, Doug Marinaro, Dain Samples, Brent Welch, Fred Douglis, Joan Pendleton, Bob Mayo, Walter Scott, Barry Fagin, Peter Van Roy and Tep Dobry.

I've had tremendous help from colleagues at other institutions, in particular Mike Lightner of Colorado and Louise Trevillyan of IBM TJ Watson Research Center.

Thanks, too, to Prof. Randy Katz, for interesting me in VLSI by teaching a fascinating Mead & Conway course and for constant help and advice thereafter; to Prof. Carlo Sequin for pointing out which fields still had interesting research problems; to Prof. John Ousterhout for an admonishment to stay broad and to Prof. Paul Hilfinger for advice. And thanks to Prof. Velvel Kahan for a head-shrinking lecture when I needed it.

I'd like to thank my Master's supervisor, Prof. Keith Geddes of Waterloo, and his colleagues Prof. Bruce Char and Prof. Gaston Gonnet. In a life full of happy accidents, the Maple project was one of life's better breaks.

Thanks to Prof. Ron Harrop, Prof. Subrata Dasgupta and Prof. Ted Edwards for inspiring me as an undergraduate; and thanks to my first CS professor, soon to be my colleague, Prof. David Kirkpatrick. If CMPT 105 at SFU had been lousy...I might be an economist today. I'm not sure whether that's me or Wall Street shuddering.

And thanks to my father and mother, who taught me by word and example that the most important thing in life is to do something productive, and that science is certainly productive. Perhaps if I'd fetched fewer live brains as a teenager for you two I wouldn't have started working on brains made out of sand. And thanks to my father for talking me out of going into physics, and to both of you for constant support and advice.

Thanks to Medonte and Elora, Hobbes and Mustang, Danielle, Bruiser, Luffra and Tucker, for reminding me about what's really important in life.

For Karen: the best poets in the English language have tried to capture what I have to say succinctly, and they haven't got it right. And a list of everything you've done over the past eight years would quintuple the length of this book, which is already too long. So...

Thanks to Karen for absolutely everything.

Money. First rule: *always thank people for money.* Thanks to the Digital Equipment Corporation for the hardware I'm currently typing on, and to the Semiconductor Research Corporation for supporting me throughout this thesis. Thanks to the other sponsors I've had throughout a chequered graduate student career: the Defense Advanced Research Projects Agency, the Naval Space Warfare Command, the Office of Naval Research, the Department of Energy, the System Development Foundation and the Natural Sciences and Engineering Research Council of Canada.

There. We're done. The whole list is there, save the landladies, but I think you'll agree it's more entertaining my way. The last words are not mine: Roger Zelazny finished off a novel this way, and I admired it...

Goodbye and Hello, as always.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The two classic parameters of integrated circuit design are speed and area. The cost of an integrated circuit is linearly related to the *yield* (that is, to the percentage of instances of the circuit which function correctly). In turn, yield is inversely related to the probability of a fatal defect in the material substrate, which is exponentially related to active area of the circuit. Hence, to a first approximation, the cost of an integrated circuit is a function of the area of the circuit.

Speed and its correct measurement affect both the performance and correctness of an integrated circuit. Performance goes without saying. Correctness follows from the observation that a circuit takes time to settle at a final value. Consider a generic integrated circuit: this consists of networks of combinational logic partitioned by storage elements called *latches* or *registers.* Such latches are typically controlled by a *load* line. When a load line is high, a latch changes state in response to changes on its input. During these periods, the latch is said to be *open.* When the latch is not responsive to changes on its input, the latch is said to be *closed.* If the load line is a clock line (as is typical in conventional designs), the circuit is said to be *synchronous.* Further, one can see that the effective value of the combinational network feeding a latch is the last value on the output of the network before the clock line goes low. Hence, if the correct value of the combinational network is to be computed in response to some input vector, the controlling clock must be high for long enough to permit the circuitry to arrive at a final value. This is called a *timing constraint* on or a *timing specification* of the circuit . A critical question concerning integrated circuits is whether they meet their timing specifications, and answering this question – and coercing circuits to meet their specification – is a major focus of research in computer-aided design.

Measurement of area is trivial at low levels of design, and is easily and accurately estimated at various higher levels through the use of abstract metrics which have been observed to correlate well with final layout area [1]. Speed – or, more precisely, delay – is far harder to measure. At the mask level, the circuit forms a network of transistors. Each transistor, when conducting or "on", acts as a resistor through which the gate on a succeeding transistor can charge or discharge, and so turn on and conduct. Analyses of this form yield a system of ordinary linear differential equations, which in turn may be solved by any number of numerical methods; in particular, the SPICE family of circuit simulators [62] [68] has enjoyed wide popularity over the last 15 years in performing this calculation. More recently, relaxation-based techniques such as RELAX [82] have been introduced to perform this calculation.

Circuit simulation techniques of this form are highly accurate, but have one drawback. Each signal contributes one differential equation to the system. Since circuits of 100,000 or so signals are fairly common, the computation task involved even for the most naive simulation technique (SPICE was originally a backward-Euler method) is herculean. Much recent work has addressed this problem through the use of specialized hardware or massively parallel computers [23] [79], with some success. However, in many CAD environments the use of hardware-intensive solutions is impractical, and software solutions are still much desired.

The software approach to this problem involves dealing with circuits at a higher level of abstraction. Conceptually, circuits may be thought of as networks of discrete components. These components may be arbitrarily large or small, though the utility of timing analyzers which work on large components is problematic, since the delay characterization of such components is usually fairly inaccurate. The most common abstraction is at the level of an atomic boolean function. With each such

[1]for example, logic synthesis tools estimate area by counting the number of literals which appear in the factored-form description of a circuit

component, or *gate*, a specific *delay* is associated. In this abstraction, both the waveforms and the static values associated with the various predecessors of the gate are ignored. The circuit is then isomorphic to a weighted, directed, graph, where the nodes of the graph are the gates of the circuit and the weights on the nodes are the delays of the gates. The delay of the circuit is simply the longest path in this graph. Finding this longest path is relatively easy; indeed, if the network is acyclic (as it is in the case of a combinational circuit), the algorithm to find the longest path is the well-known *topological sort* procedure [49], which is known to be $O(|V| + |E|)$. Programs of this sort are called *Timing Verifiers* or *Timing Analyzers.*

Timing Analysis is a good idea; and, like other good ideas, it has many parents. The idea of timing analysis dates as far back as the PERT project at IBM, and the original idea to use topological sort for the problem of timing analysis of logic circuits appears to have originated with Kirkpatrick and Clark [47]. Interest was renewed with the advent of the VLSI era in the early 1980's, and research focussed on two major areas. First, the computation of the delay associated with each discrete component (the so-called *delay model*) became a major topic of research; work on delay models was a central focus of the programs CRYSTAL [64] and TV [40]. CRYSTAL also broke circuits down not by logic gate, as was the common practice among timing analyzers, but into units called *stages.* A stage was defined as a path between the gate of a transistor or an output node and a single source. Second, the restriction to combinational (acyclic) circuits of boolean gates was thought too restrictive; both CRYSTAL [64] and TV [40] used event-driven simulators of the sort introduced by Bryant[19] in MOSSIM. In these programs, the transistors were explicitly modelled as bidirectional switches. Other innovations of the period included the introduction of *slacks* (differences between the time a signal was required and the time it arrived) by the TIMING ANALYZER [37].

Early timing analyzers were handicapped by poor delay models. Over the next several years, research continued into both scheduling procedures for non-combinational (i.e., cyclic) networks and into improved delay models. In 1984, Ousterhout [65] contrasted the accuracy of CRYSTAL under a *lumped* vs *slope* delay model. The lumped model (so-called because the capacitance is summed or lumped into a single large capacitor of value C which is presumed to discharge through a similarly-lumped resistor of resistance R, yielding a delay of RC) was shown to yield an error of 25% when compared to a SPICE simulation; using the delay models of Penfield, Rubinstein and Horowitz [70][67], in which a series of linear equations were derived for each delay[2], led to an estimate within 10% of the benchmark SPICE estimate. The relative accuracy of the latter model made the use of CRYSTAL and similar programs attractive for finding the relative ordering of paths in a circuit. Those paths found to be *critical*: those which took the longest to complete, or had the smallest slacks, or both – could subsequently be extracted and simulated in isolation, and the delay estimate refined. Later programs such as E-TV [46] took this approach to its logical conclusion, incorporating the relaxation-based circuit simulator ELOGIC [45] into the program and using the simulator to derive accurate values for the delays down the long paths.

Similarly, in 1987 Bauer, et. al, [5] introduced a new timing analyzer called SUPERCRYSTAL. SUPERCRYSTAL's two distinguishing features were, first, that the waveform over any capacitor in the circuit was approximated by a piecewise exponential waveform, and, second, that the effective resistance across a conducting transistor was determined by the voltage across the transistor, as opposed to being a single number given by the mean. In 1988, an improved version of SUPERCRYSTAL, renamed XPSIM, was announced [4]. XPSIM had been modified to explicitly simulate each stage using the approximate exponential function method [26] with a multirate time step. These improvements led XPSIM to demonstrate SPICE-level accuracy in a fraction of SPICE's runtime, making it suitable for use in timing analysis.

A difficulty with these efforts was that, in general, each timing verifier used either only a single delay model or a small set of models, which was in general only useful for one level of abstraction; timing verification was run at various levels of abstraction, each of which required a different model. In an effort at standardizing and parameterizing earlier work, Wallace and Sequin introduced an abstract version of a timing verifier, a program called ATV[77][78]. ATV's principle attraction was that

[2]Actually, the Penfield-Rubinstein model contained a logarithmic term as well as a linear term

a user could verify a design at varying levels of abstraction through the selection of parameters to Wallace's single, abstract, model. Further, since many existing models corresponded to a specific selection of such parameters, in some sense ATV represented many timing verifiers in one.

Though the use of accurate delay models has removed one source of systematic inaccuracy in timing verifiers, another remained. The purpose, after all, in discovering the delay down the longest path in a circuit is to determine how long a signal travelling down this path will take to reach the terminus. This information is irrelevant if no signal will travel down the circuit. This phenomenon is generically known as the *false path* problem.

The false path problem fundamentally arises because timing verification is *value-independent*; the states of the various wires into a node are ignored, and so presumed to always propagate the value of the preceding node on any path of interest. This is in contrast to simulators, which are value-dependent. Hence, in any *mixed-mode* simulation, where the critical path is identified by timing verifiers and whose length is determined to great accuracy by simulators, an essential problem is to find an input vector which exercises the long path identified by a timing analyzer. This is a particularly acute problem when one is using a simulator capable of simulating an entire circuit, such as XPSIM. If no such vector exists, then the path is said to be *false.*

We draw on the following observation in the analysis of the false path problem. Each node in a circuit can only propagate values from one of its inputs to its output if the other inputs are in a *sensitized* state; in the picture of a network of transistors, that the excitation of the transistor corresponding to the input must open a single conducting path from the output capacitor to ground (power). This forces the other transistors in the network to either unexcited or excited states; if one associates a boolean variable with the control on each transistor, it is easy to see that the set of such states represents a boolean function; this function is a function of the other inputs to the gate, called the *side inputs* to the gate. Indeed, if the excitation, or not, of the relevant transistor forces the output node to discharge or not, one can see that the boolean value represented by the output node is entirely determined by the value of the input control.

At a higher level of abstraction, if one views the circuit as a set of gates, the relevant states of the side inputs may be deduced from the logic function represented by the gate. In this sense, the false path problem is not merely a problem encountered in MOS VLSI designs but in all level-sensitive boolean logics; the scale and complexity of VLSI design makes the problem especially acute, however. Further, as we shall see below, the uncertainties of delay in integrated circuit design make the problem rather more rigid in this technology than in others.

The remainder of this chapter is organized as follows. In 1.1, we will discuss an abstract picture of a circuit and formulate the circuit timing analysis problem. In section 1.2, we will introduce some notation of modern logic synthesis which will aid in the analysis of the false path phenomenon. In section 1.3, we will formally define the false path problem. Finally, in section 1.4 we outline the remainder of the thesis.

1.1 Timing Analysis of Circuits

In this section, we formulate the timing analysis problem on circuits as a path-finding problem on weighted graphs. One can picture a circuit as a graph of nodes, each of which computes some function. The choice of node is arbitrary, and represents a trade-off between accuracy and efficiency. A convenient choice is the representation of a *transistor group*[4], which is a generalization of a boolean gate. A transistor group is a maximal collection of transistors and control connections such that, for every transistor in the group, its control connection lies outside the group. If there are no pass transistors or transmission gates in the design, this definition simplifies to that of a boolean gate. It is this definition that we adopt for the purposes of our discussion here. The edges of the circuit represent the interconnections of modules; since by construction each terminus of an edge represents the control of some transistor, and since signal flow is always to the control of a transistor, each edge in the graph is *directed.* If we assume that the circuit is combinational, as we do in this thesis, the circuit is further acyclic.

1.1.1 Delay Models

Once a network of nodes is chosen, delay is conventionally represented by weights on the nodes (or, equivalently, the edges) of the circuit. The derivation of these weights is called the *delay model* of the circuit. Delay models are variously derived, but basically break down into *static* and *dynamic* models. Static models have the property that the delay across each node is statically determined by the graph. The delay across a node is not then a property of the waveform emanating from an input. These are used in HUMMINGBIRD [80], and comprise the simplest (and, perhaps, the most commonly-used) of CRYSTAL's delay models. These can be represented by a graph with numeric weights on the edges.

Dynamic delays, on the other hand, are functions not only of the graph but also of the input waveforms. In general, timing analyzers using dynamic models compute for any node not only the delay across a node, or its arrival time, but also a waveform of the form $V = f(t, I)$, where V is the voltage across the node, t is time, I is an input waveform, and where f is a continuous, monotone function. The "delay" across the node is generally defined as $\{t|V(t) = T\}$, for some threshold value T. Dynamic models vary from very crude (CRYSTAL simply had a table of delays) to highly sophisticated (SUPERCRYSTAL used explicit simulation).

In general, dynamic models are more accurate than static models. Various refinements have been made to the basic static model to improve it. First, it was realized that CMOS gates are composed of dual networks of PMOS and NMOS transistors. Due to differences in electron mobility through the PMOS and NMOS transistors, and/or differences in the length of series chains through these networks, the effective resistance through the PMOS and NMOS sides can be unequal; this is reflected in unequal delays in transitioning the output node from 0 to 1 (the *pullup* transition) than in transitioning the output node from 1 to 0 (the *pulldown* transition). This is represented in the graph by assigning a pair of weights to each node, one in response to a *rising* edge, and one in response to a *falling* edge. This delay model may be thought of as an extremely crude waveform model.

Further enhancements to the static model are possible. In general, the delay response of a gate to one input may be different than that of another; the transistors corresponding to the inputs may be of different sizes, may be driven by differently-sized gates, be attached to nets of varying capacitance, or may appear in different positions in the transistor network that corresponds to the gate. Any of these factors may affect the delay across a node, and so it is natural to separate the delay across a node into delays across each input. This can be modelled by attaching the delays to the incoming edges of a node, not to the node itself[3]. Alternately, one can consider adding to each edge in the graph a node called a *static delay buffer* with the appropriate weight; in this way the delays across the edges can be modelled by delays across nodes in an isomorphic graph.

For convenience, when the theory and algorithms underlying the false path problem are developed in the sequel, a static delay model with one delay across each node is assumed. Nevertheless, the results hold for all static delay models unchanged through the isomorphisms developed above. We'll remind the reader of these and develop this theme more fully later.

1.1.2 Graph Theory Formulation

The static timing analysis problem is therefore to find the longest acyclic path in a weighted, directed graph. Consider the special case where the graph is acyclic. The sources of such a graph are called the *primary inputs* of a circuit. Some nodes, including all sinks of the graph, are designated as *primary outputs* of the circuit. We can transform the graph by attaching formal terminal output nodes to each primary output; such a transformation does not affect the timing properties of the circuit if the formal terminals have 0 weight, but permit us the convenience of treating the primary outputs and sinks as identical, so, for the remainder of this thesis, on such graphs the primary outputs are designated as the sinks. Each node n not a primary output has some set of successor nodes in the graph; these are called the *fanouts* of n, and are designated $FO(n)$. Similarly, each node n not a primary output has some set of predecessor nodes in the graph; these are called the *fanins* of n,

[3]This is TV's "dynamic" model

and are designated $FI(n)$. The transitive closure of FI is called the *transitive fanin* of n, and is denoted $TFI(n)$. The transitive closure of FO is called the *transitive fanout* of n, and is denoted $TFO(n)$. Each node n in the graph has a *level*, denoted $\delta(n)$. $\delta(n)$ is defined as follows:

$$\delta(n) = \begin{cases} 0 & n \text{ is a PI} \\ \max_{p \in FI(n)} \delta(p) + 1 & \text{otherwise} \end{cases}$$

Note that $\delta(n) > \delta(p) \ \forall p \in TFI(n)$ and $\delta(n) < \delta(p) \ \forall p \in TFO(n)$. The maximum level over all nodes in the graph is called the *diameter* of the graph, and is denoted D.

If the graph is acyclic and the weights are static, the longest-path problem is easily solved. The nodes are ordered by level by a very famous linear-time algorithm, topological sort [71] [49]. The maximum *distance* of a node n from a primary output $D_1(n)$ is thus defined:

$$D_1(n) = \begin{cases} 0 & n \text{ is a sink} \\ \max_{p \in FO(n)} D_1(p) + w(n) & \text{otherwise} \end{cases}$$

and the maximum distance of n from a primary input is defined:

$$D_2(n) = \begin{cases} 0 & n \text{ is a PI} \\ \max_{p \in FI(n)} D_2(p) + w(n) & \text{otherwise} \end{cases}$$

and a longest path is then a sequence of nodes, $\{f_0, ..., f_m\}$ such that

$$D_1(f_i) + D_2(f_i) - w(f_i) = K$$

where:

$$K = \max_{n \text{ is a node in the graph}} D_1(n)$$

If the graph contains cycles, then the problem of finding the longest path is $\mathcal{NP}$-hard [28], and in practice is solved by a technique called switch-level simulation.

Note that if $w(f) = 1$ for each f, we have $D_1(f) = \delta(f)$ for every f.

Since D_1 is the relation conventionally referred to as the delay d of a node, we also write $d(f)$ for $D_1(f)$ for a node f. Since the delay of a node is equal to the length of some path, we also write $d(\{f_0, ..., f_m\})$ for the length of a path.

Notation	Definition
$w(n)$	Weight (delay) of node n
$FI(n)$	fanins of node n
$FO(n)$	fanouts of node n
$TFI(n)$	Transitive closure of $FI(n)$
$TFO(n)$	Transitive closure of $FO(n)$
$\{f_0, ..., f_m\}$	path of nodes f_0 to f_m
$d(\{f_0, ..., f_m\})$	$\sum_{i=0}^{m} w(f_i)$
$\delta(n)$	Level of node n in the graph
D	Diameter of the graph (maximum level number)
$D_1(n)$	maximum distance of node n from a primary output
$D_2(n)$	maximum distance of node n from a primary input

Table 1.1: Basic Graph Notation

1.2 Logic Notation

In this section, we summarize basic switching theory notation. A summary appears in table 1.2 at the end of this section.

The most common method for representing a logic function is as a sum of terms, $f = t_1 + ... + t_n$, and is read $f = 1$ when $t_i = 1$ for some i. Each term is a product of *literals*; a literal is an ordered pair (v, p) where v is a boolean variable and $p \in \{0, 1\}$ is the *phase* of the variable. The term $t = (v_1, p_1)...(v_m, p_m) = 1$ whenever $v_i = p_i$ for every i. The notation (v, p) is thought too clumsy, so by abuse of notation the common notation for the positive phase $(v, 1)$ is simply v, and for the negative phase $(v, 0)$ it is simply $\overline{v}$.

$$f = xy\overline{z} + \overline{y}z$$

indicates that $f = 1$ whenever either $x = y = 1$ and $z = 0$ or when $y = 0$ and $z = 1$.

1.2.1 Cubes

In general, one can derive a geometric picture of the boolean n-space as an n-dimensional cube as follows. Consider the n-dimensional Cartesian co-ordinate system. Since each variable can only assume the values 0 and 1, the dimension of the space represented by the variable x can be restricted by planes at $x = 0$ and $x = 1$. Once this has been done in every dimension, the resulting object is an n-dimensional cube.

It is clear that a point in the Boolean n-space is a vertex of the cube; for historical reasons, a vertex is also called a minterm. Moreover, any subspace of the n space is simply another cube, albeit one of smaller dimension. Such a subspace corresponds to a specification of some variables of the space, and hence to a term. Terms are therefore generally referred to as *cubes*[13]. The *size* of a cube is therefore inversely proportional to the number of literals of the term; a term with m literals specifies a cube of size 2^{n-m}.

1.2.2 Cofactors

In much of the sequel we will be discussing the projections of boolean functions on a subspace; this corresponds to a partial evaluation of the function. If one views a function as a set of points on the n-cube (the set of points where $f = 1$), then the cofactor of a function with respect to a literal l, written f_l, is simply the collection of points on the $n - 1$ dimensional cube represented by the literal l. This can be either viewed as a function over this $n - 1$ dimensional space, or (simply by projecting each point on this space onto its neighbour on the $n - 1$ dimensional cube $\bar{l}$) as a function over the boolean $n-$space.

Since the subspace represented by the cube l_1l_2 is the same as that represented by l_2l_1, it is trivial to see that $(f_{l_1})_{l_2} = (f_{l_2})_{l_1}$, and, more generally, f_c is well-defined for any cube c.

Operationally, it is easy to take a cofactor of an m term function over the boolean $n-$space in time $O(nm)$, assuming some order on the variables. Further, the size of the cofactor of a function in the common function representations (disjunctive and conjunctive normal forms, factored forms, and boolean decision diagrams) is smaller than the size of the original function.

Cofactors derive their importance in logic synthesis due to the following theorem, which is variously credited to Shannon[73] and to Boole:

Theorem 1.2.1 (Shannon Cofactor Expansion) *For any boolean function f, any variable x,*

$$f = xf_x + \overline{x}f_{\overline{x}}$$

1.2.3 A Family of Operators

In this thesis, a pair of linear operators over the set of functions on the boolean $n-$ space will be extensively used.

One question that arises in the testing of networks and in the false path problem is the following. Given a function f, and a variable x, what are the assignments to the *remaining* variables such that the value of f is completely determined by the value of x, i.e., $f = x$ or $f = \overline{x}$ (i.e., f changes phase whenever x changes phase; in such a case, we say that f is *sensitized* by x)?

If $f = x$, from the cofactor expansion we must have that:

$$f_x = 1, \quad f_{\overline{x}} = 0$$

i.e., the logic function:

$$f_x\overline{f}_{\overline{x}}$$

must be satisfied. Similarly, if $f = \overline{x}$, we must have that:

$$f_{\overline{x}}\overline{f}_x$$

putting the cases together, we have:

$$\frac{\partial f}{\partial x} = f_x\overline{f}_{\overline{x}} + f_{\overline{x}}\overline{f}_x$$

or, more compactly:

$$\frac{\partial f}{\partial x} = f_x \oplus f_{\overline{x}} \tag{1.1}$$

$\frac{\partial f}{\partial x}$ is described in the testing literature [1] [72], and is referred to there as the *boolean difference.* From that literature, there is the following classic theorem, due to Sellers, et. al. [72].

Theorem 1.2.2 *A network N is testable for a stuck-at fault on node x through output f iff*

$$\frac{\partial f}{\partial x} \not\equiv 0$$

Further, Sellers and his co-workers proved a variety of properties on the boolean difference, which we give without proof here:

$$\begin{aligned} \frac{\partial^2 f}{\partial y \partial x} &= \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial f+g}{\partial x} &= \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x} \end{aligned}$$

Another question arises in considering the false path problem. Consider an arbitrary function f, an arbitrary variable x. Under which assignments of the other variables is there an assignment of x s.t. $f = 1$? [4]

Now, we must have $x = 1$ or $x = 0$. If xc is a satisfying assignment of f (i.e., $f(xc) = 1$) then, by the Shannon cofactor expansion, we can say that c is a satisfying assignment of f_x. Similarly, if $\overline{x}c_1$ is a satisfying assignment of f, then c_1 is a satisfying assignment of $f_{\overline{x}}$. Hence, if c is a cube such that either xc or $\overline{x}c$ is a satisfying assignment of f, then c satisfies $f_x + f_{\overline{x}}$. Hence we define the smoothing operator, $\mathcal{S}_x f$ as:

$$\mathcal{S}_x f = f_x + f_{\overline{x}} \tag{1.2}$$

and we have:

Theorem 1.2.3 *Let c be a minterm of $\mathcal{S}_x f$. Then either xc or $\overline{x}c$ is a minterm of f.*

[4]this is also called a *satisfying* assignment of f

Proof: The discussion above. ∎

Intuitively, we are interested in determining whether some function f is satisfiable, but we have no knowledge as to the value of the variable x. If the satisfiability of the function is dependent upon the value of x, then we clearly will be unable to get a precise answer to this question. Several questions that may be answered precisely arise, which are detailed in appendix B. One is worth detailing here.

Under what assignments of the other variables does there exist a value of x that gives rise to a satisfying assignment of f? Theorem 1.2.3 demonstrates that the answer to this question is the set of satisfying assignments of $\mathcal{S}_x f$. In fact, if we take the answer to the question "Does there exists a satisfying assignment of $\mathcal{S}_x f$?" as the answer to the question "Does there exist a satisfying assignment of f?", then this is an example of a *biased* satisfiability test. It is certainly the case that if there is no satisfying assignment of $\mathcal{S}_x f$, then there is no satisfying assignment of f. However, if x is not an independent variable, as is often the case, then there is a case where there is a satisfying assignment of $\mathcal{S}_x f$ but no satisfying assignment of f (consider the case where $f = xy$ and y implies $\overline{x}$). It is in this context that we will be using $\mathcal{S}_x f$: in the sequel we will be deriving a function that is satisfiable only if a path is true; since we want to reject only false paths, we want to bias this test positively, i.e., smooth out variables whose value is unknown and ask whether such a smoothed function is satisfiable.

Theorem 1.2.4 *Let f, g be any functions, x, y any variables. Then:*

$$\begin{array}{lrcl} (i) & \mathcal{S}_x\mathcal{S}_y f & = & \mathcal{S}_y\mathcal{S}_x f \\ (ii) & \mathcal{S}_x(f+g) & = & \mathcal{S}_x(f) + \mathcal{S}_x(g) \\ (iii) & \mathcal{S}_x f & \supseteq & f \\ (iv) & \mathcal{S}_x(fg) & \subseteq & \mathcal{S}_x(f)\mathcal{S}_x(g) \end{array}$$

Of these properties, we will be using (i) extensively.

Proof:

$$\begin{aligned}
(i)\quad \mathcal{S}_x\mathcal{S}_y f &= \mathcal{S}_x(f_y + f_{\bar{y}})\\
&= (f_y + f_{\bar{y}})_x + (f_y + f_{\bar{y}})_{\bar{x}}\\
&= f_{yx} + f_{\bar{y}x} + f_{y\bar{x}} + f_{\bar{y}\bar{x}}\\
&= f_{xy} + f_{\bar{x}y} + f_{x\bar{y}} + f_{\bar{x}\bar{y}}\\
&= ((f_x + f_{\bar{x}})_y + (f_x + f_{\bar{x}})_{\bar{y}})\\
&= \mathcal{S}_y(f_x + f_{\bar{x}})\\
&= \mathcal{S}_y\mathcal{S}_x f
\end{aligned}$$

$$\begin{aligned}
(ii)\quad \mathcal{S}_x(f+g) &= (f+g)_x + (f+g)_{\bar{x}}\\
&= f_x + g_x + f_{\bar{x}} + g_{\bar{x}}\\
&= \mathcal{S}_x(f) + \mathcal{S}_x(g)
\end{aligned}$$

$$\begin{aligned}
(iii)\quad \mathcal{S}_x f &= f_x + f_{\bar{x}}\\
&= xf_x + \bar{x}f_x + xf_{\bar{x}} + \bar{x}f_{\bar{x}}\\
&= f + \bar{x}f_x + xf_{\bar{x}}\\
&\supseteq f
\end{aligned}$$

$$\begin{aligned}
(iv)\quad \mathcal{S}_x(f)\mathcal{S}_x(g) &= (f_x + f_{\bar{x}})(g_x + g_{\bar{x}})\\
&= f_x g_x + f_{\bar{x}}g_{\bar{x}} + f_x g_{\bar{x}} + f_{\bar{x}}g_x\\
&= \mathcal{S}_x(fg) + f_x g_{\bar{x}} + f_{\bar{x}}g_x\\
&\supseteq \mathcal{S}_x(fg)
\end{aligned}$$

■

By induction on *(i)* we may write, for a set $U = \{x_1, .., x_n\}$ of variables

$$\mathcal{S}_{x_1}\mathcal{S}_{x_2}...\mathcal{S}_{x_n}f = \mathcal{S}_{x_1...x_n}f$$

or, more compactly, as $\mathcal{S}_U f$, and, by induction on *(ii)* and *(iii)*.

$$\begin{aligned}
\mathcal{S}_U(f+g) &= \mathcal{S}_U(f) + \mathcal{S}_U(g)\\
\mathcal{S}_U(fg) &\subseteq \mathcal{S}_U(f)\mathcal{S}_U(g)
\end{aligned}$$

Since the smoothing operator is thus implicitly defined for a set U, it is important to define its behaviour when the set $U = \emptyset$. We choose the obvious definition:

$$\mathcal{S}_\emptyset(f) = f$$

We have two important, though trivial, lemmas on the smoothing operator:

Lemma 1.2.1 *Let V be the set of inputs to a function f, and $U \subseteq V$. For any vector c of the primary inputs, let c_1 be the assignment of the variables in $V - U$ induced by c. Then $c \in \mathcal{S}_U f$ iff there exists some assignment c_2 of the variables in U such that $c_1 c_2$ is a satisfying assignment of f.*

Proof: The only if part is trivial, since $\mathcal{S}_U f \supseteq f$. If part. Let $c \in \mathcal{S}_U f$. Induction on $|U|$. If $U = \emptyset$, then $\mathcal{S}_U f = f$, and hence c satisfies f, i.e., the trivial assignment satisfies f. Suppose the statement holds for $|U| < N$. If $|U| = N$, let $U = W + \{x\}$. We can write:

$$\mathcal{S}_U f = \mathcal{S}_x \mathcal{S}_W f$$

We can write the left-hand-side as:

$$(\mathcal{S}_W f)_x + (\mathcal{S}_W f)_{\bar{x}}$$

Since c_1 satisfies $\mathcal{S}_U f$, it must satisfy at least one of $(\mathcal{S}_W f)_x$ and $(\mathcal{S}_W f)_{\bar{x}}$. If $(\mathcal{S}_W f)_x$, by induction, there is some vector c_3 satisfying $(\mathcal{S}_W f)_x$, and hence we set $c_2 = x c_3$ and done. Otherwise, there is some vector c_3 satisfying $(\mathcal{S}_W f)_{\bar{x}}$, and hence we set $c_2 = \bar{x} c_3$, and done. ■

Lemma 1.2.2 *Let U be any set of variables, and c be any cube where every variable in U is set to a value by c. Let $c^* = c - U$, i.e., the variables outside U set to a value by c. Then $(\mathcal{S}_U f)_c = (\mathcal{S}_U f)_{c^*}$.*

Proof: If g is independent of any variable x, then $g_x = g_{\bar{x}} = g$. It follows inductively that if g is independent of the variables set in a cube c, then $g_c = g$. Since $\mathcal{S}_U f$ is a function independent of all the variables in U, it follows that $(\mathcal{S}_U f)_c = \mathcal{S}_U f$. ■

The smoothing operator and the boolean difference are part of a more general family of such operators, where f_x and $f_{\bar{x}}$ are combined in various ways. For the sake of completeness we detail them in appendix B.

Notation	Definition
$x(\overline{x})$	Literal representing the value $x = 1$ $(x = 0)$
Cube c	product of literals
f_c	Evaluation of f on the subspace represented by cube c
$\frac{\partial f}{\partial x}$	Boolean Difference of f wrt x $(f_x \oplus f_{\overline{x}})$
$\mathcal{S}_x f$	$f_x + f_{\overline{x}}$
$\mathcal{S}_U f$	$\mathcal{S}_{U-\{x\}}(f_x) + \mathcal{S}_{U-\{x\}}(f_{\overline{x}})$

Table 1.2: Basic Logic Notation

1.3 The General False Path Problem

Timing verifiers are typically quite fast; indeed, for fully-restoring combinational logic the problem is simply that of finding the longest path through a directed acyclic graph, which is well known to be $O(|V| + |E|)$. However, these programs will always identify the longest path as the critical path of the circuit. This path, however, is not the path of real interest: the path of interest is the longest path down which a signal can propagate. Paths down which no signal can propagate are called *false paths*, and the problem of identifying them, and so finding the longest true path through the circuit, is known as the *false path problem.*

Consider, for example, the circuit in figure 1.1. For x to propagate to a, we must have $y = 1$. For a to propagate to b, we must have $z = 1$. But for b to propagate to c, we must have $y = z = 0$. Hence the path $\{x, a, b, c, d\}$ *appears* to be *false.*

The false path problem has been known for some time. The earliest complete discussion in the literature appears to be due to Hrapcenko [39] [5]. Hrapcenko demonstrated that, for every integer n, there exists a logic function for which the actual delay of the minimal network is $n + 8$ but for which the longest path is $2n + 8$.

[5] Hrapcenko's manuscript was kindly brought to the attention of the author by Prof. N. Pippenger of the University of British Columbia

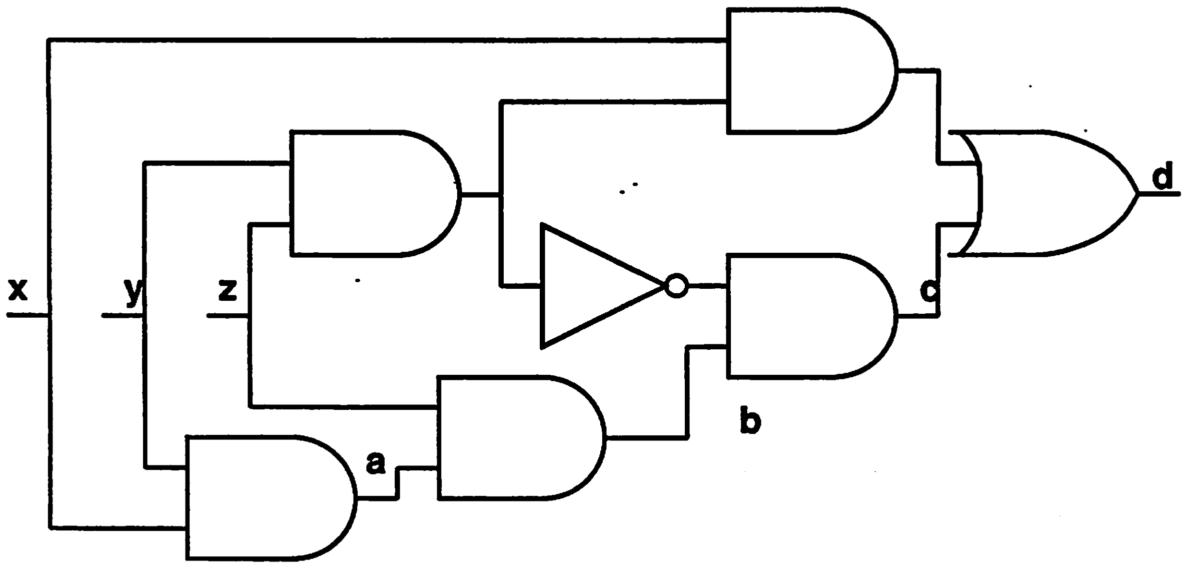


Figure 1.1: A False Path

Hrapcenko further observed that false paths arise naturally in the design of carry-acceleration adders, and suggested that the longest path through a carry-acceleration adder will be on the order of $2n$ nodes, while the delay will grow approximately as n. This observation correlates well with the experimental evidence of [6], and of this thesis.

Given the interest in accurate timing verification, considerable importance has been attached to the solution of the false path problem. Early facilities provided for this problem were largely user-oriented, either because the problem was felt to be intrinsically hard or because the authors of the software had an exaggerated respect for designers' intuition. These facilities fell into three major classes.

1.3.1 Explicit Recording of False Paths

Hitchcock's seminal timing analyzer TA [37] contained a facility, called **delay modifiers**, which indicated to TA that, *in the opinion of the designer*, certain paths would never be exercised. This approach was widely adopted; as recently as 1988, newly-reported timing analyzers included such a facility [20]. The difficulty, of course, is that a very large number of paths might eventually be indicated as false; further, there was also the unhappy possibility that the designer's intuition might fail him, though early writers evidently did not feel obliged to discuss this.

1.3.2 Case Analysis

If designer elimination of false paths was tedious, and involved an exhaustive enumeration of many separate paths, then perhaps examining the behaviour of the circuit under assumptions about the input will help. This general technique goes under the rubric of *case analysis*.

Ousterhout, in the description of his widely-used switch-level analyzer, CRYSTAL[66] describes the false path problem and this solution technique perhaps as well as anyone in the early literature:

> The value-independent approach is also responsible for the main problem in timing verification. When a timing verifier ignores specific values, it may report critical paths that can never occur under real operating conditions. These false paths tend to camouflage the real problem areas, and may be so numerous that it is computationally infeasible to process them all. In practice, all timing verifiers include a few mechanisms that designers can use to restrict the range of values considered by the program, usually by fixing certain nodes at certain values. This process is called *case analysis*; it is used to provide enough information to the timing verifier to eliminate false critical paths.

Case analysis was not a panacea, however:

> Case analysis must be used with caution. When the user specifies particular values, he restricts the timing verifier from considering certain possibilities; *this may cause critical paths to be overlooked* [italics mine]. Case analysis generally requires several different runs to be made, with

different values each run, in order to make sure that all possible states have been examined.

Case analysis was used in both CRYSTAL and in TV. The underlying assumption behind such case analysis is not stated explicitly in either Jouppi's or Ousterhout's work, but the discussions of both, taken together with the technological environments surrounding both projects, make the original motivation clear. Both CRYSTAL and TV were conceived and implemented in co-operation with groups designing VLSI microprocessors [6]. In such processors, which employ two or more non-overlapping (or *underlapped*) clock phases, the most glaringly obvious false paths are those which run through two transparent latches[7] which are active on opposite phases. Hence the original underlying assumption of case analysis was that the signals set to a value would remain constant during the period under evaluation, and, further, that the set of nodes whose values are fixed by the assignment of such constant values would have already been set to their assigned values and remain constant throughout evaluation. In other words, case analysis was never designed to account for transient effects during analysis; the false path of figure 1.1, for example, should not be detectable through case analysis, for none of the signals is constant throughout the evaluation period.

Nevertheless, case analysis *can* detect the false path in figure 1.1, by choosing $x = y = 1$; this is an abuse of case analysis, since it violates the underlying assumption of the analysis; namely, that the signal set to a value remains at that value for the entire period of analysis. Nevertheless, case analysis has probably been widely abused in precisely this fashion by designers since the introduction of timing verifiers. The consequences of this abuse we shall see later.

McWilliams, writing before Ousterhout, included case analysis in the SCALD timing verifier[61]. His justification for the use of timing analysis makes clear the underlying assumption that the signals specified by case analysis remain fixed during evaluation of the circuit; however, he permitted signals other than clocks to be specified for the purpose of case analysis:

[6]in Ousterhout's case, the RISC-II/SOAR/SPUR line of processors; in Jouppi's, the MIPS line
[7]A so-called *transparent* latch is a storage element which acts as a pass-through while it is active

> If the timing of the circuit never depended on the values of signals, but only on when they were changing or stable, the Timing Verifier would be relatively simple...The signals which are difficult to treat are those whose values affect the circuit timing, and which have different values during different clock cycles. ***For example, a control signal which determines whether a register is clocked during a given cycle affects whether the output of the register might change during that cycle.[italics mine]***

Such control signals generally remain stable throughout the clock period; the underlying assumption of case analysis is therefore that the signals set to values during analysis are presumed not to change throughout the period.

1.3.3 Directionality Tags on Pass Transistors

If one considers a network primarily made up of relay-like *pass transistors* (such as, for example, so-called *barrel shifters*), false paths arise from the fact that pass transistors, though nominally bidirectional, are in fact often unidirectional. There are a variety of ways that these can be handled. The CRYSTAL approach relied on user tagging of directionality. Jouppi's analyzer attempted to derive the signal flow direction through a series of rules; his experiments [41] suggest that upwards of 90% of transistors can be correctly categorized by such derivation. SUPERCRYSTAL attaches pass transistors to the nearest stage, and explicitly solves each such stage through the use of a circuit simulator. Cherry, in PEARL adopted both the TV and CRYSTAL approaches, using rules to automatically detect pass transistors; Cherry's ruleset was different from Jouppi's. Regarding the efficacy of the rule-based approach, Cherry notes:

> One circuit that these rules [8] are unable to cope with is a barrel shifter constructed with pass transistors. In general, the only way to determine signal flow in this type of circuit is to know what select nodes[9] are mutually exclusive.

In other words, the complete solution to this problem is derived directly from the solution to the problem outlined in figure 1.1. It is this problem – the problem of

[8]a la TV, but a non-identical ruleset

[9]i.e., which nodes on the gate terminal of the pass transistors

finding an automated solution to the false path problem – that is addressed in this thesis.

The general conclusion that one can draw from this discussion is that the timing analysis of an integrated circuit cannot be made accurately without considering the functional nature of the signals. Further, as we shall see later, the function computed by a signal cannot be accurately determined without taking the timing properties of the circuit into account. The analysis of their interaction is non-trivial, and must be done carefully and correctly in order to solve the false path problem.

1.4 Outline

The remainder of this thesis is organized as follows. In chapter 2, we outline the theoretical basis of a correct solution to the false path problem. The basic problem is that of determining when a path is *true*, or *sensitizable*, and is so referred to as a *sensitization criterion.* In the course of this analysis, we will demonstrate that a criterion which fails to take the *dynamic* nature of the signals in a circuit into account can lead to a timing analyzer which ignores the true critical paths of a circuit, and hence violates the basic *correctness* condition of a timing analyzer. Further, we argue that, since any delay model necessarily overestimates the delay across a node, that a timing analyzer must fulfill a *robustness* condition: namely, it must return an answer that is correct for *every* functionally and topologically identical circuit with identical or possibly lesser delays across individual nodes. We outline a criterion – the *viability* criterion – after demonstrating that the two most obvious criteria are either incorrect or non-robust, and prove that viability is both correct and robust. In chapter 3, we demonstrate that each program in the literature which purports to solve the false path problem is a variant on a single, parameterized algorithm, and that the sensitization criterion is but one of the parameters to this function. We demonstrate that to every criterion corresponds a logic function, and give the logic function both for one of the criteria rejected in chapter 2 and for the viability criterion. We then modify the generic algorithm to correctly compute the viability function. In chapter 4, we explore system considerations. We fully outline the parameter space: in addition to sensitization criterion, the others are the search method, the satisfiability test, and the function representation (the search method is fully developed in chapter 3). We give a general theorem of approximation, a weak – and hence correct and robust, but less tight – version of the viability function, and then demonstrate that two criteria which have appeared in the literature are approximations to weak viability, and so to viability. At an orthogonal axis of approximation, we discuss weak forms of satisfiability. We give experimental results on conjured circuits and on public benchmarks. In chapter 5, we consider hazard-free boolean functions, and show that these are isomorphic to the class of precharged-unate functions. In chapter 6 we show that dynamic sensitization – a sensitization criterion tighter than viability but non-robust on general circuits – is a correct and robust criterion on these circuits. Hence timing analysis on such circuits can yield tighter delay estimates than viability.

The appendices are organized as follows. In appendix A, we examine the complexity of the problem of finding the longest true path by various criteria, and demonstrate that each such problem is a member of the class of $\mathcal{NP}$-complete problems: loosely, the hardest problems whose solution may be verified in polynomial time. In appendix B, we review the family of operators of which the smoothing operator and the boolean difference are the most prominent members, and in appendix C we discuss a fast algorithm for a positively-biased SAT test. In appendix D we review the properties of precharge-unate logic gates.

Chapter 2

The False Path Problem

2.1 Introduction

In this chapter the false path problem is formally treated as a theoretical problem in combinational logic circuits. We begin by reviewing briefly the genesis and practical import of the problem.

Timing analysis and timing optimization of digital circuits is currently recognized as a key area. Optimization requires correct timing behavior, i.e. identification and accurate estimate of a circuit's true critical paths. Typically, critical paths are detected using static timing methods. While these methods are extremely fast, they often lead to serious overestimates of a circuit's delay due to *false paths*. A path is false if it cannot support the propagation of a switching event. In estimating the timing behavior of a circuit, we would like to find the slowest true path.

Several papers have appeared recently in which "false" paths are detected. The classic criterion, most often seen in user-supplied "case analysis" of "incompatible paths" in programs such as Crystal[64] and TV[40], is usually based on *static sensitization*. Under this criterion, a path is false if there exists no input condition such that all gates along the path are sensitized to the value of the previous gate on the path. This approach has recently been formalized in the SLOCOP timing environment[6].

Unfortunately, not every true path is statically sensitizable, so the delay of the longest statically sensitizable path is not necessarily an upper bound on the delay of the circuit. In this chapter we demonstrate that the use of static sensitization as a criterion for the truth or falsity of a path can lead to underestimates of circuit delay, possibly causing the circuit to behave incorrectly.

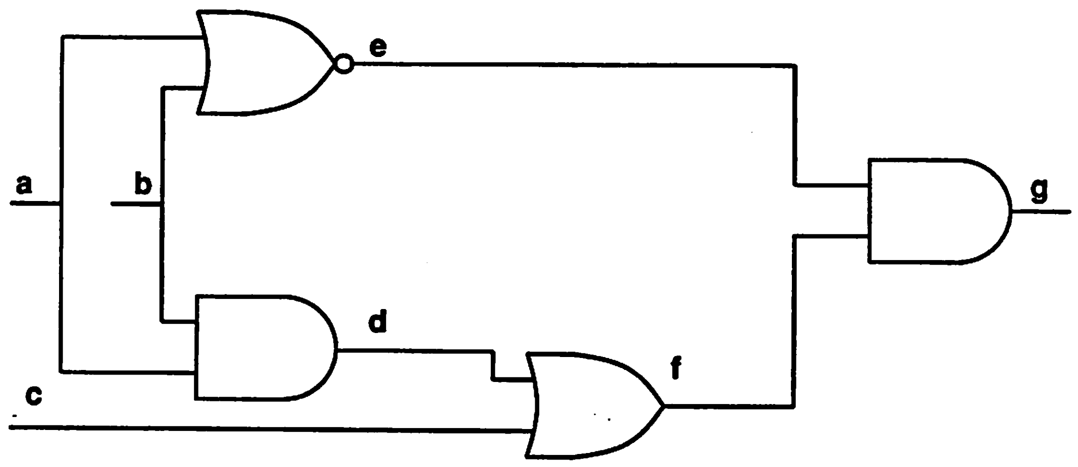


Figure 2.1: A Sensitizable "False" Path

Example: We illustrate this with a small example, taken from [10]. Consider the circuit shown in figure 2.1. Assume that all inputs arrive at $t = 0$, and that the delay on all gates is 1. Consider the path $\{a, d, f, g\}$, of length 3. For a to propagate to d we must have $b = 1$. For f to propagate to g we must have $e = 1$, which implies $a = b = 0$. Hence a static analysis would conclude that this path is false. Similarly, the path $\{b, d, f, g\}$ requires $a = 1$ and $a = 0$, and so a static analysis would conclude that this path is false. Since these are the only paths of length 3, a static analyzer concludes that the longest true path through this circuit is of length at most 2.

Nevertheless, one can see that holding c at 0 while toggling both a and b from 1 to 0 at $t = 0$ forces the output g to switch from high to low at $t = 3$. Hence one of the two paths of length 3 *must* be sensitizable, in the sense that a switching event can travel down it; further, if the clock delay is set to the value (2) of the longest *statically* sensitizable path, the circuit will behave incorrectly on this input. ∎

Our first task is the derivation of a criterion for sensitization such that the longest sensitizable path in the circuit is an upper bound for the delay of the circuit.

This task alone, however, is insufficient. Another problem must be dealt with by any algorithm which attempts to compute critical delay. The delay model used in timing analysis methods is a worst case model; it is intended to provide an upper bound for the delay of all circuits which may be manufactured and operated in particular environments. A real circuit is not the idealized circuit of timing models; it is a circuit with the same topology, but with possibly smaller delays at some of the nodes. Hence the estimate provided by the algorithm must hold for an entire *family* of circuits, the "slowest" of which – in the sense of having the slowest components – is typically the one under analysis. In order to use the slowest circuit it is necessary that any critical delay algorithm be ***robust*** in the following sense: if the delays on some or all gates in the network are reduced, then the critical delay estimate provided by the algorithm is not increased. When the algorithm is applied to the worst-case circuit, a robust criterion thus guarantees that the estimate obtained is valid for any circuit in the family. Colloquially, we refer to this robustness property as the ***monotone speedup*** property.

In this chapter, we develop a theory which correctly classifies true paths and can be used to provide a correct upper bound for the critical delay in all circuits with the same topology and with equal or less delay at each gate. In section 2.2, we develop our timing model, introduce the concept of event propagation and formally define sensitizable (or true) and critical paths. In section 2.3, the definition of a viable path is given and it is shown that every true path is viable. In section 2.4, it is shown that viability obeys monotone speedup on symmetric networks. In section 2.5, it is shown that every network may be transformed into a symmetric network, and retain all of its viable paths, and thus the longest viable path in the transformed circuit provides the upper bound we seek for the correct timing behaviour. In section 2.6, it is shown that determining the viability of a path is equivalent to computing a logic function.

Thus by finding the longest viable path in the symmetric worst-case delay circuit, we have the correct upper bound required. Of course, the longest path of a circuit also satisfies the above two criteria and so is a correct bound. However, we believe that the longest viable path is a tight bound, although we have not yet been able to show this. We have used our theory to demonstrate that the bounds given in [10] are correct; this theory also demonstrates that those bounds are looser than the bounds developed here. This construction is given in chapter 4.

2.2 Dynamic Timing Analysis

For purposes of clarity, we outline a very simple timing model here. The results of this chapter, however, do not depend on the precise characteristics of this model; we can show that they hold for slope delay models, models with separate rise and fall delays, and different delays on each pin.

Definition 2.2.1 *A* **path** *through a combinational circuit is a sequence of nodes, $\{g_0, ..., g_m\}$, such that the output of g_i is an input of g_{i+1}.*

Definition 2.2.2 *Each node g in a combinational circuit has a* **weight** *$w(g)$. The value of node g at time t is that determined by a static evaluation of the node using the values on its inputs at $t - w(g)$.*

Definition 2.2.3 *We define* **delay** *as follows:*

1. *The* **delay** *through a path $P = \{g_0, ..., g_m\}$ is defined as $d(P) = |\{g_0, ..., g_m\}| = \sum_{i=0}^{m} w(g_i)$. This is also called the* **length** *of the path.*

2. *The* **delay** *at a gate $d(f) = w(f) + max\{d(i) | i \in inputs(f)\}$ for all non-primary inputs f. For all primary inputs x, we define $d(x) = 0$. The weight of a primary input is 0.*

3. *When delays in more than one network are under consideration, the notation $d_N(f)$ denotes the delay at node f in network N.*

The restriction that the delay at the primary inputs is identically 0 is a notational convenience, and does not restrict the body of applicability of this theory.

Primary inputs which arrive at $t = T > 0$ can be modelled by assuming that the input arrives at $t = 0$ and that a static delay buffer of weight T is the sole fanout of the primary input. The buffer's fanout is the fanout of the primary input. Primary inputs that arrive ay $t = T < 0$ can be ignored by a simple translation of the time axis.

We assume that the wires of a circuit act as ideal capacitors; that is, once assigned a value the wire holds that value until changed by a computation at its source node. Further, for all negative values of t, the wires of the circuit hold the static values determined by some input vector c_1. Notationally, we capture this assumption by speaking of the value of function f at time t, $f(c_1, c_2, t)$, where c_1 is the input vector from $-\infty \leq t < 0$, and c_2 is the input vector from $0 \leq t \leq \infty$. Clearly we have:

$$f(\_, c_2, t) = f(c_2) \text{ for } t \geq d(f) \tag{2.1}$$

since after time $t = d(f)$, f has assumed its static (final) value.

As we develop the theory, we will use the concept of the "delay" of a function which is not explicitly computed in the network; specifically, of various functions which arise from the boolean difference. These functions do not have a delay within the model developed above. However, it is convenient to assign them a delay. The most reasonable choice is to assume that the computation of these functions is instantaneous. Hence:

Definition 2.2.4 *Let g be any node* not *in a network N. Then $w(g) = 0$.*

Hence, for any such node g,

$$d(g) = \max_{h \text{ is an input of } g} d(h)$$

Definition 2.2.5 *An* **event** *is the transition of a node from a value of 0 to 1, or vice-versa.*

We envision a sequence of events $\{e_0, ..., e_m\}$, each e_i occurring at node f_i, and each event e_i occurring as a direct consequence of event e_{i-1}. We say that event e_0 *propagates* down path $\{f_0, ..., f_m\}$.

Definition 2.2.6 *A path $P = \{f_0, ..., f_m\}$, f_0 a primary input, is* **sensitizable** *if some event e_0 may propagate down this path to the output f_m.*

Definition 2.2.7 *The* **critical path** *of a network is its longest sensitizable path.*

This permits us to consider the boolean conditions for a path to be sensitizable. Let event e_i be the transition of node f_i from 0 to 1. Event e_{i+1} is the transition of f_{i+1} from either 0 to 1 or 1 to 0. In the former case, we have that f_{i+1} tracks f_i, in the latter, f_{i+1} tracks $\overline{f_i}$. The conditions under which this is possible is a boolean function, the arguments of which we call *side inputs.*

Definition 2.2.8 *Let $P = \{f_0, ..., f_m\}$ be a path. The inputs to f_i that are* **not** *f_{i-1} are called the* **side inputs** *to P at f_i. We denote the set of side inputs as $S(f_i, P)$.*

Now, clearly we must have $\frac{\partial f_i}{\partial f_{i-1}} = 1$ when event e_{i-1} is propagated through f_i. We denote the time of event e_i, $t(e_i)$, as τ_i.

Lemma 2.2.1 *Let e_0 propagate down path $\{f_0, ..., f_m\}$, f_0 a primary input. Then $t(e_i) = \sum_{j=0}^{i} w(f_j)$.*

Proof: Induction on i. For $i = 0$, we have e_0 is a change in a primary input and this clearly occurs at $t = 0 = w(f_0)$. Assume for $i \leq j$. For $j + 1$, we have that e_{j+1} occurs as a direct consequence of e_j, whence $t(e_{j+1}) = t(e_j) + w(f_{j+1})$, hence $t(e_{j+1}) = \sum_{i=0}^{j+1} w(f_i)$. ■

Theorem 2.2.1 *A path $\{f_0, ..., f_m\}$, f_0 a primary input, is sensitizable iff $\exists$ input vectors $c_1, c_2 \ni \forall i \ \frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$.*

Proof:

$\Longrightarrow$ $\{f_0, ..., f_m\}$ is sensitizable. By lemma 2.2.1 we have that event e_i occurs at τ_i occurring as a result of the event at τ_{i-1} on f_{i-1}. This requires that either $f_i(c_1, c_2, \tau_i)$ tracks $f_{i-1}(c_1, c_2, \tau_{i-1})$, in which case we must have that $f_{i_{f_{i-1}}}$ is satisfied (so that f_{i-1} going high forces f_i high), and that $\overline{f_{i_{\overline{f_{i-1}}}}}$ is satisfied (so that f_{i-1} going low forces

f_i low). Further, the value on f_i at τ_i is statically determined by the values on its inputs at $\tau_i - w(f_i)$, i.e., at τ_{i-1}. This is summarized in the expression:

$$(f_{i f_{i-1}} \overline{f_{i \overline{f_{i-1}}}})(c_1, c_2, \tau_{i-1}) = 1,$$

The other case is that $f_i(c_1, c_2, \tau_i)$ tracks $\overline{f_{i-1}(c_1, c_2, \tau_{i-1})}$, in which case we must have that $f_{i\overline{f_{i-1}}}$ is satisfied (so that f_{i-1} going low forces f_i high), and that $\overline{f_{i f_{i-1}}}$ is satisfied (so that f_{i-1} going high forces f_i low). Further, these conditions must occur τ_{i-1}, as before. This is summarized in the expression:

$$(f_{i\overline{f_{i-1}}} \overline{f_{i f_{i-1}}})(c_1, c_2, \tau_{i-1}) = 1.$$

Putting the cases together, we must have $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$, as required.
$\Longleftarrow$ There exist input vectors $c_1, c_2 \ni \frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1 \forall i$. Therefore, for every i, we must have that either

$$(f_{i f_{i-1}} \overline{f_{i \overline{f_{i-1}}}})(c_1, c_2, \tau_{i-1}) = 1,$$

in which case $f_i(c_1, c_2, \tau_i) = f_{i-1}(c_1, c_2, \tau_{i-1})$ and the rising (falling) edge which is e_{i-1} is the rising (falling) edge as e_i at τ_i, whence the event e_0 propagates along the path, or

$$(f_{i\overline{f_{i-1}}} \overline{f_{i f_{i-1}}})(c_1, c_2, \tau_{i-1}) = 1,$$

in which case $f_i(c_1, c_2, \tau_i) = \overline{f_{i-1}(c_1, c_2, \tau_{i-1})}$ and the rising (falling) edge which is e_{i-1} is the falling (rising) edge as e_i, at τ_i, whence the event e_0 propagates along the path. These are the only two cases, and in either case e_0 propagates, whence $\{f_0, ..., f_m\}$ is sensitizable. ■[1]

The basic distinction between the current theory and the previous attempts may now be made clear. The previous theory required that $\frac{\partial f_i}{\partial f_{i-1}}(c_2) = \frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \infty) = 1$, a much stronger condition than $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$. Paths for which the former condition holds are called *statically sensitizable.* Paths for which the latter condition holds are called *dynamically sensitizable*, or (in light of theorem 2.2.1) *sensitizable.* It is easy to show that:

[1]In the case where the value of $\frac{\partial f_i}{\partial f_{i-1}}$ is changing at τ_{i-1}, to be conservative, we choose $\frac{\partial f_i}{\partial f_{i-1}} = 1$ at τ_{i-1}.

Theorem 2.2.2 *Every statically sensitizable path is sensitizable.*

Proof: A path is statically sensitizable iff, for all i, $\frac{\partial f_i}{\partial f_{i-1}}(c) = 1$ for some c. Clearly then $\frac{\partial f_i}{\partial f_{i-1}}(c, c, t) = 1 \; \forall t \geq 0$, giving the result. ■

Remark: This proof is obviously trivial in the sense that applying identical vectors will not generate any event to propagate. However, it is clear that if the cube c does not specify f_0, then one can obtain c_1 and c_2 by toggling the f_0 bit.
Remark: Note that the converse to this theorem is false: not all sensitizable paths

are statically sensitizable. Indeed, by appropriate adjustment of the internal delays it appears that one can make almost any path in any circuit sensitizable (of course, the timing characteristics of such adjusted circuits vary considerably). Moreover, one can demonstrate fully-testable circuits whose longest dynamically sensitizable path is not statically sensitizable; this statement demonstrates that static vs dynamic sensitizability can be an issue in the timing verification of non-contrived circuits. Indeed, in the circuit of figure 2.1, though the connections of both a and b to the AND gate are non-testable, and d is untestable for stuck-at-zero, the circuit is made fully testable through the addition of a second output, as shown in figure 2.2.

Algorithms which attempt to discover whether a given path is sensitizable must determine whether or not input vectors c_1, c_2 satisfying theorem 2.2.1 exist. There is a wide range of freedom permitted these vectors. However, we may say immediately:

Theorem 2.2.3 *Let $\{f_0, ..., f_m\}$ be a sensitizable path. If $d(\frac{\partial f_i}{\partial f_{i-1}}) \leq \tau_{i-1}$, for some i, then $\frac{\partial f_i}{\partial f_{i-1}}(c_2) = 1$*

Proof: Since $\{f_0, ..., f_m\}$ is sensitizable, $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$. But since $\tau_{i-1} \geq d(\frac{\partial f_i}{\partial f_{i-1}})$, we have $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = \frac{\partial f_i}{\partial f_{i-1}}(c_2)$ by equation 2.1, whence the result. ■

Corollary 2.2.4 *The longest path in a circuit is sensitizable iff it is statically sensitizable.*

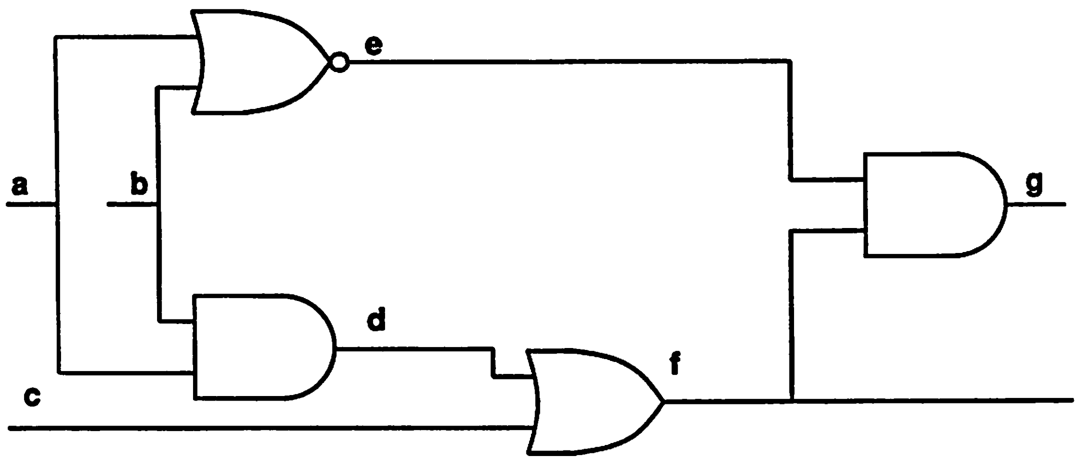


Figure 2.2: A Fully Testable Example

Proof: The if part is given by theorem 2.2.2. For the converse, observe that the premise of theorem 2.2.3 holds for every f_i on the longest path ■

Recall that every valid criterion must meet the monotone speedup property: if the delays on some or all gates in the network are reduced, then the critical delay estimate for the network is not increased. This guarantee cannot be given by the dynamic sensitization criterion, because the sensitizability of a path is inherently determined by the precise internal delays of the circuit. Hence, one can speed up a circuit and thus make a previously-unsensitizable path sensitizable. This path may be arbitrarily long (though not the longest in the circuit if such is unique); in particular, it may be longer than the longest-sensitizable path in the slower network.

An example which illustrates this phenomenon is detailed below.

Example: Consider the single-input circuit in figure 2.3. Assume the delay on all gates are as marked. Note when a is toggled from 1 to 0 at $t = 0$, from $t = 2$ to $t = 3$ there is a 0 on both u and w, so $x = 1$ from $t = 4$ to $t = 5$. However, in this case $y = 0$ throughout, so $out = 0$ throughout. Similarly, when a toggles from 0 to 1, from $t = 0$ to $t = 2$ there is a 1 on each input of y, so $y = 1$ from $t = 2$ to $t = 4$. However,

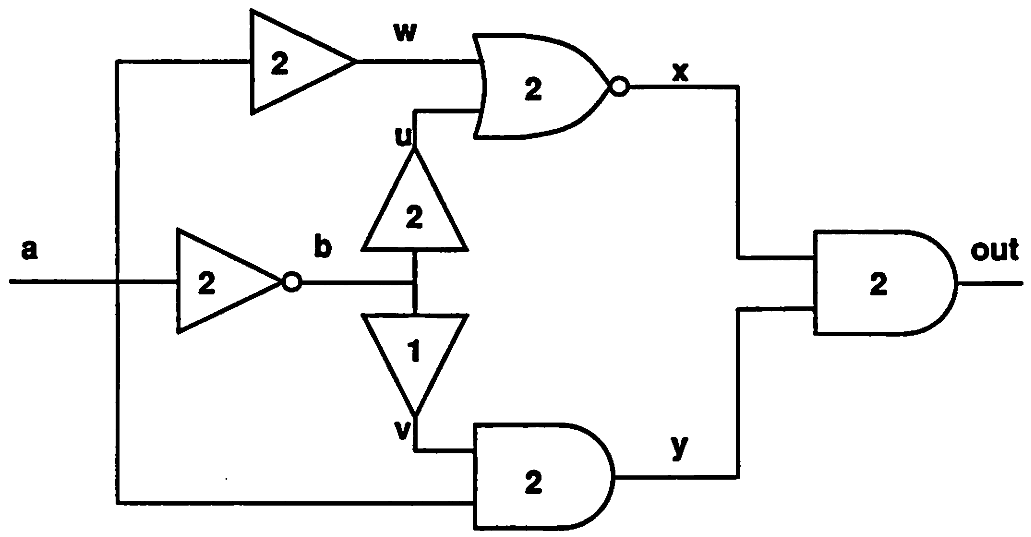


Figure 2.3: Monotone Speedup Failure

in this case $x = 0$ throughout, so *out* $= 0$ throughout. This circuit therefore has no dynamically sensitizable paths and its delay is 0.

If we now speed the circuit up by removing the delay buffer between b and u, so that u now arrives at $t = 1$, but all other delays are unchanged, when a is toggled from 0 to 1 we have a zero on each input to x from $t = 1$ (when u turns from 1 to 0) to $t = 2$ (when w turns from 0 to 1). Hence $x = 1$ from $t = 3$ to $t = 4$. But $y = 1$ from $t = 2$ to $t = 4$, so *out* $= 1$ from $t = 5$ to $t = 6$. Hence there is at least one dynamically sensitizable path in this circuit of length 6; by reducing the delay on the wire from the inverter to u from 2 to 0, we have increased the critical delay on this circuit from 0 to 6. A full timing diagram of the situation appears in figure 2.4. In this diagram, the solid lines represent the behaviour of the "slow" original circuit; the dotted lines the behaviour in the sped-up, or "fast" circuit. ∎

This phenomenon – that one can demonstrate circuits where the longest true path of a circuit increases length as components are sped up – appears to hold in

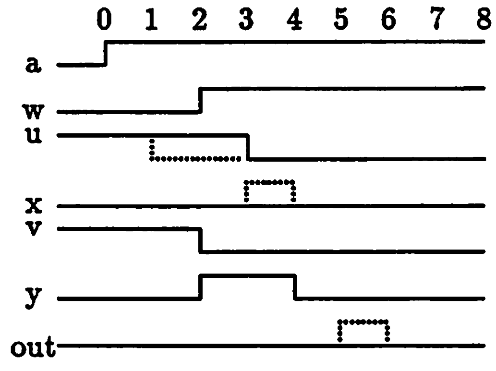


Figure 2.4: Timing Diagram of Monotone Speedup Failure

every level-sensitive logic where each wire holds its value until the value is changed. In fact, given that the exact delay times at the nodes in a circuit are only determined up to some given tolerance, the sensitizability of a path within a given circuit may vary between two "identical" but separate realizations. *Hence the longest sensitizable path appears to be an inherently nondeterminate property of logic circuits.*

2.3 Viable Paths

Since the longest sensitizable path does not satisfy monotone speedup, we cannot use this criterion to derive a correct upper bound on our family of circuits. We attempt to find a condition on circuits weaker than dynamic sensitization but one that is as strong as possible, certainly tighter than that given by a simple longest-path procedure. The condition C that we seek must possess two properties:

- Every sensitizable path must satisfy C
- C must satisfy the monotone speedup property; if network N' is obtained from N by reducing some or all delays, then the longest path satisfying C in N' must be no longer than the longest path satisfying C in N

One property that satisfies these constraints is simple longest path. However, this is too weak a condition, and we can do better. A strong property that satisfies these constraints is *viability*. Before we formally introduce the concept of viability, we wish to introduce its motivation.

Fundamentally, a node f_i is dynamically sensitized to an input f_{i-1} at τ_{i-1} but not statically sensitized to f_{i-1} only if the value of the function $\frac{\partial f_i}{\partial f_{i-1}}$ changes value at τ_{i-1} or later. This can only occur if there are events on some set of inputs to $\frac{\partial f_i}{\partial f_{i-1}}$ at or after τ_{i-1}; these are called *late side inputs*. Under these conditions, we may assume that each of these inputs are at *any* value at τ_{i-1}, and hence (to be conservative), we assume that they are set to any value which will propagate the event. Mathematically, we do this by "smoothing" the function $\frac{\partial f_i}{\partial f_{i-1}}$ over the late inputs (see 1.2).

Definition 2.3.1 *Consider a path $P = \{f_0, ..., f_m\}$. Q is said to be a* side path *of P at f_i if Q terminates in g, a side input to P at f_i.*

Definition 2.3.2 *A path $P = \{f_0, ..., f_m\}$ is said to be* viable *under an input cube c if, at each node f_i there exists a (possibly empty) set of side inputs $U = \{g_1, ..., g_n\}$ to P at f_i, such that, for each j,*

1. *g_j is the terminus of a path Q_j,*
2. *$d(Q_j) \geq \tau_{i-1}$ and Q_j is viable under c*
3. *$(S_U \frac{\partial f_i}{\partial f_{i-1}})(c) = 1$*

Intuitively, at each node we find the conditions which simultaneously permit a set of side inputs U (a subset of the *late side inputs*) to undergo events later than τ_{i-1}, and the remaining side inputs to statically sensitize the node. It is important to note that this can only occur if there is some assignment to the variables in U which statically sensitizes the node; the effect of the smoothing operator is to permit this assignment to be made, independent of conditions elsewhere in the network. Effectively, the variables in U are made independent variables by the smoothing operator. Note that the case $U = \emptyset$ corresponds to static sensitization.

The intuition behind smoothing off late side inputs may be grasped by considering the case where f_i is an AND gate, $f_i = f_{i-1}a_1...a_n$. In this case, $\frac{\partial f_i}{\partial f_{i-1}} = a_1...a_n$. If the inputs a_i and a_j are smoothed off the boolean difference, however, the resulting expression is

$$a_1...a_{i-1}a_{i+1}...a_{j-1}a_{j+1}...a_n$$

.

We demonstrate that the criterion of viability under a cube has the two properties we seek; namely, it is weaker than sensitizability, and it has the monotone speedup property. We do so by induction on the maximum distance (in nodes, not node weights) of a node n from the primary inputs, denoted $\delta(n)$, and called the *level* of n. Note for every node m in the transitive fanin of n, we have that $\delta(n) > \delta(m)$, and that the only nodes p for which $\delta(p) = 0$ are the primary inputs. Hence inductive proofs on $\delta(m)$ are really proofs on the structure of a graph; we will be showing that, given that a property holds for each node in the transitive fanin of some node, then it holds at that node.

We check our two conditions, first checking that every sensitizable path is viable.

Theorem 2.3.1 *Let $P = \{f_0, ..., f_m\}$ be a path. If P is sensitizable with $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1\ \forall i$, then P is viable under c_2.*

Proof: We prove by induction on $\delta(f_m)$. If f_m is a primary input then trivial. So suppose true for all paths terminating in some f_m such that $\delta(f_m) < L$. Now, consider a path terminating in f_m such that $\delta(f_m) = L$. Let $\frac{\partial f_i}{\partial f_{i-1}}(c_2) = 0$ for some $f_i\ i \leq m$. (if no such i exists, then the path is viable under c_2 by (1) of the definition, and done). Now, since $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$, and $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \infty) = 0$, there were events on inputs to $\frac{\partial f_i}{\partial f_{i-1}}$ at some $t \geq \tau_{i-1}$. The inputs to $\frac{\partial f_i}{\partial f_{i-1}}$ are the side inputs to P at f_i. Let $U = \{g_1, ..., g_n\}$ be the side inputs where the events occurred. Each event propagated under c_2 to g_j from some primary input h_{0j}, whence $Q_j = \{h_{0j}, h_{1j}, ..., g_j\}$ is a sensitizable path and $d(Q_j) \geq \tau_{i-1}$. Further, $\delta(g_j) < \delta(f_m) = L$, and hence by the inductive assumption Q_j is viable under c_2. Finally, the side inputs where no events occurred after τ_{i-1} is the set $S(f_i, P) - U$. These are precisely the inputs to $\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}$, whence we must have $\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}(c_2) = \mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \infty) = \mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1})$. Since $f(c) = 1 \Rightarrow \mathcal{S}_x f(c) = 1$, and since $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$, we have that $\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$, whence $\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}(c_2) = 1$ and done. ■

The converse to this theorem is false; not every viable path is sensitizable. Clearly the converse cannot hold since viability is robust and dynamic sensitization is not robust.

2.4 Symmetric Networks and Monotonicity

Viability does not possess the monotone speedup property on general networks; however, it *does* possess this property on networks composed of symmetric gates. The objective of this section is to prove this. The proof must be approached indirectly, for the set of viable paths changes as one changes the internal delays of the network. Hence the proof of the monotonicity theorem for symmetric networks is given by a construction: if N' is obtained from N by reducing some delays, and if P' is a viable path in N', then we construct a viable path P in N with $d_N(P) \geq d_{N'}(P')$. Thus N always contains a viable path at least as long as the longest viable path in N'. Having done this, in the sequel we shall show how to apply this result to networks containing asymmetric gates.

Definition 2.4.1 *A function f is said to be* **symmetric** *in some set of variables U if, for every permutation of U, there exists a phase assignment to the variables in U such that f is invariant.*

Example: f is symmetric in the variables x, y if one of the following holds:

$$\begin{aligned}
f(..., x, y, ...) &= f(..., y, x, ...) \\
f(..., x, y, ...) &= f(..., \overline{y}, x, ...) \\
f(..., x, y, ...) &= f(..., y, \overline{x}, ...) \\
f(..., x, y, ...) &= f(..., \overline{y}, \overline{x}, ...)
\end{aligned}$$

■

Example: $f(x,y) = x + \overline{y}$ is symmetric in x and y, since $f(\overline{y},\overline{x}) = \overline{y} + \overline{\overline{x}} = f(x,y)$

■

Lemma 2.4.1 *If f is symmetric in a set of variables U then for every $V \subseteq U$ where $|V| \geq 2$ and $x, y \in V$:*

$$\mathcal{S}_{V-\{y\}} \frac{\partial f}{\partial y} = \mathcal{S}_{V-\{x\}} \frac{\partial f}{\partial x}$$

Proof: Without loss of generality, assume that each variable is assigned the positive phase in the phase assignment of the symmetry definition. If $|U| = 2$, and since $f(\ldots,x,y,\ldots) = f(\ldots,y,x,\ldots)$, we have $f_x(y) = f_y(x)$, $f_{\overline{x}}(y) = f_{\overline{y}}(x)$, $\overline{f}_x(y) = \overline{f}_y(x)$, $\overline{f}_{\overline{x}}(y) = \overline{f}_{\overline{y}}(x)$, whence we have $\frac{\partial f}{\partial y}(x) = \frac{\partial f}{\partial x}(y)$ and so:

$$\mathcal{S}_x \frac{\partial f}{\partial y} = \mathcal{S}_y \frac{\partial f}{\partial x}$$

which gives us the result. Now Let $|U| = L$. Consider any $V \subset U$. We have:

$$\begin{aligned}
\mathcal{S}_{U-\{y\}} \frac{\partial f}{\partial y} &= \mathcal{S}_{U-\{x,y\}} \mathcal{S}_x \frac{\partial f}{\partial y} \\
&= \mathcal{S}_{U-\{x,y\}} \mathcal{S}_y \frac{\partial f}{\partial x} \text{ (from the base case)} \\
&= \mathcal{S}_{U-\{x\}} \frac{\partial f}{\partial x}
\end{aligned}$$

■

Symmetry is important because we can show monotone speedup for networks composed of symmetric nodes. Further, as is evident from the definition above, most networks are largely symmetric, and thus this theorem has some practical importance.

For proving monotone speedup, we need a technical lemma concerning the existence of viable paths when presented with a set of viable partial paths which conjoin. This is presented in the "viable fork" lemma. Reference to the diagram in figure 2.5 is helpful when analyzing this situation.

Lemma 2.4.2 (Viable Fork) *Let $V = \{g_0, \ldots, g_n\}$ be a subset of the inputs to some node h_0. Let each g_i be the terminus of a path P_i, a viable path under c. Let $c \subseteq \mathcal{S}_{V-\{g_i\}} \frac{\partial h_0}{\partial g_i}$ for each i. Let $Q = \{h_0, \ldots, h_p\}$ be any path, such that $c \subseteq \frac{\partial h_{j+1}}{\partial h_j}$ for every j. Then for some i, $\{P_i, Q\}$ is a viable path under c.*

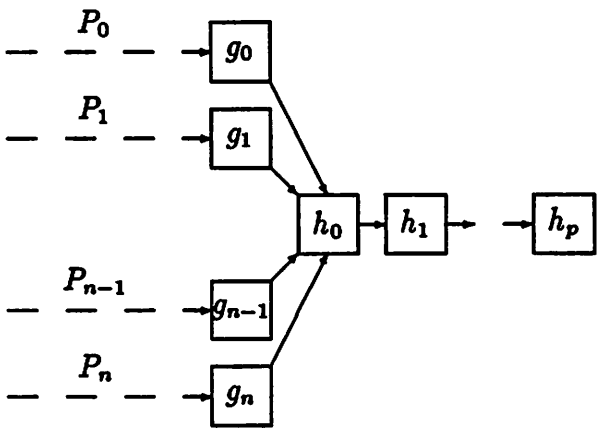


Figure 2.5: "Viable Fork" Lemma

Proof: Since $c \subseteq \frac{\partial h_{i+1}}{\partial h_i}$ for every i, all that must be shown is the viability under c of one of $\{P_i, h_0\}$. This is trivial if h_0 is statically sensitized for any of the g_i by c, so assume not. For some i, $d(P_i)$ is minimal among the P_i, and since by assumption $c \subseteq \mathcal{S}_{V-g_i}\frac{\partial h_0}{\partial g_i}$, and since for each g_j, P_j is viable under c with $d(P_j) \geq d(P_i)$ we have that $\{P_i, h_0\}$ is viable under c by the definition of viability. ■

In the monotone speedup theorem, we will be conjoining various partial paths which are known to be viable under some cube c onto the common "tail", in this lemma given by $\{h_1, ..., h_p\}$, and we will want to show that one of the resulting paths is viable under c, whence at least one path is viable under c.

Note that by lemma 2.4.1, for symmetric h_0, any V, $c \subseteq \mathcal{S}_{V-g_i}\frac{\partial h_0}{\partial g_i}$ for some $g_i \in V$ iff $c \subseteq \mathcal{S}_{V-g_i}\frac{\partial h_0}{\partial g_i}$ for each $g_i \in V$.

We now turn to the main theorem of this section, which demonstrates that viability has the monotone speedup property in symmetric networks. We proceed in this proof as follows: given a path P' viable under c in a "fast" network N', we demonstrate the existence of a slower path P in the "slow" network N viable under

c.[2]

In the proof, the diagram in figure 2.6 is helpful.

Theorem 2.4.1 *Let N' be any network obtained from a symmetric network N by reducing some internal delays. For every viable path $P' = \{f_0, ..., f_m\}$ in N', $\exists$ $P = \{k_0, ..., f_m\}$ a viable path in N with $d(P) \geq d(P')$.*

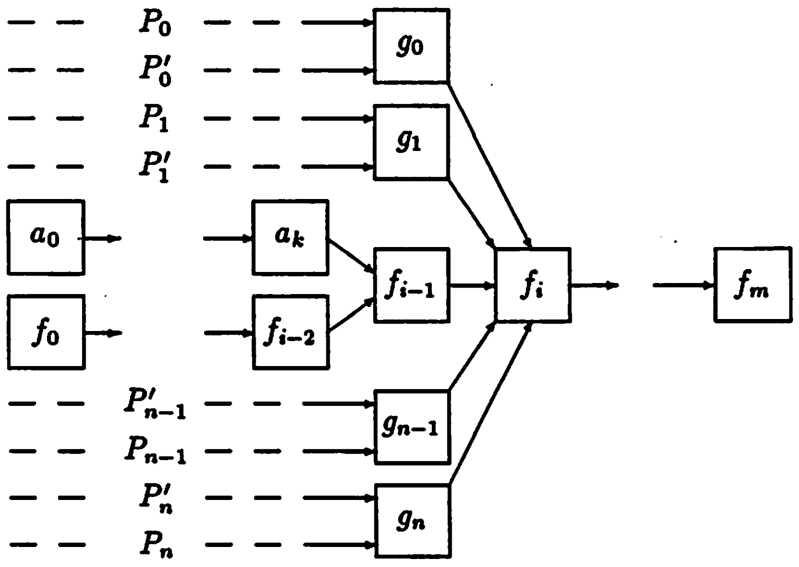


Figure 2.6: Viable Paths in N and N'

Proof: Let $\{f_0, ..., f_m\}$ be a viable path in N'. We proceed by induction on $\delta(f_m)$. The base case is trivial, so assume for $\delta(f_m) < L$. Consider the case $\delta(f_m) = L$. If $\{f_0, ..., f_m\}$ is statically sensitizable under c, then done, since this path is viable in every network in the family, and so in N. If not, let f_i be the last node that is not statically sensitized to f_{i-1} by c. Since $\frac{\partial f_i}{\partial f_{i-1}}$ is not satisfied by c, then since $\{f_0, ..., f_m\}$ is viable in N' by the definition of viability there exists a set $U = \{g_0, ..., g_n\}$

[2]By "fast" and "slow" here we mean that N' has been obtained by reducing some delays in N

of the side inputs such that for each g_j there is a P'_j viable under c in N' with $d_{N'}(P'_j) \geq d_{N'}(\{f_0, ..., f_{i-1}\})$ and with $c \subseteq S_U \frac{\partial f_i}{\partial f_{i-1}}$. Now, since $\delta(g_j) < L$, by the induction hypothesis for each $j \ \exists$ a P_j, viable under c in N and terminating in g_j, with $d_N(P_j) \geq d_{N'}(P'_j)$. Further, since $\{f_0, ..., f_{i-1}\}$ is viable under c in N', and $\delta(f_{i-1}) < L$, by induction there is a path $\{a_0, ..., a_k, f_{i-1}\}$, viable under c in N, with $d_N(\{a_0, ..., a_k, f_{i-1}\}) \geq d_{N'}(\{f_0, ..., f_{i-1}\})$. Since N is symmetric, and $c \subseteq S_U \frac{\partial f_i}{\partial f_{i-1}}$ the set of inputs $U \cup \{f_{i-1}\}$ satisfies the assumptions for the set V of lemma 2.4.2, with $\{h_0, ..., h_p\} = \{f_i, ..., f_m\}$. Therefore one of the $\{P_j, f_i, ..., f_m\}$is viable under c in N, or $\{a_0, ..., a_k, f_{i-1}, f_i, ..., f_m\}$ is viable under c in N.

$$\begin{aligned} d_N(\{a_0, ..., a_k, f_{i-1}, f_i, ..., f_m\}) &= d_N(\{a_0, ..., a_k, f_{i-1}\}) + d_N(\{f_i, ..., f_m\}) \\ &\geq d_{N'}(\{f_0, ..., f_{i-1}\}) + d_{N'}(\{f_i, ..., f_m\}) \\ &\geq d_{N'}(\{f_0, ..., f_{i-1}, f_i, ..., f_m\}) \end{aligned}$$

and:

$$\begin{aligned} d_N(\{P_j, f_i, ..., f_m\}) &\geq d_{N'}(P'_j) + d_N(\{f_i, ..., f_m\}) \\ &\geq d_{N'}(\{f_0, ..., f_{i-1}\}) + d_{N'}(\{f_i, ..., f_m\}) \\ &\geq d_{N'}(\{f_0, ..., f_{i-1}, f_i, ..., f_m\}) \end{aligned}$$

So all paths have greater delays in N than the path $\{f_0, ..., f_m\}$ in N', and since one must be viable under c in N, we are done. ■

One might wonder at the utility of this theorem, since it is easy to exhibit asymmetric gates: the gate $xy + z$ is clearly symmetric only in x and y. However, we can transform any network of asymmetric gates into a symmetric network, at some increase in the number of viable paths. We develop this in the next section.

2.5 Viability Under Network Transformations

It is well-known that any boolean function may be written in sum-of-products form. Now, we may consider any gate in an arbitrary network to be implemented in this way: for each term, there is a single *and* gate realizing the term, and the *and* gates are the inputs to a single *or* gate which realizes the function. Now, if we assign the internal *and* gates to have weight 0, and the *or* gate realizing node f to have weight $w(f)$, we will not have changed the timing characteristics of the network in any way. Our purpose in this section is to show that the set of viable paths of the network is not decreased by this transformation. In practice, the *and/or* transform is one of a large class of such transforms, which we call *macroexpansion* transforms (since each gate is macroexpanded). Formally, we can write the definition this way:

Definition 2.5.1 *Let N be a network of arbitrary gates. A transform $\mathcal{T}$ is called a* **macroexpansion** *of N if $\mathcal{T}(N)$ has the properties:*

1. *Each gate in $\mathcal{T}(N)$ belongs to precisely one subnetwork $\mathcal{T}(f_i)$.*
2. *For each $f_i \in N$, $\mathcal{T}(f_i)$ is an acyclic digraph consisting of zero or more* **internal** *nodes, each of which has weight 0 and whose fanouts are nodes of $\mathcal{T}(f_i)$, and one output node, designated $\mathcal{O}(f_i)$, whose weight is $w(f_i)$ and whose fanouts are the fanouts of f_i.*
3. *For each $f_i \in N$, $\mathcal{T}(f_i)$ realizes the logic function f_i.*

Informally, each gate in the network is replaced by a subnetwork implementing it.

The example of the *and/or* transform is instructive. If $\mathcal{T}$ is the *and/or* transform, and if $f = c_1 + c_2 + ... + c_n$, each c_i a cube, then $\mathcal{O}(f)$ is an *or* gate whose inputs consist of n *and* gates, $\mathcal{T}(c_1), ..., \mathcal{T}(c_n)$. $w(\mathcal{T}(c_i)) = 0 \;\; \forall\, i$, $w(\mathcal{O}(f)) = w(f)$.

Our purpose is to show for any generic macroexpansion transform $\mathcal{T}$, and for every viable path P in N, there is at least one corresponding viable path $\mathcal{T}(P)$ in $\mathcal{T}(N)$, with $d_N(P) = d_{\mathcal{T}(N)}(\mathcal{T}(P))$. This shows immediately that the critical path delay returned by the viable path algorithm on a macroexpanded network is an upper bound on the true critical path delay. Moreover, since we can certainly macroexpand any network into a symmetric network, we can apply the monotonicity theorems to the symmetric network and be assured that the critical delay obtained is not only upper-bounded in the network N, but also in any faster network N'.

This theorem is a little difficult, and rests heavily on the relationship between the boolean difference, static sensitization, and testing, and on the properties of the smoothing operator. First, note that since the operations of cofactoring and complementation are independent of the implementation of a function, then so to is the boolean difference. Hence we have that

$$\frac{\partial \mathcal{O}(f_m)}{\partial \mathcal{O}(f_{m-1})} = \frac{\partial f_m}{\partial f_{m-1}}$$

The next piece of this puzzle comes from a technical lemma. We wish to show that for every gate in the macroexpanded network, if c satisfies $\mathcal{S}_U \frac{\partial f_m}{\partial f_{m-1}}$, then there is a path viable under c from f_{m-1} to $\mathcal{O}(f_m)$. Now, recall that for any f, and any cube $c^* \supseteq U$, we have:

$$(\mathcal{S}_U f)_{c^*} = f_{c^*-U}.$$

Using this identity, we can consider the macroexpanded subnetwork for f_m. Since the inputs for f_m not in U may be taken as specified by some cube c^*, we wish to consider the network $\mathcal{T}(f_m)_{c^*}$. Since the function $\frac{\partial \mathcal{O}(f_m)}{\partial f_{m-1}}$ has a satisfying assignment, a test exists for both stuck-at-0 and stuck-at-1 on the input lead $\mathcal{O}(f_{m-1})$ for this network. Now we must show:

Lemma 2.5.1 *A shortest path through any node f in a non-trivial network is viable under the cube 1, and hence under any cube c.*

Proof: Induction on $\delta(f)$. If f is a primary input, trivial. Assume for all f s.t. $\delta(f) < L$. If $\delta(f) = L$, let $\{\{P, h\}, f\}$ be a shortest path through f. By induction, the path $\{P, h\}$ is viable under 1 and there is a path viable under 1 through each input of f. Since each such path must be at least as long as $\{P, h\}$, each side input of f is late under every cube, and so under 1. Now, it is trivial that for any non-zero function g, $\mathcal{S}_{fanins(g)} g = 1$, and hence when U is equal to the entire set of side inputs of f we have that $\mathcal{S}_U \frac{\partial f}{\partial h} = 1$, thus $\{P, h, f\}$ is viable under 1. ∎

Theorem 2.5.1 *Let N be any network, $\mathcal{T}(N)$ be the network obtained by any transformation $\mathcal{T}$ satisfying definition 2.5.1. Then for every viable path P in N terminating in f_m, there is at least one viable path $\mathcal{T}(P)$ in $\mathcal{T}(N)$ terminating in $\mathcal{O}(f_m)$, with $d(P) = d(\mathcal{T}(P))$.*

Proof: Consider some viable path $P = \{f_0, ..., f_m\}$ in N, its viability cube c and its transformation, $\{\mathcal{T}(f_0), ..., \mathcal{T}(f_m)\}$. Induction on $\delta(f_m)$. For $\delta(f_m) = 0$, the result is trivial, since f_0 is then a primary input. Assume for a path with $\delta(f_m) < L$. Suppose a path with $\delta(f_m) = L$. Then by assumption there is a path κ in $\mathcal{T}(N)$, viable under c, of length $d(\{f_0, ..., f_{m-1}\})$ terminating in $\mathcal{O}(f_{m-1})$. All that must be shown is the existence of an extension ν of κ of length $w(f_m)$, viable under c, from $\mathcal{O}(f_{m-1})$ to $\mathcal{O}(f_m)$. Since $\{f_0, ..., f_m\}$ is viable under c, there is a set of late inputs U under c such that $\mathcal{S}_U \frac{\partial f_m}{\partial f_{m-1}}$ is satisfied by c. By induction, each of the viable paths to these side inputs in the original network produces a viable path within the transformed network, so the same set U of inputs may be chosen to be late in $\mathcal{T}(N)$. The remaining inputs to $\mathcal{T}(f_m)$ have delays smaller than τ_{i-1} in the initial network, and so we must take their values as their static values under c. These values form a cube, c^*. This cube may be taken to propagate through the network $\mathcal{T}(f_m)$ producing $\mathcal{T}(f_m)_{c^*}$, and hence there is a viable extension of κ from $\mathcal{O}(f_{m-1})$ to $\mathcal{O}(f_m)$ in $\mathcal{T}(f_m)$ iff there is a viable path from $\mathcal{O}(f_{m-1})$ to $\mathcal{O}(f_m)$ in the cofactored network $(\mathcal{T}(f_m))_{c^*}$. Each path through the $(\mathcal{T}(f_m))_{c^*}$ has its length given by the arrival time of the input at its head. Since the only inputs to the cofactored network are $\mathcal{O}(f_{m-1})$ and the late side inputs, if any path from $\mathcal{O}(f_{m-1})$ to $\mathcal{O}(f_m)$ exists it is a shortest path. Hence by lemma 2.5.1 if such a path exists it is viable. We know that such a path exists since $\mathcal{S}_U \frac{\partial f_m}{\partial f_{m-1}}$ is satisfied by c, and so $f_{m_{c^*}}$ is a non-trivial function of f_{m-1}. Let ν be this path. By construction, ν is of length $w(f_m)$. By construction, the delay to $\mathcal{O}(f_m)$ through κ and the prefix of ν is τ_{m-1}, and $\{\kappa, \nu\}$ is viable under c in $\mathcal{T}(N)$. ∎

The converse to this theorem is false. Consider the circuit in figure 2.7. This is the and/or transform circuit of $f = ab + eb$. Now, suppose the variable b is late, and the value of e is set to 1. The function of the original gate is now $ab + b$, which simplifies to b: the reader can easily verify that $\mathcal{S}_b \frac{\partial f}{\partial a} = \bar{e}$, and so this gate is not sensitized to a when $e = 1$. However, in the transformed network, the path $\{a, x, f\}$ is viable when $e = 1$, since the variable y at the or-gate is late.

The fact that the converse is false has an important consequence: we cannot conclude that monotone speedup holds for arbitrary networks. However, we are guaranteed correctness of our algorithms if we transform the network N into a symmetric

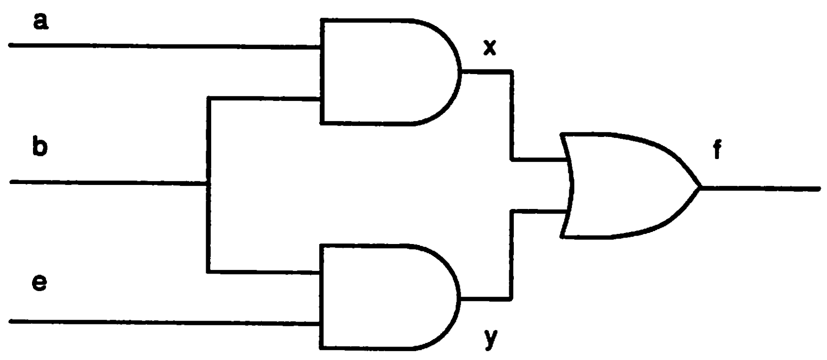


Figure 2.7: And/Or Transform of $f = ab + eb$

one, since theorem 2.5.1 guarantees that for each viable path in the original network we will find a viable path of equal length in the transformed network.

In the next chapter, we shall turn to algorithms. The general method is quite clear; transform the network into the and/or network, and then find the longest viable path in this network. We first introduce a mathematical tool that permits us to view viable paths as the satisfying set of a logic function; this in turn permits the development of a dynamic programming procedure to compute viable paths.

2.6 The Viability Function

At some very fundamental level, any set condition may be expressed as a multiple input logic function. We are interested in making explicit the logic function that underlies viability, because the computation of the viable paths may be made more efficient through explicit computation of this function, because various properties may be proved through use of this function, and because we shall develop a powerful theorem which permits us to quickly establish the correctness of approximation procedures.

The viability function ψ_P on a path P is easy to define: $\psi_P(c) = 1$ iff P is viable under c. This hardly gives more insight than the viable path definition. We prefer to define ψ_P in terms of some function $\psi_P^{f_i}$ at each node f_i. We develop this function intuitively, then justify its definition in a theorem at the end of this section.

Intuitively, we expect that the function ψ_P will be the product of the viability conditions at each node f_i along the path P, which in turn are captured in the function $\psi_P^{f_i}$

$$\psi_P = \prod_{i=0}^{m} \psi_P^{f_i}$$

Since time plays an important part in the definition of viable path, it is convenient to capture it in the definition of the viability function. For any node g, let $\mathcal{P}_{g,t}$ be the set of paths terminating in g of length at least t. Define

$$\psi^{g,t} = \sum_{Q \in \mathcal{P}_{g,t}} \psi_Q$$

Letting U be any subset of $S(f_i, P)$ we can express the viability condition on the subset as:

$$(S_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \psi^{g, \tau_i - 1}$$

since this condition must be satisfied for one subset, we may write:

$$\psi_P^{f_i} = \sum_{U \subseteq S(f_i, P)} (S_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \psi^{g, \tau_i - 1}$$

In summary, we define:

Definition 2.6.1 *The set of paths which terminate in g of length $\geq t$ are denoted $\mathcal{P}_{g,t}$.*

Definition 2.6.2 *The* **viability function** *(also* **viable set***) of a path $P = \{f_0, ..., f_m\}$ is defined as:*

$$\psi_P = \prod_{i=0}^{m} \psi_P^{f_i} \tag{2.2}$$

where

$$\psi_P^{f_i} = \sum_{U \subseteq S(f_i, P)} (S_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \psi^{g, \tau_i - 1} \tag{2.3}$$

and

$$\psi^{g,t} = \sum_{Q \in \mathcal{P}_{g,t}} \psi_Q \tag{2.4}$$

We have immediately:

Theorem 2.6.1 *$P = \{f_0, ..., f_m\}$ is viable under some minterm c iff c satisfies ψ_P.*

Proof:
$\Longrightarrow$ $P = \{f_0, ..., f_m\}$ is viable under c. Induction on $\delta(f_m)$. The base case is trivial, so assume for all paths with $\delta(f_m) < L$. Let $\delta(f_m) = L$. We must show that for each f_i, $c \in \psi_P^{f_i}$. Now, if $c \in \frac{\partial f_i}{\partial f_{i-1}}$, done. Otherwise, since the path is viable under c, we must have that there is a subset $U = \{g_1, ..., g_k\}$ of $S(f_i, P)$, where each g_j terminates a path Q_j which is viable under c and $d(Q_j) \geq \tau_{i-1}$. Since $\delta(g_j) < L$, by induction then $c \in \psi_{Q_j}$ whence $c \in \psi^{g_j, \tau_{i-1}}$ for every j. Further, $c \in S_U \frac{\partial f_i}{\partial f_{i-1}}$, and done.
$\Longleftarrow$ $c \in \psi_P$. Induction on $\delta(f_m)$. The base case is trivial, so assume for all paths with $\delta(f_m) < L$. Let $\delta(f_m) = L$. We must show that the definition of viability holds at each node f_i on the path. Now, if $c \in \frac{\partial f_i}{\partial f_{i-1}}$, done. Otherwise, we must show that there exists a set of side inputs U meeting the conditions of the definition of viable path with $S_U \frac{\partial f_i}{\partial f_{i-1}} \supseteq c$. Since $c \in \psi_P^{f_i}$, then we must have that there is a subset $U = \{g_1, ..., g_k\}$ of the side inputs, and for each g_j, $c \in \psi^{g_j, \tau_{i-1}}$. Now, by the definition of $\psi^{g_j, \tau_{i-1}}$, we must have

$$c \in \sum_{Q_{jl} \in \mathcal{P}_{g_j, \tau_{i-1}}} \psi_{Q_{jl}}$$

Since c is a minterm, for each j it must be in $\psi_{Q_{jl}}$ for some k, and since $\delta(g_j) < L$, by induction Q_{jl} is viable under c, and $c \in S_U \frac{\partial f_i}{\partial f_{i-1}}$, and so done. ∎

Observe that $\psi_P^{f_i}$ is a series, with one term for each subset U of the side inputs. The flaw in using static sensitization is that only one term of this series is taken (the term for $U = \emptyset$).

Chapter 3

False Path Detection Algorithms

Once a correct, robust sensitization criterion has been found, there remains the task of incorporating this criterion in an algorithm which finds the longest path satisfying this criterion; such a path is often called a *longest true path*. The development of such an algorithm is the subject of this chapter and the argument that is to be made is twofold. First, the methods that have appeared in the literature thus far which claim to solve this problem may be viewed as different parameterizations of a single algorithm, and, second, that this algorithm can be modified to compute the viability procedure corresponding to the viability criterion devised in the last chapter.

All of the algorithms that have appeared in the literature to date are of one broad, generic, parameterized class. A collection of partial paths, each of which is known to be true, is maintained. At each step of the algorithm, one such partial path, say $P = \{f_0, ..., f_m\}$ is removed from the structure. If f_m is an output of the circuit, then this is a full true path, the fact is recorded and (if this is a so-called *best-first procedure*) the procedure terminates. If this is not an output, then some unexamined fanout of f_m, say g, is selected to extend the path. A boolean function $\gamma(P, g)$ is computed. If $\gamma(P, g)$ is satisfied, then P, g is a true path and is inserted into the structure of true paths. This procedure continues until a termination criterion is met.

The unity of the algorithms which address this question is not generally recognized; in particular, the identification of the sensitization criterion with a logic function is not usually made. Procedures based on the D-Algorithm (e.g., [10] [6]) compute this function *implicitly.* Nevertheless, it is important to recognize that such a function exists for each algorithm, and indeed, in the case of the papers cited, has a simple explicit form, which we shall divulge in the sequel.

The parameters to this generic procedure are:

1. The search method, which is expressed in terms of the maintained data structure of true paths and in the termination conditions;
2. The definition of what constitutes a true path, the so-called *sensitization conditions*, which expresses itself in the choice of the boolean function γ; and
3. The method used to determine whether the sensitization function is satisfiable.

The choice of these parameters are largely independent. The choice of search method affects the computational complexity of the procedure, while the choice of sensitization condition and satisfiability test affect the *tightness* and also the *correctness* of the procedure: a false-path procedure is *tight* if it provides a least upper bound on the delay of the longest true path; a false-path procedure is *incorrect* if it underestimates the delay before a circuit output settles to its final value.

3.1 Generic False Path Detection Algorithm

We begin our discussion of the single, generic, procedure by mentioning that most authors in the field would argue that there are at least two different methods, *depth-first* and *best-first* search[6]. To a large extent whether one considers these different parameterizations of a generic procedure or two different procedures is a matter of taste; one can, after all, view *any* algorithm as an appropriate parameterization of a Universal Turing Machine. We hope to show that the two search procedures can be viewed as the same basic routine, differing only in termination condition and in the data structure used to store the partial paths.

The *best-first* procedure maintains the partial true paths in a priority queue ordered by the *potential full length*, or length of the longest extension, of the partial path; this quantity is named the *esperance* of the partial path[6]. The best-first procedure terminates when an output is reached, since by construction no longer true path can exist. The depth-first procedure maintains the partial paths on a LIFO stack. When a full path is reached, the path is examined to see if it is of greater length than the longest full true path found so far; if it is, then this is recorded. When the stack is empty, the procedure terminates, and the best path found is returned.

The best-first procedure is slightly more complex than the depth-first procedure, and must be carefully implemented. If it is, then at most KD paths are examined by the procedure, where K is the number of long false paths and D the diameter of the graph. The depth-first procedure will in general examine an exponential number of paths.

3.1.1 Depth-First Search

Depth-first search is a classic graph search algorithm. The central idea is that the fanouts of every node in the graph are ordered, and the subgraph headed by each fanout of a node is explored in its entirety before the succeeding fanout in the order is examined.

The basic, recursive depth-first procedure is depicted in figure 3.1. The basic routine in this code is the function `find_path_dfs`. This function takes one argument, the current true partial path, and returns the longest true full path containing this partial path as a prefix.

The code is fairly self-explanatory. If the partial true path is a full path, then simply return. Otherwise, if there is a longest true full path containing this partial path as a prefix, then this path must be obtainable through some output of the last node of the path, named `node` in this code. Hence, for each such output, attempt to extend the current partial path; if such an extension succeeds, then the answer is to be found by calling the procedure recursively on the obtained successor partial path; finding the longest path over all fanouts yields the solution.

Of course, any recursive procedure can be phrased as an iterative procedure by keeping a last-in, first-out (LIFO) stack; the stack maintains the information that

```
find_longest_true_path() {
    max_length <- 0;
    long_path <- $\emptyset$
    foreach primary input p {
        path <- find_path_dfs(p);
        length <- length(path);
        if(length > max_length) {
            max_length <- length;
            long_path <- path;
        }
    }
}
find_path_dfs(path) {
    node is the last node on path;
    if(output(node)) return path;
    else {
        max_len <- 0;
        long_path <- $\emptyset$;
        foreach fanout p of node {
            if($\gamma\{p, path\} \not\equiv 0$) {
                long1 <- find_path_dfs({p, path});
                length1 <- length(long1);
                if(length1 > max_len) {
                    max_len <- length1;
                    long_path <- long1;
                }
            }
        }
    }
}
```

Figure 3.1: Recursive Depth-First False Path Detection Algorithm

otherwise is implicitly maintained by the sequence of outstanding function calls (in fact, this is precisely the way a machine keeps track of the various function arguments to a procedure). A push onto this stack is equivalent to a recursive call; a pop to a return. For the basic depth-first search procedure, this rephrasing is well worth doing. First, exposing the underlying stack structure inherent in a recursion yields insight into the unity of this approach with the best-first approach; and, second, this permits the search to be easily pruned. We give the code for the iterative version of the depth-first procedure in figure 3.2.

The items which need to be stacked are the explicit argument to `find_path_dfs`, namely the current partial path, and some local variables of that routine. In this case, the only such variable is the fanout next to be explored (the variable p). We represent this as a counter (`path.next_fanout`) associated with the partial path.

An improvement is possible to this routine. In general, this procedure will explore an exponential (in the number of nodes of the graph) number of paths, and so take a very long time to perform the computation. In practice, the search space can be pruned. We are only interested in the longest true path, or in a set of such true paths. If the longest possible extension of a given partial path is shorter than the longest path already found, then there is no point in exploring any extension at all of this path, and one might as well terminate the search immediately. This is easily accomplished. The longest possible extension of a given partial path `path` which terminates in `node` is given by the conjunction of `path` with the longest path originating in `node`. The length of this path is deduced easily in linear time for all the nodes in the network, and may be stored at each node. The code for this calculation is given in figure 3.3.

Once the calculation of the longest path from every node is done, pruning the search space is easy. In figure 3.2, the line

```
if(γ({path, g})≢ 0)
```

is replaced by the line

```
if(((length(path) + best_path_from(g)) > max_length) and (γ({path,
g})≢ 0))
```

```
find_longest_true_path() {
    Initialize stack to primary inputs of the circuit
    max_length <- 0;
    long_path <- $\emptyset$;
    while(path <- top(stack) $\neq \emptyset$) {
        k is the last node on path;
        if(k is an output) {
            if(length(path) > max_length) {
                max_length <- length(path);
                long_path <- path;
            }
        }
        if(path.next_fanout > k.num_fanouts) {
            pop path from stack;
        } else {
            g <- k.fanouts[path.next_fanout];
            path.next_fanout = path.next_fanout + 1;
            if($\gamma$({path, g})$\not\equiv 0$) {
                new_path <- {path, g} is true;
                push new_path on stack;
                new_path.next_fanout <- 0;
            }
        }
    }
    return long_path;
}
```

Figure 3.2: Depth-First False Path Detection Algorithm

```
longest_path_from_nodes() {
    nodes <- array sorted in topological order;
    for i <- |nodes| downto 0 {
        length <- 0;
        foreach fanout p of nodes[i] {
            if(length < best_path_from(p))
                length <- best_path_from(p);
        }
        best_path_from(nodes[i]) <- length + weight(nodes[i]);
    }
}
```

Figure 3.3: Procedure Calculating the Longest Path from Every Node

to obtain the variant of the algorithm with pruning.

Pruning is an effective heuristic technique. Despite this, however, the complexity of depth-first search is still exponential in the number of nodes; there is no guarantee that the pruning technique will eliminate a substantial fraction of the paths. It would be better if *only* the longest true path and the longer false paths are examined. Since these paths must be examined by any algorithm which purports to solve this problem, this procedure is quite efficient.

3.1.2 Best-First Search

The inefficiency in the depth-first procedure arises from the fact that the path removed from the stack at each iteration is the last path shoved on the stack; this makes both the push and pop operations $O(1)$, but gives a deleterious effect on the performance of the algorithm as a whole. It would be better if the partial path of maximum potential length were removed from the stack at every iteration. Indeed, if this were done one might expect a polynomial bound on the complexity of the algorithm if there were not an exponential number of long false paths.

This assurance can be given by revising the data structure underlying the iterative construction. If a *priority queue* is used instead of a LIFO stack, then this guarantee can in fact be given.

A priority queue is a data structure with two major properties.

1. At each iteration, the maximum element of the queue (with respect to some standard order) is at the head of the queue
2. A sequence of n *enqueue* and *dequeue* operations takes $O(n \log n)$ time.

Priority queues are typically implemented on top of *heaps*. A *heap* is defined as a full binary tree with the property that every element is greater than each of its descendants. A full discussion of priority queues and heaps can be found in any good sophomore or junior algorithms text; we took our implementation from [71].

The priority queue is ordered by the *esperance* of a *minimal extension* of a partial path.

Definition 3.1.1 *An* **extension** *of a partial path $P = \{f_0, ..., f_n\}$ is a partial path Q such that P is a prefix of Q. A* **minimal extension** *of P is an extension $\{P, f\}$ where f is a single node. A* **full extension** *of a path P is any extension of P terminating in a primary output.*

Definition 3.1.2 *The* **esperance** *of a partial path P denoted $E(P)$, is defined as the length of the longest full extension Q of P.*

Definition 3.1.3 *The set of* **unexplored extensions** *of a path P with respect to a priority queue, denoted $UE(P)$, is the set of minimal extensions Q of P such that no extensions of Q appear on the queue.*

Note that the esperance of a partial path P is greater than or equal to the esperance of any extension of P.

The operation of the best-first procedure is that, at each iteration, the path with the longest full extension is popped off the queue and extended through the minimal extension with the greatest esperance. Since there is little point in extending a partial path through the same minimal extension twice, the list of minimal extensions through which a given partial path has not been extended is crucial; this is the set of unexplored minimal extensions. Similarly, the metric of interest for determining which path should be extended on the next iteration is not its longest full extension, but rather its longest full extension through an unexplored minimal extension (other full extensions have already been "covered" by preceding extensions). Hence the priority queue is ordered by the esperance of a path through an unexplored minimal extension.

Theorem 3.1.1 *For every partial path P, we have:*

$$E(P) = \begin{cases} d(P) & P \text{ terminates in an output} \\ d(P) + \max_{h_0 \in FO(P)} D(h_0) & \text{otherwise} \end{cases}$$

Proof: If P terminates in an output, then trivial. Otherwise, let Q be the longest full extension of P. Since every full extension of P is obtained through some minimal extension $\{P, h_0\}$ of P, and since $D(h_0)$ is the length of the longest path from h_0 to a primary output, we must have:

$$E(P) = d(P) + D(h_0)$$

for some fanout h_0 of P. ■

Note that this theorem implicitly requires that the fanouts of a node be explored in order of decreasing maximum distance from a primary output. Hence we assume that the fanouts of each node have been sorted in decreasing order by maximum distance from a primary output.

We show the code for the best-first procedure in figure 3.4.

We now prove that the algorithm of figure 3.4 is correct, i.e., returns the longest path true by the sensitization criterion γ. The technique used in the proof of this theorem is the classic *loop invariance* technique[32]. In some sense, this is the program correctness version of induction. In this proof technique, a statement is shown to hold on the $N+1$st iteration of the loop if it holds on the Nth, and is also shown to hold upon entry into the loop.

```
find_longest_true_path() {
    Initialize queue to primary inputs of the circuit
    while((path <- pop(queue)) $\neq$ nil) {
        k is last node on path;
        if(k is an output) return path;
        if(path.next_fanout $\leq$ k.num_fanouts) {
            g <- k.fanouts[path.next_fanout];
            if($\gamma$({path, g})$\not\equiv$0) {
                new_path <- {path, g} is true;
                insert new_path on queue;
                new_path.next_fanout <- 0;
            }
            path.next_fanout = path.next_fanout + 1;
            if(path.next_fanout $\leq$ k.num_fanouts) {
                $E$(path,path.next_fanout) <- $d$(path) +
                    best_path_from(k.fanouts[path.next_fanout]);
                insert path on queue;
            }
        }
    }
}
```

Figure 3.4: Best-First False Path Detection Algorithm

Theorem 3.1.2 *Through each loop of the algorithm every true path Q has a prefix P on the queue such that $E(P) \geq d(Q)$.*

Proof: Loop invariance. As the main loop of the algorithm is entered, this is clearly true, for the set of esperances of the partial paths consisting of only the inputs is equal to the lengths of the longest paths from each input, and one of these is clearly at least as long as the longest true path. Suppose true through N iterations. On the $N+1$st iteration, if the condition is violated then let Q_1 be the true path. Since the condition held through N iterations, then $Q_1 = \{P_1, h_0, ..., h_n\}$ for some P_1, and

P_1 was on the queue through the Nth iteration with $E(P_1) \geq d(Q)$. Further, on the $N+1$st iteration either P_1 is no longer on the queue, or $E(P_1) < d(Q)$. The latter case can only occur if every extension of P_1 with esperance $\geq d(Q)$ has been rejected as false; but the extension $\{P_1, h_0\}$ is true and has esperance $\geq d(Q)$, and so this cannot occur. Similarly, if P_1 has been removed from the queue, then every extension of P_1 through one of its fanouts has been processed, and those found to be true inserted on the queue; this set includes $\{P_1, h_0\}$, and this has esperance $\geq d(Q)$; hence we conclude the statement holds through $N+1$ iterations ∎

Corollary 3.1.3 `find_longest_true_path()` *finds a longest true path.*

Proof: Let Q be a longest true path, Q_1 the longest true path reported by the best-first procedure. All we must show is that $d(Q) = d(Q_1)$. By the theorem, Q has a prefix P on the queue through each loop with $E(P) \geq d(Q)$. But since Q_1 was at the top of the queue on the last iteration, we must have that $E(Q_1) \geq E(P) \geq d(Q)$, and, since $E(Q_1) = d(Q_1)$, we have that $d(Q_1) \geq d(Q)$, hence Q_1 is a longest true path. ∎

For the efficiency of this algorithm, we note the following. Let Q be the longest true path reported by algorithm 3.4. If there are K full false paths longer than Q, and if the diameter of the graph is D, then at most KD partial paths have esperance greater than Q. (This upper bound is obtained through the observation that each full path has at most D prefixes, and the only paths with esperances greater than Q are the false paths and their prefixes). At each loop of the algorithm, one of these partial path or some prefix of Q was examined and extended or rejected as false. There are at most $D(K+1)$ such paths, and hence at most $D(K+1)$ iterations of the algorithm. For the general algorithm, the cost of each iteration is dominated by the determination of whether or not a new partial path is true, which we denote by S[1]; this cost is therefore

$$O(KDS)$$

The remainder of the cost of the algorithm is dominated by the insertions and deletions from the priority queue. The cost of a single insertion or deletion on a priority

[1]As we see in appendix A, this problem is $\mathcal{NP}$-complete for most definitions of γ

queue containing n elements is well-known to be $O(\log n)$. There are at most $O(D)$ elements on the queue at any time, and hence for $O(KD)$ insertions and deletions we have a cost of:

$$O(KD \log D)$$

and hence the cost of the algorithm is:

$$O(KD \log D + KDS)$$

The depth-first and best-first procedures have been compared [6] [83]. The former experiments concluded that the depth-first procedure with pruning outperformed the best-first procedure by a wide margin; this result is surprising and anomalous, given that a complexity analysis would indicate that the overhead of the best-first procedure is at most logarithmic, while the overhead of the depth-first procedure is in general exponential.

The latter set of experiments [83] concluded that the best-first procedure outperformed the depth-first procedure in the related problem of finding the n longest paths in a directed acyclic graph. This result one would expect; however, their experiments also concluded that the margin was very slight, much less than one would expect.

The variance in the best- vs depth-first procedures reported in [6] offers one possible explanation. A minor error in the implementation of the best-first procedure can lead to a large number of paths being searched. If, when a path P is found true, *every* one-node extension of P (as opposed to merely the best) is placed on the queue, then if every node has an average fanout of k, tracing a single path to the output will result in kD nodes being placed on the queue. Note in the best-first procedure given above only *one* extension of P is explored when P is found true, and hence only D paths are placed on the queue as P is explored.

3.1.3 Generic Procedure

Now, notice that the code in algorithms 3.4 and 3.2 are very similar; the principal distinction is in the data structure used to represent the set of active partial paths. A second difference is in the termination condition used and in the steps the algorithm performs when it finds a full true path. The distinction between priority queue and stack has been treated adequately above. The difference in termination condition and action on finding a full true path is an artifact of the fact that the best-first procedure is so constructed that first true path found is also a longest true path; the depth-first procedure offers no such assurance, so the search must be continued, after recording the fact that a new long path has been found.

Nevertheless, the two procedures are similar enough that one can consider them a single generic procedure, parameterized by search method. The code is shown in figure 3.5.

The generic procedure can easily be modified to permit pruning under depth-first search.

3.2 Variants on the Problem

Two related problems to the basic timing verification problem are also addressed by timing analyzers: viz., finding *every* longest true path, and the problem of finding the true path of minimum slack. It is the purpose of this section to demonstrate that the generic procedure is capable of solving either of these problems.

3.2.1 Modifying the Generic Procedure to find Every Longest True Path

In some applications (for example, resynthesis for timing) it is desirable not only to find the longest true path, but also every true path within some ϵ of the longest true path; the rationale is that if the longest true path fails to meet timing specifications, and the circuit must be resynthesized for timing [74], then it is of little use to modify only the critical path if another series of paths remain true and equally long. Another variant on this procedure is to report every true path of length greater than some threshold, which represents the maximum allowable delay of the circuitry.

These two problems require the same modification to the basic algorithm;

```
find_longest_true_path() {
    Initialize paths to primary inputs of the circuit
    if(depth_first_search)
        long_path <- ∅; maxlen <- 0;
    while(path <- pop(paths)) {
        k is last node on path;
        if(k is an output) {
            if(depth_first_search) {
                if ($d$(path) > maxlen) {
                    maxlen = $d$(path);
                    long_path <- path;
                }
            }
            else return path;
        }
        if(path.next_fanout ≤ k.num_fanouts) {
            g <- k.fanouts[path.next_fanout];
            path.next_fanout = path.next_fanout + 1;
            if(path.next_fanout ≤ k.num_fanouts) {
                $E$(path) <- $d$(path) +
                    best_path_from(k.fanouts[path.next_fanout]);
                insert path on paths;
            }
            if($\gamma$({path, g}) $\not\equiv 0$) {
                new_path <- {path, g} is true;
                insert new_path on paths;
                new_path.next_fanout <- 0;
            }
        }
    }
    if(depth_first_search) return long_path;
    else return ∅;
}
```

Figure 3.5: Generic False Path Detection Algorithm

a threshold T is computed, and all paths of length $> T$ are reported; in the case of the first variant, $T = L - \epsilon$, where L is the length of the longest path and ϵ is the user-supplied tolerance. In the case of the second variant, T is user-supplied.

The modification to the generic procedure is conceptually fairly simple; the routine merely maintains a list of paths of length $> T$, and returns this list once it is clear that no further paths will be found of length $> T$; in the case of the best-first procedure, this occurs when the top path on the queue has *esperance* $\leq T$; in the case of the depth-first procedure, this is simply when the stack is empty, though pruning can be used effectively here, as well.

A little bookkeeping is required when the threshold is set dynamically to $L - \epsilon$. Again, in the case of the best-first procedure, this is a fairly simple matter; the threshold is initially set to 0, and when the longest path is found the threshold is set to $L - \epsilon$. In the case of a depth-first search, the threshold is recalculated every time a new longest path is found. The code is shown in figure 3.6

One interesting note here is that the time of the best-first procedure to find all paths of length greater than the threshold is determined simply by the number of paths, false and true, of length greater than the threshold. This may or may not be greater than the total number of long false paths in the circuit, if the threshold is fixed.

3.2.2 Varying Input Times, Output Times, and Slacks

In practice, system requirements often dictate that inputs to a circuit arrive at differing times, or that outputs are required at varying times, or both. Timing analyzers often take this into account by computing the arrival and required times for a signal separately (using the analogous formula for required time), and then compute the *slack* for each node as the difference between the required and arrival time; it is easy to show that there is at least one sequence of nodes with the minimum slack, and this is defined as a *critical path* of the circuit.

We define the *slack* of a path P, $slack(P)$ as follows. Let $P = \{i_k, f_0, ..., f_m, o_j\}$, input arrival time of i_k is t_k, output required time of o_j is t_j. Then the slack of P is

```
find_longest_true_path() {
    Initialize paths to primary inputs of the circuit
    long_paths <- $\emptyset$;
    if(depth_first_search) maxlen <- 0;
    if(search_by_epsilon) $T$ <- 0;
    while(path <- pop(paths)) {
        k is last node on path;
        if(k is an output)
            output_reached(path, $T$, long_paths, $\epsilon$, maxlen);
        if(best_first_search and $E$(path) $\leq T$) return long_paths;
        if(path.next_fanout $\leq$ k.num_fanouts) {
            g <- k.fanouts[path.next_fanout];
            path.next_fanout = path.next_fanout + 1;
            if(path.next_fanout $\leq$ k.num_fanouts) {
                $E$(path) <- $d$(path) +
                     best_path_from(k.fanouts[path.next_fanout]);
                insert path on paths;
            }
            if($\gamma$({path, g}) $\not\equiv 0$) {
                new_path <- {path, g} is true;
                new_path.next_fanout <- 0;
                insert new_path on paths;
            }
        }
    }
    return long_paths;
}
```

Figure 3.6: Procedure Returning All Longest Paths

```
output_reached(path, $T$, long_paths, $\epsilon$, maxlen) {
    if($d$(path) $> T$) long_paths <- long_paths $\cup$ {path};
    if(depth_first_search and $d$(path) > maxlen and search_by_epsilon) {
        must update the threshold and determine which paths still
        are longer than the threshold
        $T$ <- $d$(path) - $\epsilon$; new_long_paths <- {path};
        foreach path p1 on long_paths
            if $d$(p1) > $T$ new_long_paths <- {p1} $\cup$ new_long_paths;
                maxlen <- $d$(path); long_paths <- new_long_paths;
    }
    else if($T = 0$) {
        long_paths <- {path};
        $T$ <- $d$(path) - $\epsilon$;
    }
}
```

Figure 3.7: Auxiliary Procedure to Procedure Returning All Longest Paths

defined:

$$slack(P) = t_j - [t_k + \sum_{i=0}^{m} w(f_i)].$$

Intuitively, the slack of P is the difference between the required time of o_j and its arrival time down the path. The *critical path* of a circuit is the true path of minimum slack.

It is possible to perform an analogous calculation and eliminate false paths, but this would require a significant restructuring of the algorithm given above. All things being equal, it would be better to adapt the existing algorithm by appropriate manipulation of the graph structure of the circuit.

Fortunately this adaptation is trivial. If inputs arrive at different times, we might as well consider the time when the earliest input i_1 arrives to be 0 (to avoid the inconvenience of negative weights). Let each input i_j arrive at $t = t_j$; we can then consider i_j as an internal node of the circuit, the output of a static delay buffer of weight t_j whose input is a new input i'_j, which arrives at $t = 0$. The resulting circuit has the property that every path originating in i_j has a corresponding path in the original circuit originating in i'_j of identical delay.

Similarly, if outputs have differing required times, let t_{max} be the maximum required time of all the outputs. To each output o_j with required time t_j attach a static delay buffer of weight $t_{max} - t_j$, input o_j, output o'_j. Hence the required time of o_j in the resulting circuit is t_j. The resulting circuit has the property that every path in the original circuit terminating in o_j with delay d has a corresponding path in the resulting circuit terminating in o'_j of delay $d + t_{max} - t_j$. Ideally, we wish to show that this transformation of the circuit graph preserves the set of critical paths of the circuit; in particular, we wish to show the following:

Theorem 3.2.1 *Let $P = \{i_k, f_0, ..., f_m, o_j\}$ be any path in the original circuit. Then $P' = \{i'_k, f_0, ..., f_m, o'_j\}$ is the corresponding path in the resulting circuit, and $slack(P) = slack(P')$.*

Proof:

$$\begin{aligned} slack(P) &= t_j - [t_k + \sum_{i=0}^{m} w(f_i)] \\ &= t_{max} - (t_{max} - t_j) - [t_k + \sum_{i=0}^{m} w(f_i)] \\ &= t_{max} - [t_k + (\sum_{i=0}^{m} w(f_i)) + (t_{max} - t_j)] \\ &= slack(P') \end{aligned}$$

■

Now, note that P' is true iff P is true, and, hence, if P is a critical path then P' is a critical path. Further the primary inputs in the resulting network all have arrival times of 0, and the primary outputs all have required times of t_{max}. Hence the critical path in the transformed circuit is the longest true path, as desired.

3.3 Dynamic Programming Procedure for Viability

The assumption underlying the generic procedure was that the function γ was a function only of the path being extended. An examination of the viability equations (2.2)-(2.4) and the surrounding discussion demonstrates that this assumption is unfulfilled by the viability function. The viability function of a path is not only a function of the path itself, but also of all adjoining paths at least as long; this dependence expresses itself in the subfunction $\psi^{g,\tau_{i-1}}$. Correct computation of the viability function requires that this function be computed for each side input to each node on the candidate path as that node is encountered. Conceptually, this could be done by recursively tracing the set of viable paths terminating in each side input as a node is encountered on the candidate path.

Our first algorithm is designed to find a viable path of length at least L, terminating in some target node f_i, with a prefix on a given `stack`. Conceptually, it is based on the iterative version of the depth-first procedure explored earlier in this chapter. It is shown in figure 3.8.

If we assume for the moment that the `viable_set` computes the function:

$$\sum_{U \subseteq S(f_i,P)} (S_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \psi^{g,\tau_{i-1}}$$

then the correctness of this algorithm is easily established. A partial path is popped off the stack; if it is complete, then we are done and return. Otherwise, each successor node is examined to see if it may extend this path fruitfully; if it can, the extended path is pushed on the stack. Those nodes which cannot possibly extend this path to the desired length at the target node, and those which are not viable, cannot extend this path. This process continues until either no paths may be extended (the stack is empty), or one path is complete. Note that in addition to returning a viable path of the appropriate length terminating in `target_node`, this routine also returns the final `stack`; this is to permit this routine to be called iteratively by a procedure which finds all the viable paths of the appropriate length.

```
find_viable_path(stack, network, target_node, length)
{
    while((path <- pop(stack)) ≠ ∅) {
        if(last_node(path) ≡ target_node) return(path, stack);
        foreach fanout c of last_node(path) {
            if(path_length(path)+longest_path(c,target_node)<length)
                continue;
            if (c ≡ target_node) psi <- 1;
            else psi <- viable_set(c, last_node(path), network,
                path_length(path));
            new_psi <- psi * path_psi(path);
            if(new_psi ≢ 0) {
                new_path <- c,path;
                path_length(new_path)<-weight(c)+path_length(path);
                path_psi(new_path) <- new_psi;
                push(new_path, stack);
            }
        }
    }
    return (∅,∅);
}
```

Figure 3.8: Naive Algorithm to Find the Longest Viable Path

It is now time to write the function which computes all the viable paths of length at least L, and which terminate in node target_node. It is shown in figure 3.9. Note this procedure consists simply of calling find_viable_path repeatedly until the stack is exhausted. The mechanism of passing the stack into and out of find_viable_path is simply a means of preserving the stack over calls to find_viable_path.

The correctness of this routine is easy to establish, given the correctness of

```
find_all_viable_paths(node, network, length)
{
    list <- [];
    stack <- [];
    foreach primary_input p of network {
        P <- a new path of node p, length 0, psi <- 1;
        push P on stack;
    }
    /* Initialize path and stack */
    (path, stack) <- find_viable_path(stack, network, node, length);
    while(path $\neq \emptyset$) {
        list <- list, path;
        (path, stack)<-find_viable_path(stack,network,node,length);
    }
    return list;
}
```

Figure 3.9: Naive Algorithm to Find All the Long Viable Paths

find_viable_path, simply by observing that the correct set of initial paths are the paths consisting of only the primary inputs. These paths have viability function 1, and length 0.

With these procedures in hand, viable_set falls out easily from the definition above, and is shown in figure 3.10. This procedure simply finds all the viable paths of length τ_{i-1} terminating in each side input k to c, and sums up their viability functions in the field k.psi. The function $\psi^{\mathbf{k},\tau_{i-1}}$ is thus computed and stored in k.psi, and the viable set falls out easily by equation 2.3.

The algorithm to find the longest viable path is similarly easy (assuming we have made the obvious trivial change to find_viable_path so that the target node may be any one of a set). It is shown in figure 3.11

```
viable_set(c, prev_node, network, length)
{
    psi <- 0;
    foreach input k of c, k $\neq$ prev_node {
        k.psi <- 0;
        list <- find_all_viable_paths(k, network, length);
        foreach path p on list
            k.psi = k.psi + path_psi(p);
    }
    foreach subset $U$ of the side inputs {
        new_psi <- $S_U \frac{\partial c}{\partial \text{prev\_node}}$;
        foreach k in $U$
            new_psi <- new_psi * psi[k];
        psi <- psi + new_psi;
    }
    return psi;
}
```

Figure 3.10: Viable Set Algorithm

This, however, may cause the same partial path to be traced potentially many times, each time it is encountered as an abutting path to a shorter candidate path. A second alternative is to store the viability functions for each traced path, and only trace side paths recursively when these are known not to have been traced. A still better alternative would be to avoid the recursive path tracing at all. This can be done if:

1. It is known that every longer side path to the candidate path has been traced; and

2. The function ψ^{g,τ_i-1} is maintained in a variable attached to g and is known to be correct, i.e., is known to contain the sum of the viability functions of all such

```
find_longest_viable_path(network)
{
    foreach primary_input p of network {
        P <- a new path with nodelist <- {p}, length <- 0, psi <- 1;
        push P on stack;
    }
    (path, stack) <-
        find_viable_path(stack, network, primary_outputs(network), 0);
    while(path $\neq \emptyset$) {
        oldpath <- path;
        length <- path_length(path);
        (path, stack) <-
            find_viable_path(stack, network, primary_outputs(network), length);
    }
    return oldpath;
}
```

Figure 3.11: Naive Algorithm to Find the Longest Viable Path

longer side paths.

These assurances can be given by a *dynamic programming* procedure based on the best-first procedure. Recall that the best-first procedure examines partial paths in decreasing order of their *esperance*, which is also their *potential full length.*

Simply using an ordering on esperance is insufficient to our purposes, however. If we are examining a node f_i and attempting to extend a path P of length τ_{i-1}, we must ensure that *all* side viable paths of P at f_i of length $\geq \tau_{i-1}$ have been examined. However, a side path of length precisely τ_{i-1} may well have esperance equal to $E(P)$. We must break this tie in favour of the path whose last node is of *lesser* level. With this in mind, we can define a successor relation $\succ$ on partial paths.

Definition 3.3.1 *Let $Q = \{g_0, ..., g_n\}$, $P = \{f_0, ..., f_m\}$. $Q \succ P$ iff $E(Q) > E(P)$ or $E(P) = E(Q)$ and $\delta(g_n) < \delta(f_m)$.*

Lemma 3.3.1 *Let $P = \{f_0, ..., f_i\}$ be a partial path. Let $Q = \{g_0, ..., g_n\}$ be a side path to P at f_i. If $d(Q) \geq \tau_{i-1}$ then $Q \succ P$.*

Proof: Let P, Q be as stated in the premise of the lemma, $d(Q) \geq \tau_{i-1}$. Then $E(P) = \tau_{i-1} + w(f_i) + K$, where K is the maximum distance from P to a primary output. But $E(Q) \geq d(Q) + w(f_i) + K$, whence $E(P) \leq E(Q)$. Since $\delta(f_i) > \delta(g_n)$, $Q \succ P$. ∎

This result gives us the tool we need to avoid excessive computation of the viable sets if we replace the stack in `find_viable_path` with a priority queue of extensions ordered in descending order under $\succ$; thus, at each iteration, we attempt the extension that is maximal wrt $\succ$, and we are guaranteed that we have examined all partial paths Q such that $Q \succ P$. Note that this procedure is very similar to the best-first procedure, differing only marginally in the function by which the priority queue is ordered.

We need variables in which to keep the sums of the viability functions of the paths traced thus far. At each connection, from a node g to some node f we keep a field $f[g].psi$. Each such field is initially 0. As we pop an extension $\{P, f\}$ off the queue, ending in node g, we set $f[g].psi = f[g].psi + \psi_P$ The rationale is that the path P is certainly true, and since it was on the top of the queue it is certainly longer than any other path on the queue that may be extended through f. A depiction of the dynamic programming variables appears in figure 3.12.

One other minor modification is necessary: *all* unexplored minimal extensions of maximal esperance of a true partial path P must be placed on the priority queue, not merely any unexplored minimal extension. This must be done to correctly handle the case of extensions which have equal *esperance*.

At first glance, it would appear that this procedure is sufficient to ensure that $f_i[g]$.`psi` is equal to $\psi^{g,\tau_{i-1}}$ when the best-first procedure attempts to extend $\{f_0, ..., f_{i-1}\}$ through f_i. In fact, this procedure underestimates $\psi^{g,\tau_{i-1}}$. Certainly if Q is a path terminating in g such that $d(Q) > \tau_{i-1}$ then the procedure forces $f_i[g]$.`psi`

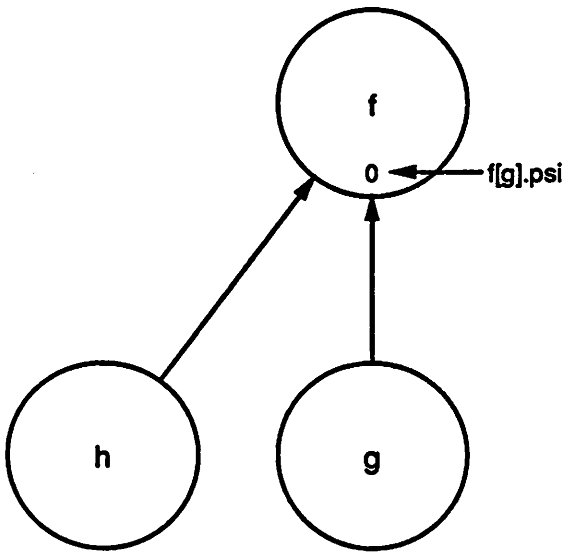


Figure 3.12: Viability Algorithm Variables

to contain ψ_Q, for then $\{Q, f_i\} \succ \{f_0, ..., f_i\}$ and hence Q was popped off the queue and extended through f_i before $\{f_0, ..., f_{i-1}\}$ was. However, there remains the case where $d(Q) = \tau_{i-1}$. In particular, consider the case where a number of paths of equal length conjoin at f_i, each path P_j terminating in g_j. Before any P_j can be extended through f_i, the field $f_i[g_j]$.psi must be updated for each g_j. But the field for an arbitrary g_j is not updated until the attempt is made to extend P_j, i.e., when we attempt to extend the first P_j none of the relevant fields have been updated. The problem is illustrated in the diagram in figure 3.13. In this figure, g_0, g_1, and g_2 all terminate viable paths of length 10, with the viability cubes shown underneath each path. If P_0 is picked to extend first, the viability functions kept at g_1 and g_2 are both 0, even though g_1 and g_2 both terminate viable paths as long as P_0.

The solution is to sum the viability functions for the P_j into $f_i[g_j].psi$ when we attempt to extend *any* of the P_j. The difficulty is in finding the P_j. We have the following.

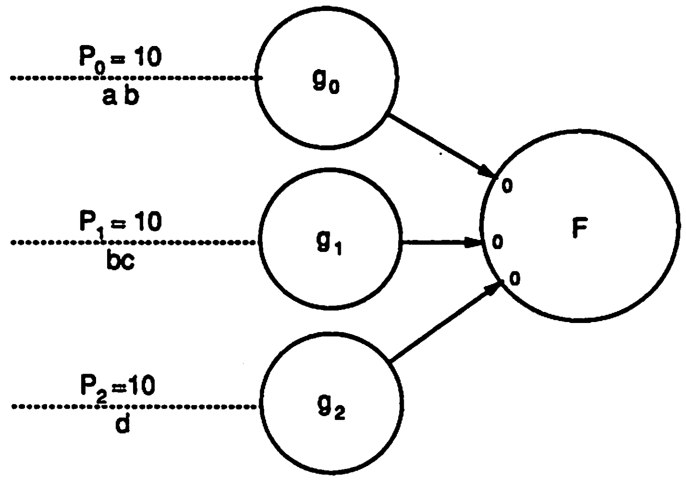


Figure 3.13: Underestimation of Viability Function

Lemma 3.3.2 *Let Q, P be paths such that $d(Q) = d(P)$, and such that $\{P, f_i\}$ is an extension on the queue maximal under $\succ$. Then if $\{Q, f_i\}$ is unexplored, it is also on the queue and maximal under $\succ$.*

Proof: Note that $E(\{Q, f_i\}) = d(Q) + w(f_i) + D(f_i)$, and since $d(P) = d(Q)$, we have $E(\{Q, f_i\}) = E(\{P, f_i\})$. Further, since the paths $\{Q, f_i\}$ and $\{P, f_i\}$ have the same terminal node, the levels of their terminal nodes are equal. Hence $\{Q, f_i\} \not\succ \{P, f_i\}$ and $\{P, f_i\} \not\succ \{Q, f_i\}$. Hence if $\{Q, f_i\}$ is on the queue, it is maximal under $\succ$, for $\{P, f_i\}$ is maximal under $\succ$ by assumption. Since $\{Q, f_i\}$ is unexplored, then either it is on the queue or some prefix is. Every prefix Q' of $\{Q, f_i\}$ (excepting $\{Q, f_i\}$) is such that $Q' \succ \{Q, f_i\}$, and hence $Q' \succ \{P, f_i\}$. Hence Q' is not on the queue, for that would violate the assumption that $\{P, f_i\}$ is maximal under $\succ$. ■

It is important to show that the converse holds as well.

Lemma 3.3.3 *Let Q, P be paths such that $\{P, f_i\}, \{Q, f_i\}$ are extensions on the queue maximal under $\succ$. Then $\{Q, f_i\}$, $\{P, f_i\}$ are both unexplored, and $d(Q) = d(P)$.*

Proof: Both P and Q are viable paths such that $\{P, f_i\}, \{Q, f_i\}$ are unexplored, by construction of the queue. Further, since both are maximal under $\succ$, we must have $E(\{Q, f_i\}) = E(\{P, f_i\})$, and hence

$$d(Q) + w(f_i) + D(f_i) = d(P) + w(f_i) + D(f_i)$$

hence $d(Q) = d(P)$ and done. ■

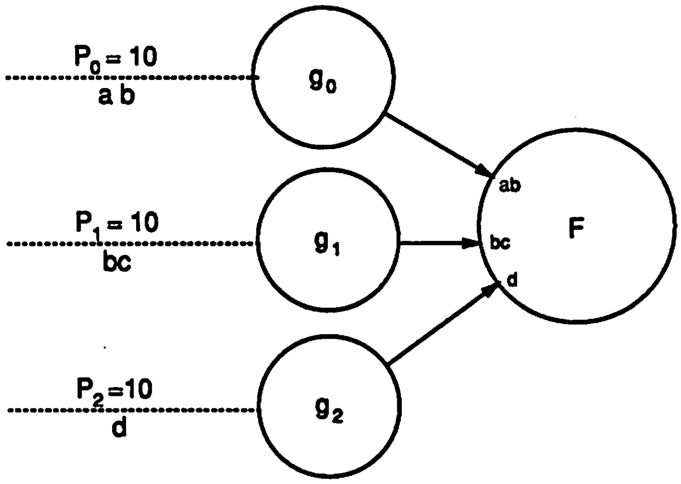


Figure 3.14: Correct Calculation of Viability Function

The picture the above two lemmas gives us is of a frontier of paths, maximal under $\succ$, each of whose viability functions must be summed into the relevant dynamic programming variable before any can be extended. If this is done, then for the previous example we have the correct calculation shown in figure 3.14.

We now restate, operationally, how the field $f[g].psi$ is maintained for nodes f and g.

$$f[g].psi = \begin{cases} 0 & \text{initially} \\ f[g].psi + \psi_{\{P,f,g\}} & \{P, f, g\} \text{ is popped off the queue and found to be viable} \end{cases} \tag{3.1}$$

Let Q_{g,f_i} be the set of paths $\{Q,g\}$ such that $\{Q,g,f_i\}$ is an extension on the queue maximal under $\succ$. The computation of `viable_set`$(\{P,f_i\})$ is then given by

$$\texttt{viable\_set}(\{P,f_i\}) = \sum_{U\subseteq S(f_i,P)} (S_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g\in U} \left(f_i[g].psi + \sum_{Q\in Q_{g,f_i}} \psi_Q \right) \qquad (3.2)$$

We can now state the main result of this section, which proves the dynamic programming algorithm, detailed intuitively above and given in detail in figure 3.15, is correct.

Theorem 3.3.1 `viable_set`$(f_i) = \psi_P^{f_i}$

Proof: Induction on $\delta(f_i)$. For the base case, f_i is a primary input, whence $\frac{\partial f_i}{\partial f_{i-1}} = 1$ and done. Assume for $\delta(f_i) < L$. Now, for $\delta(f_i) = L$ the theorem holds if we can show that:

$$f_i[g].psi + \sum_{Q\in Q_{g,f_i}} \psi_Q = \psi^{g,\tau_{i-1}}$$

for the general case. But:

$$\psi^{g,\tau_{i-1}} = \sum_{d(Q)\geq\tau_{i-1}} \psi_Q$$

where Q is a partial path ending in g. Now, if $d(Q) > \tau_{i-1}$, or if the level of g is less than that of the last node of P, then $Q \succ P$, and hence has been examined previously by the algorithm. Since $\delta(g) < L$, by the induction hypothesis for each such path Q ψ_Q was correctly calculated. Further, as each such Q was popped off the path for extension through f_i, ψ_Q was added into $f_i[g].psi$ by equation (3.1) and hence $f_i[g].psi \supseteq \psi_Q$ for all such paths Q. There remains the case where Q and P are incomparable under $\succ$. In this case, by lemmas 3.3.2-3.3.3, Q and P are on the same frontier, and hence the viability function of Q is added into $f_i[g].psi$ before P is extended through f_i. Hence

$$f_i[g].psi + \supseteq \psi^{g,\tau_{i-1}}$$

For equality, all we must show is that no path Q either

1. has had its viability function ψ_Q summed incorrectly into $f_i[g].psi$; or
2. is incorrectly in Q_{g,f_i}

The first can only occur if the algorithm has examined some path Q before P with $d(Q) < d(P)$. But then $E(\{P, f_i\}) > E(\{Q, f_i\})$, contradiction. The second is forbidden by lemma 3.3.3, and done. ∎

The code for the formal procedure is given in figure 3.15 and in figure 3.16. Much care is taken in this procedure to ensure that the scheduling assumptions of theorem 3.3.1 are met. In particular, when a path is found to be viable, *every* minimal extension of maximal esperance of that path is added to the queue; this is used to enforce the assumption that every unexplored path of maximal esperance terminating in minimal level (that is, every unexplored path maximal under the relation $\succ$) is present on the queue at all times; this is a requirement for lemmas 3.3.2-3.3.3 to hold. Similarly, when the last extension of a path of some fixed esperance is found to be viable, or not, then the set of extensions of that path of the next higher esperance must be added to the queue.

While this algorithm finds every viable path of maximal length, it does so at great expense. Recall that the relation $\succ$ required that we break ties in esperance in favour of the path whose terminus is of lesser level. This forced the best-first procedure ordered on esperance into what is almost a *breadth-* first procedure, and leaves us open to the possibility that a larger number of paths will be explored than really need to be. In fact, if there are K_1 paths longer than the longest true path, and K_2 paths as long as the longest true path, this procedure explores $O((K_1 + K_2)D)$ paths, as opposed to the $O(K_1 D)$ paths that strictly need to be explored.

What we would like is some way to break ties in favour of the path whose terminus is of *greater* level. We can do this if we guarantee that if we reject any viable path incorrectly, we will subsequently accept one at least as long. This is obtained as a consequence of the following theorem and corollary: in a symmetric network, if there are a set of partial paths of equal length conjoined at a single node and if one may be extended to a full path, then all may. This theorem also shows that we need not add the sum of the viability functions of the maximal paths on the queue into

```
find_longest_true_path(){
    Initialize queue to primary inputs of the circuit
    while(queue $\neq$ nil) {
        frontier <- set of paths on queue maximal under $\succ$;
        foreach extension (path,g) on frontier {
            k is the last node of path;
            g[k].psi <- g[k].psi + $\psi$(path);
        }
        while(((path, g) <- pop(frontier)) $\neq$ nil) {
            $\psi$ <- viability_function(path, g)
            if($\psi \not\equiv 0$) {
                new_path <- {path, g};
                if(g is an output) return new_path;
                $\psi$(new_path) <- $\psi$;
                foreach extension np <- {new_path, h} of new_path
                    if($E$(np) = $E$(new_path))
                        insert np on queue
            }
            if every extension ext of path s.t.
                $E$(ext) = $E$(new_path) has been explored {
                ext1 is next best extension of path
                push ext1 on queue;
                foreach extension ext2 of path with $E$(ext1) = $E$(ext2)
                    push ext2 on queue;
            }
        }
    }
}
```

Figure 3.15: Dynamic Programming Procedure to Find the Longest Viable Path

```
viability_function(path, g) {
    k is the last node of path;
    sense_fn <- 0;
    esp = $E$({path, g});
    foreach subset $U$ of the side fanins of g {
        product <- $S_U \frac{\partial g}{\partial k}$;
        foreach j $\in U$ while product $\not\equiv 0$ {
            sum <- g[j].psi;
            product <- product * sum;
        }
        sense_fn <- sense_fn + product;
    }
    return $\psi$(path) * sense_fn;
}
```

Figure 3.16: Viability Function for Dynamic Programming Procedure

the field $f[g].psi$ when computing the viable set for extending a path from g through f. We now prove this statement.

Theorem 3.3.2 *Consider a symmetric network. Let P_1, P_2 be partial paths, viable under c, with $d(P_1) = d(P_2)$, P_1 terminates in g_1, P_2 terminates in g_2, g_1 and g_2 are fanins to a node h, such that $\{P_1, h\}$ is viable under c. Then $\{P_2, h\}$ is viable under c.*

Proof: Suppose $g_1 \neq g_2$. Since $\{P_1, h\}$ viable under c, there is some set U of the fanins of h such that:

$$c \subseteq S_U \frac{\partial h}{\partial g_1} \prod_{g \in U} \psi^{g, d(P_1)}$$

since there is a path terminating in g_2 of length $d(P_1)$ (namely P_2), $c \subseteq \psi^{g_2, d(P_1)}$.

Hence we can choose $g_2 \in U$, for certainly if $g_2 \notin U$:

$$c \subseteq \mathcal{S}_{g_2}\mathcal{S}_U \frac{\partial h}{\partial g_1} \prod_{g \in U} \psi^{g,d(P_1)} \psi^{g_2,d(P_1)} = \mathcal{S}_{U+\{g_2\}} \frac{\partial h}{\partial g_1} \prod_{g \in U+\{g_2\}} \psi^{g,d(P_1)}$$

Since, by symmetry, $\mathcal{S}_{U-\{g_2\}+\{g_1\}} \frac{\partial h}{\partial g_2} = \mathcal{S}_U \frac{\partial h}{\partial g_1} \supseteq c$, and since $\psi^{g_1,d(P_2)} \supseteq c$, we have:

$$c \subseteq \mathcal{S}_{U-\{g_2\}+\{g_1\}} \frac{\partial h}{\partial g_2} \prod_{g \in U-\{g_2\}+\{g_1\}} \psi^{g,d(P_2)}$$

and so $\{P_2, h\}$ is viable under c. Now, if $g_1 = g_2$, and we have the set U such that that:

$$c \subseteq \mathcal{S}_U \frac{\partial h}{\partial g_1} \prod_{g \in U} \psi^{g,d(P_1)}$$

since $g_1 = g_2$ and $d(P_1) = d(P_2)$, a simple substitution shows that:

$$c \subseteq \mathcal{S}_U \frac{\partial h}{\partial g_2} \prod_{g \in U} \psi^{g,d(P_2)}$$

and hence $\{P_2, h\}$ is viable under c. ■

Corollary 3.3.3 *In a symmetric network, if $\{P_1, .., P_n\}$ are partial paths, viable under c, each terminating in a fanin to some node h_0, such that $d(P_i) = d(P_j)$ for all i, j, and if one of the P_i is a prefix (through h) to a path $Q_i = \{P_i, h_0, h_1, ..., h_r\}$, viable under c, then each P_j is a prefix through h to a path $Q_j = \{P_j, h_0, h_1, ..., h_r\}$, viable under c, such that $d(Q_i) = d(Q_j)$.*

Proof: Induction on r. If $r = 0$, immediate from theorem 3.3.2. Assume for $r < R$. If $r = R$, let the P_1 of theorem 3.3.2 be $\{P_i, h_0, ...h_{r-1}\}$, and h of that theorem be h_R. Result follows immediately. ■

This gives us the tool we need. We can only incorrectly reject a path if it is of the form P_i in the above the theorem. However, of the set $\{P_1, .., P_n\}$, we will examine *one* last, and if we have accepted no other prior to that we shall accept that one. Note that this will not give us the list of *all* viable paths of delay equal to the delay of the longest viable path; that must be accomplished by a close variant of the original algorithm, detailed below.

```
find_longest_true_path(){
    Initialize queue to primary inputs of the circuit
    while(((path, g).<- pop(queue)) $\neq$ nil) {
        k is the last node of path;
        g[k].psi <- g[k].psi + $\psi$(path);
        $\psi$ <- viability_function(path, g)
        if($\psi \not\equiv 0$) {
            new_path <- {path, g};
            if(g is an output) return new_path;
            $\psi$(new_path) <- $\psi$;
            np <- {new_path, h} is best extension of new_path
            insert np on queue
        }
        ext <- {path, f} is the next best extension of path
    if ext is not nil push ext on queue;
    }
}
```

Figure 3.17: Improved Dynamic Programming Procedure to Find an LVP

The improved dynamic programming procedure is given in figure 3.17. Note that this procedure is much closer in form to the best-first procedure demonstrated above.

The correctness of this procedure is easy to establish.

Theorem 3.3.4 *Let $P = \{f_0, ..., f_m, h_0, ..., h_p\}$ be a viable path under c. Then either P is reported as viable by the algorithm, or some path $Q = \{g_0, ..., g_n, h_0, ..., h_p\}$, with $g_n \neq f_m$ and $d(\{f_0, ..., f_m\}) = d(\{g_0, ..., g_n\})$ is reported as viable by the algorithm.*

Proof: If $P = \{f_0, ..., f_m, h_0, ..., h_p\}$ is rejected by the algorithm, then at some node the computed viability function of the path as computed is a proper subset of the

```
viability_function(path, g) {
    k is the last node of path;
    sense_fn <- 0;
    esp = $E(\{path, g\})$;
    foreach subset $U$ of the side fanins of g {
        product <- $S_U \frac{\partial g}{\partial k}$;
        foreach j $\in U$ while product $\not\equiv 0$ {
            sum <- g[j].psi;
            product <- product * sum;
        }
        sense_fn <- sense_fn + product;
    }
    return $\psi$(path) * sense_fn;
}
```

Figure 3.18: Viability Function for Improved Dynamic Programming Procedure

viability function of the partial path to that node. Let h_0 be the first such node. The viability function to that point is

$$\psi_P^{f_m}[\sum_U S_U \frac{\partial h_0}{\partial f_m} \prod_{g \in U} \psi^{g, d(\{f_0, \ldots, f_m\})}]$$

Now, by assumption, $\psi_P^{f_m}$ has been correctly calculated by the procedure, and hence we must have that

$$[\sum_U S_U \frac{\partial h_0}{\partial f_m} \prod_{g \in U} \psi^{g, d(\{f_0, \ldots, f_m\})}]$$

is undercomputed. Hence

$$\psi^{g_i, d(\{f_0, \ldots, f_m\})} \supseteq h_0[g_i]\texttt{.psi}$$

for some set of side inputs g_i, and each $g_i \neq f_m$. Hence there exist viable paths, terminating in one of the g_i, of length $\geq d(\{f_0, \ldots, f_m\})$ that had not been traced when P was extended. Each path of length $> d(\{f_0, \ldots, f_m\})$ had been traced, and hence each such path is of length $d(\{f_0, ..., f_m\})$. Let $Q_j = \{g_{0j}, ..., g_i\}$ be the set of such paths. Now, by theorem 3.3.2 one of the paths $\{Q_j, h_0, ..., h_p\}$ is the long viable path reported by the algorithm. This path satisfies the assumptions of the path Q of the theorem, and so done. ■

3.4 Dynamic Programming Algorithm Example

We now demonstrate an example of the dynamic programming procedure in action.

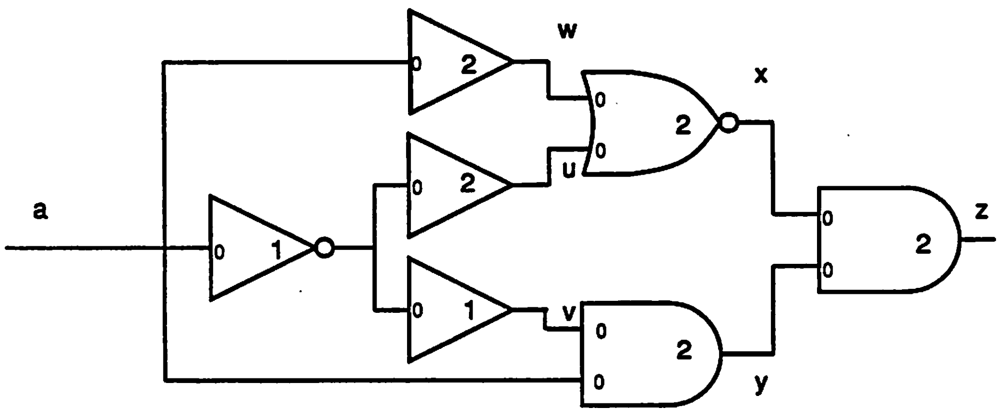


Figure 3.19: Circuit on Algorithm Entry

Consider the circuit of figure 2.3. We analyze this circuit using the algorithm of figure 3.17 and figure 3.18. For this circuit, we have $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} = 0$, $\frac{\partial x}{\partial u} = \frac{\partial y}{\partial a} = \overline{a}$, $\frac{\partial x}{\partial w} = \frac{\partial y}{\partial v} = a$, $\frac{\partial u}{\partial a} = \frac{\partial v}{\partial a} = 1$. Since each gate in this circuit is one or two input, note that the viability equations reduce to

$$\psi_P^{f_i} = \frac{\partial f_i}{\partial f_{i-1}} + \psi^{g,\pi_{i-1}}$$

We represent the variable `g[k].psi` at the k input connection of gate g. For example, in figure 3.19 the variable `x[w].psi` is explicitly indicated. The value of each such variable is 0 at the entry of the algorithm, as indicated in the figure. For convenience we have explicitly represented the delay buffers in this example.

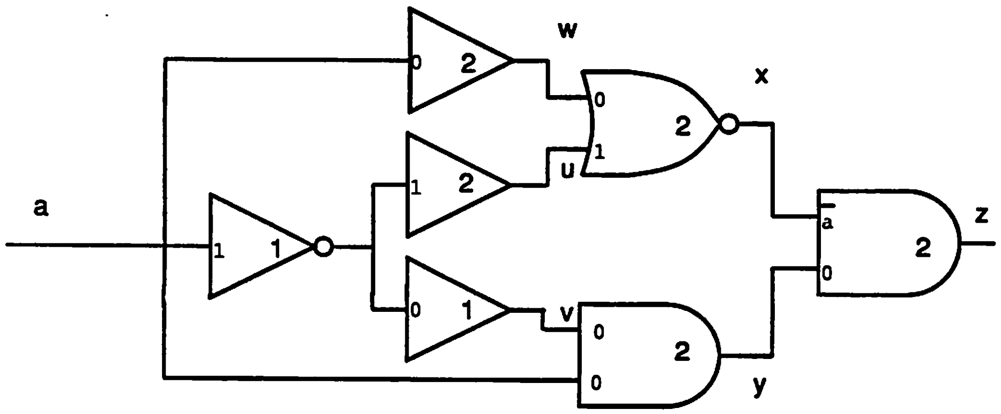


Figure 3.20: Circuit After {a,u,x,z} Explored

The algorithm first attempts to extend the path a through u, ORing the viability function of the partial path $\{a\}$ (1) into the variable u[a].psi[2]; as the extension through u succeeds with viability function 1, the algorithm sets x[u].psi = 1. The extension through x succeeds with viability function $\overline{a}$, and z[x].psi is set to $\overline{a}$. The extension through z fails, and the circuit now appears as in figure 3.20.

The algorithm now explores the path $\{a, v, y, z\}$. Again, $\{a, v, y\}$ is a viable path with viability function a. The algorithm now attempts to extend y to z. The function $\psi^{z}_{\{a,v,y\}}$ is $\frac{\partial z}{\partial y} + \bar{a}$; hence $\psi_{\{a,v,y,z\}} = a(0 + \bar{a}) = 0$, and the extension fails, leaving the circuit in figure 3.21.

The algorithm now explores the path $\{a, w, x, z\}$. $\{a, w, x\}$ is a viable path with viability function 1. The algorithm now attempts to extend x to z. The function $\psi^{z}_{\{a,w,x\}}$ is $\frac{\partial z}{\partial x}+a$; hence $\psi_{\{a,w,x,z\}} = 1(0+a) = a$, and the extension succeeds, reporting the longest viable path as $\{a, w, x, z\}$, a path of length 6. The final circuit is as appears in figure 3.22

[2]for convenience, u[a].psi and v[a].psi are represented in the same variable

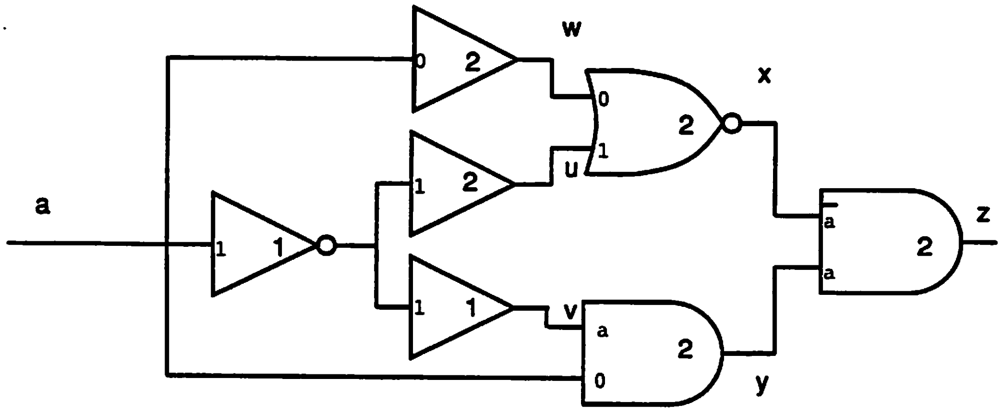


Figure 3.21: Circuit After {a,v,y,z} Explored

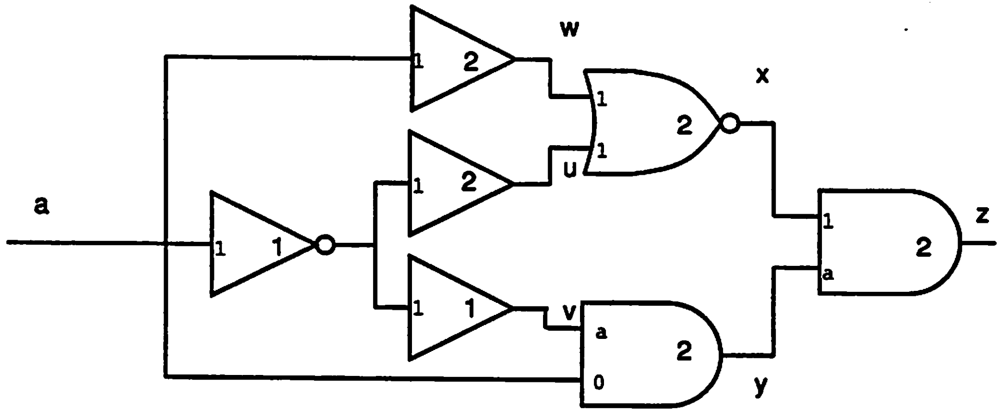


Figure 3.22: Circuit After {a,v,y,z} Explored

3.5 Modifying the Dynamic Programming Procedure to find all the Longest Viable Paths

The improved dynamic programming procedure described in the preceding sections has the flaw that only one of the longest viable paths is reported; if all the longest viable paths are to be reported, then the original dynamic programming procedure is preferred. As mentioned above, we often wish a procedure which returns all the longest paths at or above a given length. This procedure can be easily obtained by a simple modification of the first dynamic programming procedure, given in figure 3.23.

This procedure returns all viable paths of length $\geq T$, for some T. T is either directly set, or is chosen to be some ϵ less than the length of the longest viable path; hence choosing $\epsilon = 0$ finds all the longest viable paths.

If search-by-ϵ is selected, the initial threshold is set to 0; this simply ensures that no path will fail to meet the threshold test. When a viable path is found, if the threshold is still 0 then this is the first viable path found; the threshold is set to the length of this path $-\epsilon$.

Whenever a full viable path is found, it is added to `long_paths`. By construction, this path is viable and is of length $\geq T$. When no more paths of potential full length $\geq T$ remain to be explored, the algorithm terminates and `long_paths` is returned.

By construction it is easy to see that only viable paths of length $\geq T$ are returned. To see that all such paths are returned, the similarity of this procedure and the procedure of figure 3.15, together with theorem 3.3.1 suffice.

The complexity of this algorithm is $O(KDS \log D)$, where D is the diameter of the graph, S is the cost of a SAT call, and K is the number of paths, false and true, of length $\geq T$. It is easy to see that this is the minimum number of paths which must be examined by any procedure which purports to solve this problem.

```
find_longest_true_path(){
    Initialize queue to primary inputs of the circuit
    long_paths = $\emptyset$;
    if search_by_epsilon $T = 0$;
    while((((path, g) <- pop(queue) ) $\neq$ nil) and ($E$(path, g) $\geq T$)) {
        k is the last node of path;
        g[k].psi <- g[k].psi + $\psi$(path);
        $\psi$ <- viability_function(path, g)
        if($\psi \not\equiv 0$) {
            new_path <- {path, g};
            if(g is an output) {
                if (search_by_epsilon and $T = 0$)
                    T = length(new_path) - $\epsilon$;
                long_paths = long_paths $\bigcup$ {g};
            }
            else {
                $\psi$(new_path) <- $\psi$;
                foreach extension np <- {new_path, h} of new_path
                    if($E$(np) = $E$(new_path))
                        insert np on queue
            }
        }
        if every extension ext of path s.t.
            $E$(ext) = $E$(new_path) has been explored {
            ext1 is next best extension of path
            push ext1 on queue;
            foreach extension ext2 of path with $E$(ext1) = $E$(ext2)
                push ext2 on queue;
        }
    }
return long_paths;
}
```

Figure 3.23: Dynamic Programming Procedure to Find All the Long Viable Paths

Chapter 4

System Considerations and Approximations

The theory and algorithms described to this point capture the nature of the problem. Nevertheless, the viability procedure is enormously expensive; the inner loop of the procedure is a general satisfiability problem, and hence is strongly suspected to be of exponential complexity; further, there appears to be no polynomial upper bound on the size of the viability function. In general, some applications may prefer a faster answer and a poorer approximation to the longest viable path, so long as the assurance is given that such an approximation will not *underestimate* the length of the longest viable path. In this section we explore such performance/quality tradeoffs, and end by giving a polynomial approximation to viability.

4.1 Approximation Theory and Practice

The algorithms developed in the previous chapter removed the unnecessary inefficiencies in computing the viable set. However, the necessity of computing a sum over the power set of the inputs at each node remains a potentially expensive operation. This operation appears unavoidable for exact computation of the viable sets for the network. However, approximate solutions may be found. Of course, these approximations must be conservative: each approximation, to be valid, must be shown to upper bound the delay down the longest (dynamically) sensitizable path, and must be shown to obey monotone speedup. In this section we explore such approximation techniques.

In previous chapters, we have noted the duality of network or path conditions and some boolean functions, to which we should give a name; let us call these *path logic functions.* Since every algorithm which attempts to solve the false path problem explicitly or implicitly associates a logic function Γ_P with each path P and computes its satisfying set, it seems that one way to validate (or not) such algorithms is to consider the properties of its associated family of logic functions. The most clearly interesting property is whether or not the function is *correct* in the obvious sense: if one accepts the critical delay as the longest path P such that Γ_P is satisfiable, then no longer path P' may transmit an event in this or any faster network. We formalize this in the following definition.

Definition 4.1.1 *A family of path logic functions Γ_P on a boolean network N is said to be* **critical path correct** *iff the longest path P such that Γ_P is satisfiable is longer than the longest dynamically sensitizable path in any network N', where N' is obtained from N by reducing some or none internal delays.*

Our familiar properties of dynamic sensitizability and monotone speedup may be expressed as properties of path logic functions:

Definition 4.1.2 *A family of path logic functions Γ_P on a boolean network N is said to have the* **dynamic sensitization property** *iff Γ_P is satisfiable for every dynamically sensitizable path P.*

Definition 4.1.3 *A family of path logic functions Γ_P on a boolean network N is said to have the* **monotone speedup property** *iff for each satisfiable $\Gamma_{P'}$ in a sped-up network N' there is a path P in N such that $d_N(P) \geq d_{N'}(P')$ and Γ_P is satisfiable.*

These definitions may seem like old wine in new bottles, inasmuch as these are merely near-repetitions of our earlier definitions. However, most find old wine highly palatable, and occasionally new bottles permit us to see the body more clearly. We will try to peer through the new clear glass now.

One fact that is immediately apparent is that the two enumerated properties on a family of path logic functions are *not* required for critical path correctness. While it is true that monotone speedup and dynamic sensitization together imply critical path correctness (as we have seen), the converse is not true. The following theorem gives us the family of counterexamples.

Theorem 4.1.1 (Approximation Theorem) *Let Γ_P be a critical path correct family of path logic functions. Every family of path logic functions $\widehat{\Gamma_P}$ .s.t. at least one of the following holds:*

1. *$\widehat{\Gamma_P} \supseteq \Gamma_P$, for every P or*
2. *for every satisfiable P s.t. Γ_P is satisfiable there is a P' with $d(P) \leq d(P')$ s.t. $\widehat{\Gamma_{P'}}$ is satisfiable*

is also critical path correct.

Proof: The proof is almost a triviality. If $\widehat{\Gamma_P} \supseteq \Gamma_P$ for every P, and Q' is the longest sensitizable path in any of the family of networks (Q' is sensitizable in some N'), then we must have some Q s.t. $d_N(Q) \geq d_{N'}(Q')$ and Γ_Q is satisfiable. Since $\widehat{\Gamma_Q} \supseteq \Gamma_Q$, we have that $\widehat{\Gamma_Q}$ is satisfiable, and so $\widehat{\Gamma_P}$ is critical path correct. For the second item, we have Q, N, Q', N' and Γ_Q satisfiable as before. Since Γ_Q is satisfiable, there is Q'' s.t. $d(Q'') \geq d(Q)$ and $\widehat{\Gamma_{Q''}}$ is satisfiable, and since $d_N(Q'') \geq d_N(Q) \geq d_{N'}(Q')$ we have that $\widehat{\Gamma_P}$ is critical path correct. ■

Examples of functions which are critical path correct but which do not have one, or both, of the two critical properties abound. For example, the (trivial) family of functions

$$\Xi_P = \begin{cases} 1 & P \text{ is the longest path in the network} \\ 0 & \text{otherwise} \end{cases}$$

obviously does not have the dynamic sensitization property, but is critical path correct. Similarly, it seems likely that there are families of functions which do not obey monotone speedup but which are critical path correct.

On the other hand, function families which do not satisfy the two properties but which are critical path correct seem to be fated to be relatively weak upper bounds, unless there is some further theory to be discovered here. The only tool one has for proving such a function family correct is theorem 4.1.1, for which one must have shown the existence of another correct function family which yields a better result, and so one can always do better by computing the latter. This theorem has its uses, however. First, we may analyze and prove correct existing algorithms. Second, though we have no rigorous proof, this theorem and its associated discussion leads us to the belief that one can do no better than viability. Third, the viability function is expensive to compute, since it involves computing a sum over the power set of the side inputs at every node; indeed, even if the sum-of-products expression for the viability function at each node involved only two terms, it is easy to see that the function on some path P could grow as large as 2^D, where D is the diameter of the graph. This theorem permits us to consider other, easier-to-compute functions which satisfy the assumptions of theorem 4.1.1. We turn to the first of these, an analysis of the weak viability procedure.

4.2 "Weak" Viability

Recall the viability equations (2.2)-(2.4)

$$\begin{aligned}
\psi_P &= \prod_{i=0}^{m} \psi_P^{f_i} \\
\psi_P^{f_i} &= \sum_{U \subseteq S(f_i,P)} \left(S_U \frac{\partial f_i}{\partial f_{i-1}}\right) \prod_{g \in U} \psi^{g,\tau_i - 1} \\
\psi^{g,t} &= \sum_{Q \in \mathcal{P}_g, d(Q) \geq t} \psi_Q
\end{aligned}$$

The heart of the definition is equation (2.3)

$$\psi_P^{f_i} = \sum_{U \subseteq S(f_i,P)} \left(S_U \frac{\partial f_i}{\partial f_{i-1}}\right) \prod_{g \in U} \psi^{g,\tau_i - 1}$$

The power-set sum in this equation is also the heart of the inefficiencies inherent in the definition. The expression cannot be immediately simplified, for no term of

$$\sum_{U \subseteq S(f_i,P)} (S_U \tfrac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \psi^{g,\tau_i - 1}$$

inherently covers any other. Indeed, for $V \subset U$, we have:

$$S_V \tfrac{\partial f_i}{\partial f_{i-1}} \subseteq S_U \tfrac{\partial f_i}{\partial f_{i-1}}$$

but also:

$$\prod_{g \in V} \psi^{g,\tau_i - 1} \supseteq \prod_{g \in U} \psi^{g,\tau_i - 1}$$

However, if we weaken the definition of viability slightly:

Definition 4.2.1 *The* **weak viability function** *of a path $P = \{f_0, ..., f_m\}$ is defined as:*

$$\varphi_P = \prod_{i=0}^{m} \varphi_P^{f_i} \tag{4.1}$$

where

$$\varphi_P^{f_i} = \sum_{U \subseteq S(f_i,P)} (S_U \tfrac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \varphi^{g,\tau_i - 1} \tag{4.2}$$

and

$$\varphi^{g,t} = \begin{cases} 0 & \sum_{Q \in \mathcal{P}_{g,t}} \varphi_Q = 0 \\ 1 & \textit{otherwise} \end{cases} \tag{4.3}$$

Paths P such that $\varphi_P \not\equiv 0$ are said to be weakly viable.

We have immediately:

Theorem 4.2.1 *For every path P, $\varphi_P \supseteq \psi_P$.*

Proof: Let $P = \{f_0, ..., f_m\}$. Induction on $\delta(f_m)$. If f_m is a primary input, trivial. Suppose $\varphi_Q \supseteq \psi_Q$ for paths $Q = \{g_0, ..., g_n\}$ with $\delta(g_n) < N$. If $\delta(f_m) = N$, the result holds if we can show:

$$\varphi_P^{f_m} \supseteq \psi_P^{f_m}$$

But this is immediate, for we have that:

$$\varphi_P^{f_m} = \sum_{U \subseteq S(f_m,P)} (S_U \frac{\partial f_m}{\partial f_{m-1}}) \prod_{g \in U} \varphi^{g,\tau_{m-1}}$$

Since by induction we have that:

$$\varphi_Q \supseteq \psi_Q$$

for each Q terminating in a fanin of f_m, we must have that:

$$\sum_{Q \in \mathcal{P}_{g,t}} \varphi_Q \supseteq \sum_{Q \in \mathcal{P}_{g,t}} \psi_Q$$

Hence if $\psi^{g,\tau_{m-1}} \not\equiv 0$ we must have $\varphi^{g,\tau_{m-1}} = 1$, whence $\varphi^{g,\tau_{m-1}} \supseteq \psi^{g,\tau_{m-1}}$ for each g, and hence we may write:

$$\varphi_P^{f_m} \supseteq \sum_{U \subseteq S(f_m,P)} (S_U \frac{\partial f_m}{\partial f_{m-1}}) \prod_{g \in U} \psi^{g,\tau_{m-1}}$$

and the right-hand side is obviously $\psi_P^{f_m}$, whence the result. ■

This serves to show that φ is a critical-path correct path function on symmetric networks, by the approximation theorem. The attractive thing about φ is the following observation:

Theorem 4.2.2 *Let $P = \{f_0, ..., f_m\}$. Let V be the maximal set of g s.t. $\varphi^{g,\tau_{i-1}} = 1$. Then*

$$S_V \frac{\partial f_i}{\partial f_{i-1}} = \varphi_P^{f_i}$$

Proof: From (4.2):

$$\varphi_P^{f_i} = \sum_{U \subseteq S(f_i,P)} (S_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \varphi^{g,\tau_{i-1}}$$

Now, we have that

$$S_V \frac{\partial f_i}{\partial f_{i-1}} \prod_{g \in V} \varphi^{g,\tau_{i-1}}$$

is a term of this series, and since $\varphi^{g,\tau_{i-1}} = 1$ for all $g \in V$ this is

$$S_V \frac{\partial f_i}{\partial f_{i-1}}$$

whence

$$\mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}} \subseteq \varphi_P^{f_i}$$

Further, consider an arbitrary term of (4.2), say:

$$(\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \varphi^{g, \pi_{i-1}}$$

Now, if $U \subseteq V$, this is certainly $\subseteq \mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}}$. However, if $U \supset V$, then by construction there is some $g \in U$ such that $\varphi^{g,\pi_{i-1}} = 0$, by choice of V. Hence we must have that:

$$(\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \varphi^{g, \pi_{i-1}} = 0$$

and so

$$(\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \varphi^{g, \pi_{i-1}} \subseteq \mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}}$$

and hence:

$$\mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}} \supseteq \varphi_P^{f_i}$$

and therefore:

$$\mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}} = \varphi_P^{f_i}$$

■

In other words, only a single term of the series (4.2) (the last nonzero term) need be taken. While this can still lead to an exponential blowup in the size of the function, it seems far less likely. Further, under some circumstances one can guarantee that the size of φ_P will remain bounded (in fact, one can guarantee that φ_P will consist of a single cube). We detail these circumstances here.

Theorem 4.2.3 *Let N be any network such that for any node f, any input x of f, $\frac{\partial f}{\partial x}$ is a single cube. Then for each path P through the network, φ_P is a single cube*

Proof: For the proof, we note that:

$$\varphi_P = \prod_{i=0}^{m} \varphi_P^{f_i}$$

where

$$\varphi_P^{f_i} = \sum_{U \subseteq S(f_i,P)} (S_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \varphi^{g,\tau_{i-1}}$$

now, from theorem 4.2.2, we know that this last can be written

$$\varphi_P^{f_i} = S_V \frac{\partial f_i}{\partial f_{i-1}}$$

Where V be the maximal set of g s.t. $\varphi^{g,\tau_{i-1}} = 1$. Further, for any function h, any set of inputs V, if h is a single-cube function then so is $S_V h$, hence $\varphi_P^{f_i}$ is a single-cube function for each f_i. The product of single-cube functions is a single-cube function, and so done. ■

Weak viability is an attractive alternative to viability when computation time is at a premium, and can be used in conjunction with other approximation techniques. For example, the weak-viability function is therefore a single cube for a network consisting only of NOR, NAND, OR, AND, and NOT gates, or some subset thereof. One can always transform *any* network into such a network, through a variety of macroexpansion transformations, and guarantee correctness through theorem 2.5.1; one can then use weak viability on the transformed network.

It is interesting to formally define the set of paths P for which $\varphi_P \neq 0$. We give these here.

Definition 4.2.2 *A path $P = \{f_0, ..., f_m\}$ is said to be* **weakly viable** *under an input cube c if, at each node f_i there exists a (possibly empty) set of side inputs $U = \{g_1, ..., g_n\}$ to P at f_i, such that, for each j,*

1. *g_j is the terminus of a path Q_j,*
2. *$d(Q_j) \geq \tau_{i-1}$ and Q_j is weakly viable*
3. *$(S_U \frac{\partial f_i}{\partial f_{i-1}})(c) = 1$*

Notice that the definition of weakly viable paths differs only marginally from that of viable paths; the principle difference is that no record is kept of the cube under which the side path is viable (item 2).

We must formally show the equivalence of the function and the definition.

Theorem 4.2.4 *$P = \{f_0, ..., f_m\}$ is weakly viable under some minterm c iff c satisfies φ_P*

Proof: $\Longrightarrow$ $P = \{f_0, ..., f_m\}$ is weakly viable under c. Induction on $\delta(f_m)$. The base case is trivial, so assume for $\delta(f_m) < L$. Let $\delta(f_m) = L$. We must show that for each f_i, $c \in \varphi_P^{f_i}$. Now, if $c \in \frac{\partial f_i}{\partial f_{i-1}}$, done. Otherwise, since the path is weakly viable under c, then we must have that there is a subset $U = \{g_1, ..., g_k\}$ of $S(f_i, P)$, where each g_j terminates a path Q_j with $d(Q_j) \geq \tau_{i-1}$ and Q_j is weakly viable. By induction, $\varphi_{Q_j} \neq 0$. Hence $\varphi^{g_j, \tau_{i-1}} = 1$ for every j by 4.3. Further, $c \in S_U \frac{\partial f_i}{\partial f_{i-1}}$, and done.
$\Longleftarrow$ $c \in \varphi_P$. Induction on $\delta(f_m)$. The base case is trivial, so assume for $\delta(f_m) < L$. Let $\delta(f_m) = L$. We must show that the definition of weak viability holds for each f_i. Now, if $c \in \frac{\partial f_i}{\partial f_{i-1}}$, done. Otherwise, we must show that there exists a set of side inputs U meeting the conditions of the definition of weak viability with $S_U \frac{\partial f_i}{\partial f_{i-1}} \supseteq c$. Since $c \in \varphi_P^{f_i}$, then we must have that there is a subset $U = \{g_1, ..., g_k\}$ of the side inputs, and for each g_j, $\varphi^{g_j, \tau_{i-1}} = 1$. Now, by the definition of $\varphi^{g_j, \tau_{i-1}}$, we must have for each j there must be a path Q_j terminating in g_j weakly viable with $d(Q_j) \geq \tau_{i-1}$. $\delta(g_j) < L$, so by induction Q_j is weakly viable, and $c \in S_U \frac{\partial f_i}{\partial f_{i-1}}$, and so done. ∎

The dynamic programming procedure of figures 3.15-3.16 is easily adapted to compute the weak viability function. For the line:

```
    g[k].psi <- g[k].psi + ψ(path);
```

is replaced by

```
    g[k].psi <- 1;
```

Further, the improved dynamic programming procedure of figures 3.17-3.18 is adapted in precisely the same fashion.

4.3 The Brand-Iyengar Procedure

Brand and Iyengar have published a solution to the false path problem[10]. We have not discussed this approach in much detail yet, since we wished to have the mathematical tools in place to establish the correctness of their procedure, and to show that it gives an upper bound on the procedure developed in this paper.

The Brand-Iyengar procedure is much like the later procedure of Benkoski, et al, save that the paths are traced depth-first from the output, rather than from the inputs. A further and more important difference is the sensitization criterion. Brand and Iyengar realized the difficulty with static sensitization, referring to the errors that a straight static sensitization approach would yield as due to "a sort of circularity". Though they did not fully analyze this difficulty, they used a strategy which returns correct, though suboptimal results.

At each node f, Brand and Iyengar number the inputs to the node; for purposes of illustration, let us call these $x_1, ..., x_k$. If the current path under consideration proceeds from input x_j, then inputs $x_1, ..., x_{j-1}$ are ignored. Effectively, the Brand-Iyengar sensitizing function at node f may be written:

$$\xi^f_{x_j} = S_{x_1,...,x_{j-1}} \frac{\partial f}{\partial x_j}$$

and for the path $P = \{f_0, ..., f_m\}$:

$$\xi_P = \prod_{i=1}^{m} \xi^{f_i}_{f_{i-1}}$$

We call paths P for which ξ_P is non-zero *Brand-Iyengar* paths.

Now, note that the Brand-Iyengar function is not a cover of the weak viability function. Nevertheless, we may show that this procedure is correct using theorem 4.1.1. We do this by showing:

Theorem 4.3.1 *For each path $P = \{f_0, ..., f_m\}$ weakly viable under c in a symmetric network N, there is a Brand-Iyengar path $P' = \{g_0, ..., f_m\}$ with $\xi_{P'}(c) = 1$ and $d(P) \leq d(P')$.*

Proof: Induction on $\delta(f_m)$. The base case is trivial, so assume for $\delta(f_m) < L$. Consider the case $\delta(f_m) = L$. Now, consider the set of paths weakly viable under c which terminate in an input to f_m of length at least τ_{m-1}. These paths terminate in a set of inputs to f_m, $U = \{h_0, ..., h_k\}$. Now, by induction, each such h_i terminates a Brand-Iyengar path of length $\geq \tau_{m-1}$. One of the h_i is f_{m-1}, and one is maximal under the Brand-Iyengar ordering. (The Brand-Iyengar ordering is any ordering of the inputs at all). Let h be the maximal element of U under the ordering, and V be the set of inputs to f_m which precede h in the standard order, including h. We claim that c satisfies $\xi_h^{f_m}$. Now, if $h = f_{m-1}$, then $V \supseteq U$, and

$$\xi_h^{f_m} = \xi_{f_{m-1}}^{f_m} = \mathcal{S}_{V-\{f_{m-1}\}} \frac{\partial f_m}{\partial f_{m-1}}$$

Since $V - \{f_{m-1}\} \supseteq U - \{f_{m-1}\}$, and since $\mathcal{S}_R f \supseteq \mathcal{S}_S f$ for any sets R, S such that $R \supseteq S$ and any function f, we have that:

$$\xi_{f_{m-1}}^{f_m} \supseteq \varphi_P^{f_m}.$$

and since c satisfies $\varphi_P^{f_m}$, c must satisfy $\xi_{f_{m-1}}^{f_m} = \xi_h^{f_m}$. If $f_{m-1} \neq h$, then we can do a similar calculation: the set $U - \{f_{m-1}\} \subseteq V - \{h\}$, since h is the maximal element of U under the ordering, and hence every other element of U must precede it in the ordering. Since f_m is symmetric, we have that

$$\mathcal{S}_{U-\{f_{m-1}\}} \frac{\partial f_m}{\partial f_{m-1}} = \mathcal{S}_{U-\{h\}} \frac{\partial f_m}{\partial h}$$

and since $V \supseteq U$, we therefore have

$$\mathcal{S}_{U-\{f_{m-1}\}} \frac{\partial f_m}{\partial f_{m-1}} \subseteq \mathcal{S}_{V-\{h\}} \frac{\partial f_m}{\partial h}$$

and since this function is satisfied by c, we have that $\xi_h^{f_m} = \mathcal{S}_{V-\{h\}} \frac{\partial f_m}{\partial h}$ is satisfied by c, and so in either case the claim is shown. Once the claim is given, done, since we have by the inductive assumption a Brand-Iyengar path P'' terminating in h s.t $\xi_{P''}(c) = 1$, of length $\geq \tau_{m-1}$. The path $P' = \{P'', h\}$ clearly has $\xi_{P'}(c) = 1$, and $d(P') \geq \tau_{m-1} + w(f_m) \geq d(P)$, and done. ∎

From this, we may immediately conclude from theorem 4.1.1 that the Brand-Iyengar procedure is critical-path-correct for all symmetric networks, since it is an approximation to weak viability, a known critical-path-correct criterion. This is a somewhat stronger result than Brand and Iyengar proved in their paper (their proof was only valid for gates whose values could be controlled by a single input); further, the Brand-Iyengar proof of correctness relied on an induction whose base case was both non-trivial and not properly established, and so there is some doubt as to the correctness of their proof. This result firmly establishes the correctness of their procedure. Further, Brand and Iyengar made no mention of the robustness requirement for false path elimination, and hence did not prove that their criterion was robust. This proof demonstrates that. It also, however, guarantees that the critical delay reported by the Brand-Iyengar procedure will be an upper bound on that returned by the weak viability procedure.

The bound returned by the Brand-Iyengar procedure is highly dependent on the variable ordering chosen. Consider the path in the graph that runs through the last fanin (in the standard order) to each node. The Brand-Iyengar function for this path $\{f_0, ..., f_m\}$ is

$$\prod_{i=1}^{m} \mathcal{S}_{S(f_i,P)} \frac{\partial f_i}{\partial f_{i-1}}$$

Now,

$$\mathcal{S}_{S(f_i,P)} \frac{\partial f_i}{\partial f_{i-1}} = 1$$

for every i, whence the Brand-Iyengar function for this path is 1; i.e., the path is always true by the Brand-Iyengar criterion. If the standard order is increasing in the maximum distance of a node from the primary inputs, then this path will be the longest in the graph. In other words, for every network, there is a variable order such that the longest path is true by the Brand-Iyengar criterion. This order is the worst possible for this criterion.

Better orders may be chosen; in fact, a good order is probably the reverse of the bad order for this path. This order (decreasing in distance from the primary inputs) is the one recommended by Brand and Iyengar in their paper; however, they seemed to view this as an efficiency issue rather than an issue of quality of results. They used a depth-first search with pruning, and pointed out that if an algorithm had explored the long paths first and found one true, then the search space is drastically trimmed.

In any case, the best order for this function – whatever it may be – will at best equal the weak viability criterion.

4.4 The Du-Yen-Ghanta Criteria

Other work on this phenomenon has recently emerged, most notably the work due to Du, et. al. [24]. These authors considered networks composed of simple gates, for which each input can assume either a *control* value (a value which determines the value of the gate), or a *non-control* value (broadly, an identity for the gate – e.g., 0 for OR or NOR, 1 for AND or NAND). Static sensitization can be viewed, on such networks, as asserting a non-control value on each side input of each gate along the path.

When tracing a path $P = \{f_0, ..., f_m\}$, Du et. al split the side inputs of f_i into two sets:

1. Early-arrive-signals(f_i): Those inputs g such that the length of the longest path terminating in g is $< \tau_{i-1}$; and

2. Late-arrive-signals(f_i): Those inputs g such that the length of the shortest path terminating in g is $> \tau_{i-1}$.

Note that while the intersection of these sets is always empty, their union is not necessarily equal to the set of side inputs to f_i.

These two sets may be thought of as follows: Early-arrive-signals is the set of signals that have settled to their final value before the event propagates to f_i; late-arrive-signals are those signals which undergo events only *after* τ_{i-1}.

The Du criterion encompasses two rules:

1. If $g \in$ Early-arrive-signals(f_i), assert a non-control value on g.

2. If Late-arrive-signals(f_i) $\neq \emptyset$, assert a *control* value on f_{i-1}

The rationale behind the first rule is by now familiar to most readers. The side inputs $g \in$ Early-arrive-signals(f_i) have already settled to their final values at τ_{i-1}, and hence must satisfy $\frac{\partial f_i}{\partial f_{i-1}}$; i.e., must have assumed non-controlling values. Note that this rule is equivalent to taking the static boolean difference and then smoothing off all the side inputs *not* in Early-arrive-signals(f_i).

It is the second rule that distinguishes this transformation. Du et al reasoned that if the path $\{f_0, ..., f_m\}$ was a longest true path, then every longer path through f_i must have been rejected as false. If $g \in$ Late-arrive-signals(f_i), then every path through g must have been rejected as false. In this case, they reasoned, a controlling value must have been asserted on the wire f_{i-1}.

Now, notice that every connection in Late-arrive-signals(f_i) must be untestable. For suppose $g \in$ Late-arrive-signals(f_i) is testable. By definition, if $g \in$ Late-arrive-signals(f_i) then no event on g can propagate to the primary outputs, otherwise there would be a longer sensitizable path running through g (since every path through g is longer than the current path). Now, if g is testable, then the value of some primary output is determined by the value of g under c; i.e, the value of g has propagated to the primary output. The value of g is also its last event, and so the last event on g has propagated to the primary output. This event must have travelled down some sensitizable path from g, contradiction. Hence g must be untestable. In a fully-testable network, therefore, only rule (1) need be considered. We call this the *weak Du* criterion.

Lemma 4.4.1 *Consider the Brand-Iyengar criterion, with the inputs to every gate ordered in decreasing order by static delay (length of longest path). Every path true by this criterion is also true by the weak Du criterion, and hence the delay estimate produced by this procedure is a lower bound on the delay estimate given by the weak Du procedure.*

Proof: Let $P = \{f_0, ..., f_m\}$ be a path true by the Brand-Iyengar criterion under this ordering. We claim that P is true by the weak Du criterion. Induction on $\delta(f_m)$. Trivial for $\delta(f_m) = 0$. Now suppose the claim holds for $\delta(f_m) < N$. If $\delta(f_m) = N$, by induction the statement holds for $\{f_0, ..., f_{m-1}\}$. We must show that it holds for $\{f_0, ..., f_m\}$. Let f_{m-1} be the kth input of n to f_m under the ordering. Now, each input of order $< k$ terminates a path of length $\geq \tau_{m-1}$ by the definition of the order, no such input is in Early-arrive-signals(f_m). Hence under the weak Du criterion each input in the range $1, ..., k$ is left unspecified, i.e., smoothed off. Hence the set of signals left unspecified by the weak Du criterion is a superset of those left unspecified by Brand-Iyengar under this order; i.e., the path logic function corresponding to the weak Du criterion contains the Brand-Iyengar function under this order, giving the result. ■

Note that lemma 4.4.1 tells us that Brand-Iyengar is a lower bound on the strong Du criterion (that using both rules) only on fully-testable networks. On networks with redundant connections, the delay estimate given by the strong Du criterion may be unequal to that given by the weak Du criterion. In any case, it may be shown that the strong Du criterion is weaker than viability; this is shown in the following theorem, which demonstrates that, if there is a longest viable path $P = \{f_0, ..., f_m\}$, viable under c such that for node f_i Late-arrive-signals(f_i) is non-empty, then c sets f_{i-1} to a controlling value. This suffices to show that the strong Du criterion is an approximation to the viability criterion, and hence the strong Du criterion is a correct, robust criterion by the approximation theorem.

Theorem 4.4.1 *Let N be a symmetric network. Let $P = \{f_0, ..., f_m\}$ be a longest viable path through f_i, P viable under c. Let g be a side input to f_i such that every path through g is longer than τ_{i-1}. Then either f_{i-1} is set to value a by c such that*

$$\left.\frac{\partial f_i}{\partial g}\right|_{f_{i-1}=a} \equiv 0$$

or there is another viable path under c at least as long as P.

Note that if simple gates are assumed, this simplifies to the statement that f_{i-1} is set to a controlling value for f_i by c.

Proof: Assume that

$$\left.\frac{\partial f_i}{\partial g}\right|_{f_{i-1}=a} \not\equiv 0$$

Then we claim that for each $i \leq k \leq m$ we can construct a path P_k at least as long as P, viable under c, terminating in f_k. Induction on k. For $k = i$, let U be the set of inputs to f_i terminating viable paths under c of length $\geq \tau_{i-1}$. Now, consider the set of paths viable under c terminating in an input to f_i of length $\geq \tau_{i-1}$, excluding the path $\{f_0, ..., f_{i-1}\}$. Note this set is nonempty since the set of shortest paths through g are all viable under 1, and are of length $> \tau_{i-1}$. Since the set is nonempty, it has an element of least length, P_h, and P_h terminates in h. We claim that $\{P_h, f_i\}$ is viable under c. Consider the term of the viability series for P_h:

$$\mathcal{S}_{U-\{h\}}\frac{\partial f_i}{\partial h}\prod_{h'\in U-\{h\}}\psi^{h',|P_h|}$$

Since P_h is chosen to be minimal in the set, every member of U aside from h terminates a viable path of length $\geq P_h$, i.e. $\psi^{h',|P_h|}$ is satisfied by c for every h' in $U-\{h\}$. Now, all we must show is that

$$\mathcal{S}_{U-\{h\}}\frac{\partial f_i}{\partial h}$$

is satisfied by c. Since f_i is symmetric, we have three cases:

1. $$\mathcal{S}_{U-\{h\}}\frac{\partial f_i}{\partial h}$$

   is satisfied by c; or

2. $$\mathcal{S}_U\frac{\partial f_i}{\partial f_{i-1}}$$

   is not satisfied by c; or

3. f_{i-1} is set to a value a by c such that

$$\mathcal{S}_{U-\{h\}}\frac{\partial f_i}{\partial h}\bigg|_{f_{i-1}=a}\equiv 0$$

Case 1 proves the claim, so we must dispose of (2) and (3). (2) is false, since if it were true, P would not be viable under c, contradiction. For case 3, since $\mathcal{S}_V f\equiv 0 \Rightarrow f\equiv 0$, we have that:

$$\frac{\partial f_i}{\partial h}\bigg|_{f_{i-1}=a}\equiv 0$$

But then, since f_i is symmetric:

$$\frac{\partial f_i}{\partial g}\bigg|_{f_{i-1}=a}\equiv 0$$

Contradiction. We are left with case (1), which proves the claim: $\{P_h, f_i\}$ is viable under c. Hence the statement of the theorem holds for the case $k=i$; the path

$\{P_h, f_i\}$ is viable under c, is of length $\geq \tau_i$ and is not equal to $\{f_0, ..., f_i\}$. Assume inductively that the statement holds for $k < K$. Consider the case $k = K$. By induction, we have a path P_{K-1} terminating in f_{K-1} viable under c at least as long as $\{f_0, ..., f_{K-1}\}$ that is *not* $\{f_0, ..., f_{K-1}\}$. Now, we have two cases:

1. f_K is statically sensitized to f_{K-1} by c, in which case the claim holds for f_K;

2. There are a set of paths U_K at least as long as $\{f_0, ..., f_{K-1}\}$ terminating in some side inputs to f_K. Of the set of paths $U_K \cup \{\{f_0, ..., f_{K-1}\}\}$ at least one had a viable extension under c through f_K. Hence the set of paths $U_K \cup \{P_{K-1}\}$ met the assumptions of lemma 2.4.2, and so by the terms of that lemma at least one of these paths has a viable extension under c through f_K. This path is not $\{f_0, ..., f_K\}$ and is of length $\geq \tau_K$, proving the theorem.

■

This theorem suffices to demonstrate that the strong Du criterion is an approximation to the viability criterion, since it demonstrates that the use of Du's second rule does not affect the viability of the longest viable path. Further, in practice, only a few connections will be untestable, and the second rule need not apply to all of those. Hence the second rule will be invoked only rarely. Hence, the strong Du and weak Du algorithms should only rarely give different bounds.

4.5 More Macroexpansion Transformations

Theorem 2.5.1 is valid for *any* macroexpansion transformation. We have only used it for the *and/or* transform, since it appears that this introduces the fewest *spurious* viable paths, and hence yields the lowest bound on the critical delay. However, it is possible that other transforms, though yielding poorer results, might well entail much less expensive computation. In particular, consider the *two-input nand* transform.

It is well-known that any function can be realized by a network of two-input nand gates. The two-input nand transform of a node f is any transform of f meeting the conditions of definition 2.5.1. This transform is attractive when one considers that there is at most one side input to a gate on any path $P = \{f_0, ..., f_m\}$ in a two-input network, and (if we represent the side input to node f_i as g_i), then the only subsets of the set of side inputs to f_i are $\emptyset$ and g_i. Hence g_i is the only possible input to $\frac{\partial f_i}{\partial f_{i-1}}$, and thus $S_{g_i} \frac{\partial f_i}{\partial f_{i-1}} = 1$. Given this, we can simplify the definition of the viability function for a path in such a network as:

$$\begin{aligned} \psi_P &= \prod_{i=0}^{m} \psi_P^{f_i} \\ \psi_P^{f_i} &= \frac{\partial f_i}{\partial f_{i-1}} + \psi^{g_i, \tau_{i-1}} \\ \psi^{g,t} &= \sum_{Q \in \mathcal{P}_{g,t}} \psi_Q \end{aligned}$$

This function is much simpler to compute than the viability function of definition 2.6.2, since the power-set sum in $\psi_P^{f_i}$ has disappeared. Moreover, the network is symmetric, and hence its correctness is guaranteed by the theorems previously proved. The reader is cautioned, however, that an increase in the number of paths may be expected in such a network, in which case some percentage of the gains realized may be offset.

This observation suggests that the algorithm presented above may be considered a *family* of algorithms, with different performance and result characteristics. The members of the family may be distinguished by the transform taken.

4.6 Biased Satisfiability Tests

Given that the algorithms enumerated so far all have a satisfiability test in the inner loop, it is unsurprising that these procedures take a very long time. Nevertheless, one may hope to use the approximation theorem to yield up a hint of polynomial-time approximations in this domain as well. Consider any positively-biased satisfiability test; i.e., a procedure which guarantees to report that a function f is nonzero when it is not identically zero, but may occasionally report that a function is nonzero when it is identically zero. Such a procedure may be considered an *exact* satisfiability test on a function $\hat{f} \supseteq f$. This consideration yields immediately the conclusion that if γ is any critical-path correct path logic function, then a procedure using γ as the sensitization criterion with a positively biased satisfiability test will also yield a critical path correct family of logic functions.

There are undoubtedly many such biased tests. We detail one here, and more fully in appendix C. Conceptually, one can think of such a test on a multi-level network as follows.

The functions we have enumerated above are, in general, functions of not only the primary inputs but also of the intermediate nodes in the network. From a theoretical point of view, this is a distinction without a difference; since the intermediate nodes themselves realize functions of the primary inputs, the functions we have been describing above must necessarily be functions of the primary inputs; moreover, that function may be (conceptually) realized by substituting the function (in terms of the primary inputs) for each intermediate node. This process is known as *collapsing.*

Mechanically, the way that functions are discovered to be 0 in the collapsing process is that for each cube of f, there is some x such that both x and $\overline{x}$ appear in the cube; i.e., for the cube to be satisfied we must have $x = 1$ and $x = 0$. Since this is clearly not possible, the cube is unsatisfiable. More generally, each such cube of the function is found to be inconsistent; no primary input vector gives rise to such a cube.

This gives us a clue as to how to proceed. Consider a procedure that directly simulates the effect of attaching values to the various wires in the circuit. Such a simulation will not be entirely complete, in the sense that the effect of the assertions will not be carried back to the primary inputs. Under such a simulation, a function is determined to be 0 if for each cube there is some x such that both x and $\overline{x}$ appear in the cube; i.e., each cube is *explicitly* inconsistent. Now, it is obvious that every cube that is explicitly inconsistent is inconsistent; however, the converse is not the case. Hence this is a positively-biased satisfiability test.

Such simulations appear to be done by Benkoski et al [6]; he uses a "D-Algorithm without justification", which presumably means computing implications. Brand and Iyengar also compute implications, and refer specifically to the NOR-gate rules formulated by Trevillyan, Berman, and Joyner[7]. We have reformulated those rules for general gates, and present an efficient algorithm for computing these in appendix C; this algorithm features a quartic preprocessing phase and a quadratic main phase. The difference in order is important, for only the quadratic main phase is called during path tracing.

4.7 Axes of Approximation

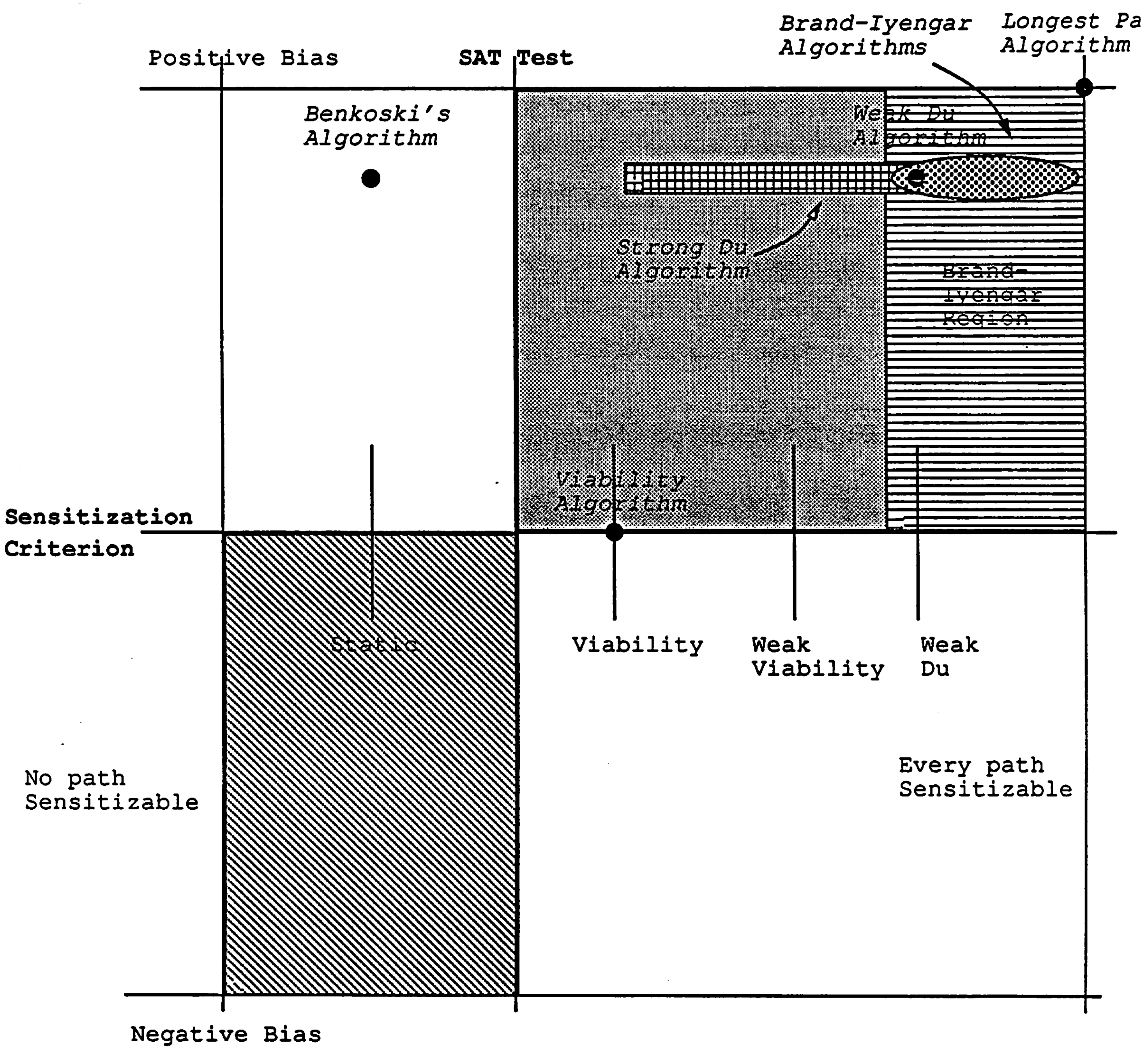
The theory developed above leads us to the conclusion that there are several dimensions of approximation, more or less orthogonal. Along one axis lies the sensitization criterion; along a second lies the satisfiability test. Along a third lies the macroexpansion transform. Because few other procedures use macroexpansions, we omit them in this picture.

The two major axes of approximation – sensitization criterion and satisfiability test – are depicted in figure 4.1. The x dimension represents sensitization criterion, and ranges from too restrictive (no paths sensitizable) to too loose (every path sensitizable). Similarly, the SAT test axis ranges from negative bias (all functions reported unsatisfiable) to positive bias (all functions reported satisfiable). Both dimensions increase in the direction of safety. The origin of the graph is the exact, minimum criterion. Note that no known sensitization criterion is at the origin.

Hence, programs depicted in the upper-right quadrant are known safe; they will always return an upper bound on the true delay. Programs in the lower-left quadrant are known unsafe; they will always return a lower bound on the true delay. Programs in the other quadrants give no guarantees at all.

4.8 The Llama Timing Environment

The observation of section 2 that the false path detection procedures were all parameterized variants of the same algorithm led to the development of a program which serves as an experimental testbed for these procedures. This system, the LLAMA timing environment, has been built on top of the MIS II logic synthesis system at UC-Berkeley [14, 12], and uses the underlying facilities of MIS for boolean function



Shaded Region is Safe; Striped is Known to be Wrong. Clear Regions are uncertain. Best Method is the origin.

Figure 4.1: Axes of Approximation

manipulation and for the extraction of delay information at the nodes. Further, the MIS II command interpreter is used to permit user-level experimentation with the parameters of LLLAMA.

LLLAMA is parameterized on five distinct axes: search method (best-first or depth-first), sensitization criteria (static, viability, or Brand-Iyengar), satisfiability test (test to determine whether the sensitization function is 0), representation of functions (Bryant's graph-based representation[18][53] vs sum-of-products form), and delay model. The selection of the various parameters is made at the beginning of a timing run by the user through MIS II command-line switches.

To date, we have experimented extensively with function representation and the various sensitization criteria. We have used two delay models in the calculations, a *unit* delay model, under which each gate has unit delay, and a *library* delay model. Under the latter, the network has been mapped to a network of standard cells, each of which has a well-characterized delay.

Before the main loop of the algorithm is entered, LLLAMA goes through a pre-processing phase. During this phase, a static delay trace is done to compute the delays and the esperance of each node, the static sensitization and Brand sensitization functions for each input of each gate are computed, and the variables `g[k].psi` are set to 0 for each input k to gate g. Further, asymmetric gates are macroexpanded into subnetworks of symmetric gates, if necessary. The data structure of partial paths is initialized to the set of primary inputs of the circuit.

4.9 Experimental Results

LLLAMA has been run on two broad classes of circuits: the public benchmark circuits, and parameterized circuits which are known to contain false paths. For each circuit, we report the critical delay according to the longest path procedure and the static, Brand-Iyengar, and viability conditions. We also report whether the circuit was optimized by the MIS-II standard script (O), and whether the circuit was mapped to the MSU standard-cell library (M). Mapped circuits have delays reported by the mapped model; unmapped circuits have delays reported by the unit model.

Ckt	O/M	Long	Brand	Viable	Static
5xpl	OM	21.80	21.80	20.40	19.20
5xpl	M	19.80	19.80	18.40	18.40
C7552		43.00	43.00	42.00	42.00
Des	O	11.00	11.00	10.00	10.00
Des	OM	68.20	68.20	66.40	64.00
Rot	O	10.00	10.00	9.00	9.00
Rot	OM	29.60	28.60	27.20	27.20

Table 4.1: Critical Delay of Benchmark Circuits

Bits	Block	O/M	Long	Brand	Viable	Static
8	2		13.00	13.00	8.00	8.00
8	2	M	17.80	15.40	15.40	15.40
8	4		11.00	11.00	10.00	10.00
8	4	M	14.80	13.60	13.60	13.60
16	2		25.00	25.00	12.00	12.00
16	2	M	35.40	27.40	27.40	27.40
16	4		21.00	21.00	12.00	12.00
16	4	M	29.20	20.00	20.00	20.00
32	4		41.00	41.00	16.00	16.00
32	4	M	58.00	32.00	32.00	32.00

Table 4.2: Critical Delay of Carry-Bypass Adders

Two sets of public benchmark circuits were run: the IWLS and ISCAS benchmark suites. Of the IWLS circuits, only the benchmark circuit 5xp1 showed any false paths under any criterion. The benchmarks DES and ROT were also run. Though not part of the IWLS benchmark suite, these circuits are available from UC-Berkeley in BLIF format. Of the ISCAS circuits, C880, C432, C499, and C17 all had no false paths. C7552 is shown in the table. The remainder not shown failed to complete.

It has long been known that carry-bypass adders exhibit false paths.[1] A final set of experiments involved the generation of carry-bypass adders of varying sizes, and block sizes for the bypass chain. Integers N and M in the first two columns of table 4.2 represents an N-bit adder, with M bits in the bypass chain. BLIF descriptions of these circuits, and a program to generate the BLIF description from arbitrary N and M, are available from the author.

[1]Prof. H. De Man kindly brought this fact to the attention of the author

Chapter 5

Hazard Prevention in Combinational Circuits

5.1 Introduction

Previous research into timing properties of circuits has led to considering the problem of *hazards* or *glitches* in combinational circuits. One can demonstrate that in the absence of hazards, a variety of strong properties hold which are not valid in the general case: in particular, in the next chapter we will show that timing analysis can obtain tight bounds on the critical path of a circuit, which was shown to be impossible for a hazardous circuit.

Given these desirable properties, it is worth considering whether certifiably hazard-free circuits may be synthesized. It is well-known that the class of *precharged unate* circuits (e.g., NORA, DOMINO, and DCVS) are hazard-free; indeed, these circuits can only function if they are hazard-free. The characteristics of these circuits are reviewed in appendix D. What we wish to discover is whether any fully-restoring circuits are hazard free.

The remainder of this chapter is organized as follows. In section 5.2, hazards will be defined and a set of assumptions concerning the way that signals change. In section 5.3 we relate function evaluation with walks on the Boolean n-cube, and show that every hazard-free circuit is precharged-unate, under our assumptions. In sections

5.4-5.5, we relax two of the initial assumptions and show that the results of section 5.3 hold even if either, or both, of these assumptions are relaxed.

5.2 Hazards

A *hazard* at a node is a multiple change in its value during an evaluation period (typically, say, a clock cycle or phase). Statically, we view a logic functions as an ideal switch, which remains at a value until some input has switched and then immediately switches to the new, output value. Further, in analyzing hazard-free circuits, we make the following assumptions about the behaviour of the inputs to the node:

1. The inputs to a node begin in some initial steady state and change, one at a time, until they reach some final steady state.
2. Each input to a node may change at most once during the evaluation of a node; this condition is assured if every node in, and every primary input of, the circuit is hazard-free
3. Each input to a node may change during evaluation.
4. The order in which the inputs change is unpredictable.
5. No combination of the inputs is forbidden as either the initial nor the final steady state.

These assumptions are strong in the sense that they enhance hazards. All but the last two are easily justified. The last is not only unjustifiable, it is generally false. In practice, only a few states can occur. Indeed, multi-level logic optimization makes heavy use of such forbidden states, which are vectors covered by the satisfiability don't-care set[3]. We will be relaxing this assumption later.

The fourth assumption is also a little shaky. In practice, some rough guesses can be made, though, as we shall see below, due to the statistical nature of delays in a MOS circuit, precise orders on variable arrival can only be guaranteed at some cost

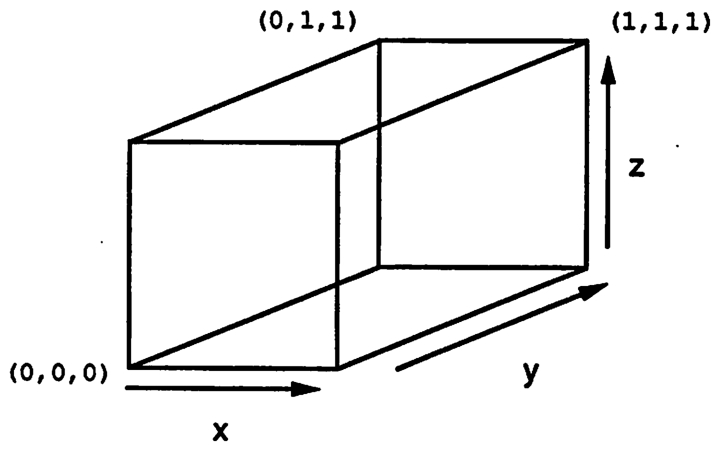


Figure 5.1: The Boolean 3 Cube

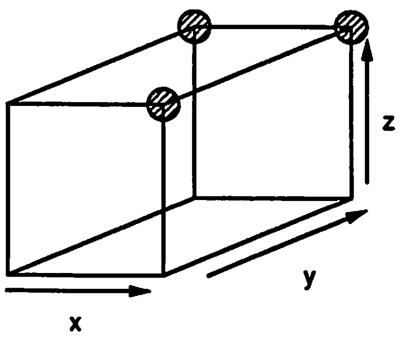
in circuit speed. However, as will be demonstrated, neither redundancy information nor information on variable order affect the major results of this paper.

Eichelberger [25] attempted to characterize hazard-free circuits, and detect hazards. He concluded that *function hazards* were unremovable, whereas *M-hazards* were removable by adding redundancy to a two-level realization of a circuit. Function hazards are those inherent in the cube representation of the function, and occur in every non-trivial Boolean function. *M-hazards* are those that occur due to differences in arrival times of wires attached to the same net (so, for example, a variable y may show a 1 on one lead of the net and a 0 on another lead).

Breuer and Harrison [17] attempted to design tests that would not excite hazards of a circuit. They designed a multivalue calculus to detect necessary and sufficient conditions for tests to not excite hazards in a circuit. Their results for hazard-free circuits required that all the gates in the network be unate, and that the initial input vector on each gate be $\vec{0}$ or $\vec{1}$.

5.3 The Boolean $n-$Space

In general, an n input boolean function can be described as a set of points in n-space. Since each coordinate of each point in the space is either 0 or 1, it is natural



$f = x\bar{y}z + \bar{x}yz + xyz$

Figure 5.2: A Function on the Boolean 3 Cube

to view each point as a vertex of an n-dimensional cube. Each vertex of the cube represents a unique combination of input values, and hence there is a value of the function at that point. In the diagrams in this paper, the vertices where the function is 1 are shaded. The initial values of the inputs form one vertex of the cube; the final values of the inputs are at the furthest vertex from the initial vertex[1]. The initial value of the function is its value at the initial vertex; As the inputs change (assuming each input undergoes exactly one change), we move to a corresponding vertex on the cube, terminating finally at the final vertex. Since the inputs change one at a time, we move a distance one along the cube for each change; since each input changes exactly once, we make precisely n moves. (If an input does not change, we may consider that there are only $n - 1$ variables for the purposes of this discussion).

As we make each move, the value of the function changes to the value of the visited vertex. A *hazard* exists iff the value of the function changes more than once. We formalize these intuitive notions as follows.

Definition 5.3.1 *A sequence (or "walk") of vertices $v_0, ..., v_j$ is said to be* valid *iff*

1. $dist(v_i, v_{i+1}) = 1 \; \forall i$; *and*
2. $dist(v_0, v_j) = j$.

[1]We can assume that if any variable does not change, then it is not a dimension of the n-cube; hence we can assume that all variables change value

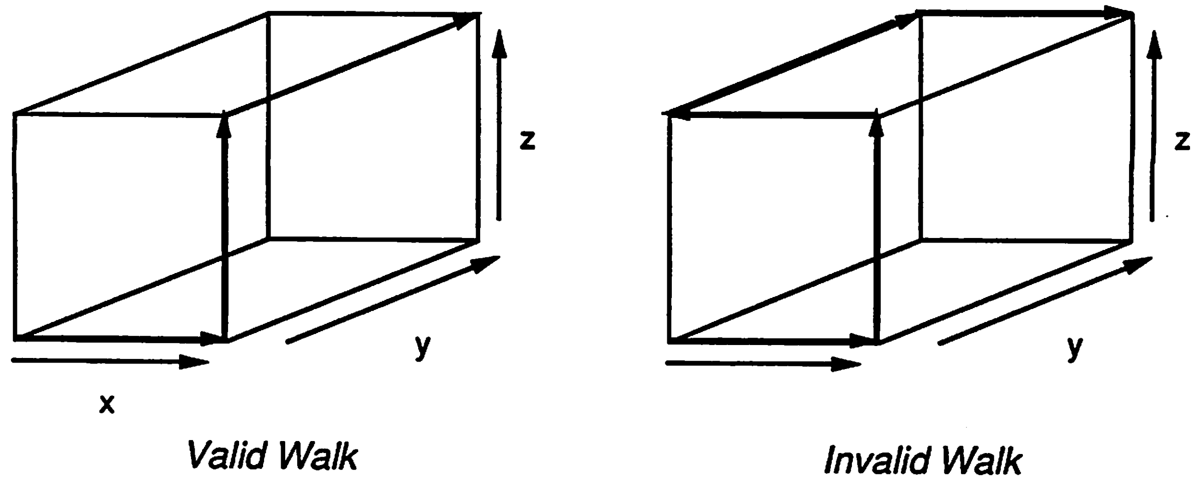


Figure 5.3: Valid and Invalid Walks on the N-Cube

If $j = n$, the sequence is said to be a full walk, and v_n is said to be v_0's **off-vertex,** *denoted $\overline{v_0}$*

Given the assumptions under which we are working, we can make some observations about the number of different walks on the $n-$ cube. At distance exactly k from v_0 there are $2^{\min(k,n-k)}$ unique vertices. Now, on a walk of length k, k variables change, and since they can change in any order they give rise to $k!$ distinct sequences. Hence there are precisely $2^{\min(k,n-k)}k!$ sequences of length k, and so

$$\sum_{k=0}^{n} 2^{\min(k,n-k)}k!$$

valid sequences beginning at some vertex v_0. We are particularly interested in the set of length n sequences. There are $2^n n!$ such sequences, and each such sequence terminates at the *unique* vertex $\bar{v_0}$.

We consider valid walks and their properties. Assumptions 1-5 guarantee that every evaluation of a function corresponds to some valid sequence. The value of the function will change at least twice during the walk iff the walk corresponds to a hazard. We can then characterize walks in terms of the number of times that a function changes value.

Definition 5.3.2 *A valid sequence of vertices $\{v_0, ..., v_n\}$ is said to be* **monotone** *iff, for every $0 \leq i \leq n$, $f(v_i) = f(v_0)$ implies $f(v_j) = f(v_0)\ \forall j \leq i$.*

Non-monotone walks and hazards are equivalent. We can immediately say, when assumptions 1-5 hold:

Theorem 5.3.1 *Let v be any vertex on the n-cube, f any function with its inputs hazard free. f undergoes a hazard during the transition from initial input state v to final input state $\overline{v}$ iff there is at least one non-monotone walk from v to $\overline{v}$*

Proof: Let f undergo a hazard. Now, the transition from v to $\overline{v}$ for the inputs involves some walk on the n-cube, $\{v, v_1, ..., v_{n-1}, \overline{v}\}$, and f assumes the value $f(v_i)$ as the walk transits through v_i. Since f undergoes a hazard, there is some least j such that $f(v_{j-1}) = f(v)$, $f(v_j) \neq f(v_{j-1})$ (the first transition of f), and there is some least $k > j$, such that $f(v_j) = f(v_{j+1}) = ... = f(v_{k-1}) \neq f(v_k)$ (the second transition, inducing the hazard). Such a walk is non-monotone. Conversely, suppose there exists a non-monotone walk. To each valid walk there corresponds an order on the arrival time of the variables, and we have assumed that any variable order may occur, so choose the variable order that corresponds to the non-monotone walk. This order induces the non-monotone walk, and so induces a hazard. ■

Given this, the set of functions which contain only monotone walks is of interest.

Definition 5.3.3 *A logic function is said to be* **statically hazard-free** *iff each valid sequence on the cube is monotone.*

An interesting question that arises is the determination of necessary and sufficient conditions for a function to be statically hazard-free. Intuitively, we expect that a necessary condition is that the ways in which it can change value are highly restricted. We examine the conditions for which walks are monotone.

Lemma 5.3.1 *Let u be any arbitrary point on the n-cube. Then for every point v there is a valid sequence, $\{v, ..., u, ..., \overline{v}\}$.*

Proof: Induction on n, the dimensionality of the cube. For $n = 0$, trivial. Now suppose true for $n < N$, and consider the problem on the N-dimensional cube. If

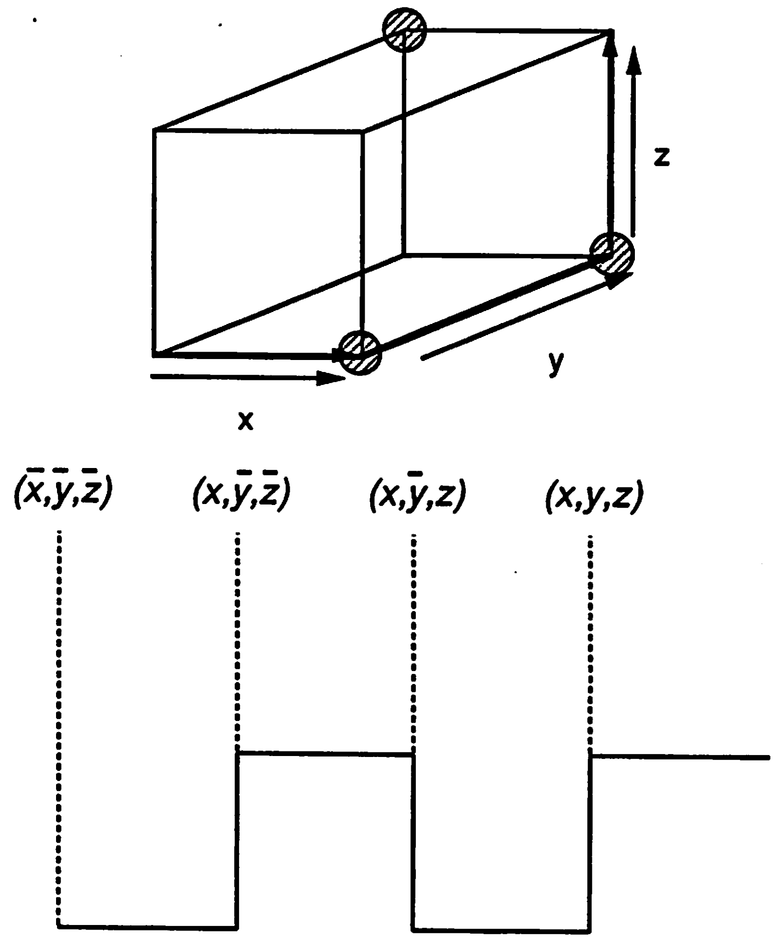


Figure 5.4: Hazards Arising from a Walk

$u \neq v$, $u \neq \overline{v}$ [2], then there is an $N-1$ dimensional face of the $N-$ cube, distance 1 from v, containing both u and $\overline{v}$. Let w be the unique point on this face distance 1 from v. Now, every valid sequence $\{w, ..., \overline{v}\}$ is a suffix of a valid sequence $\{v, w, ..., \overline{v}\}$, and is confined to the $N-1$ dimension face containing w, u and $\overline{v}$. Now, by induction there is at least one valid sequence $\{w, .., u, ..., \overline{v}\}$, and hence there is a valid sequence $\{v, w, .., u, ..., \overline{v}\}$. ■

This lemma leads immediately to strong characterization of the statically hazard-free functions.

Theorem 5.3.2 *Let f be a non-trivial statically hazard-free function on the n-cube*

[2]The problem is trivial if $u = v$ or $u = \overline{v}$

c. Then for each vertex v_i of c, $f(v_i) \neq f(\overline{v_i})$.

Proof: Suppose $f(v_i) = f(\overline{v_i})$ for some v_i. Since f is non-trivial there exists some vertex u such that $f(u) \neq f(v_i)$. By lemma 5.3.1, there is some valid sequence $\{v_i, ..., u, ..., \overline{v_i}\}$, and since $f(v_i) = f(\overline{v_i})$, this is non-monotone. ■

Corollary 5.3.3 *On the n-cube, $f = 0$ on precisely half of the 2^n vertices for all statically hazard-free f.*

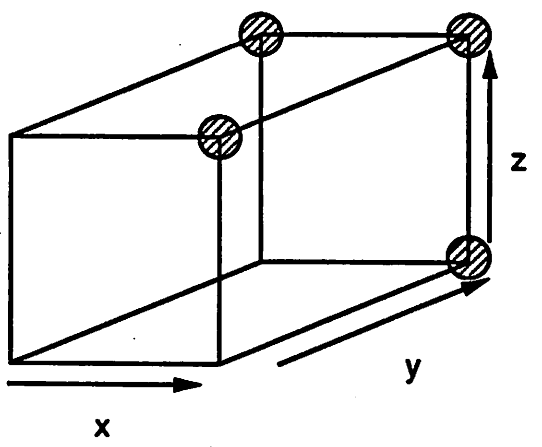
Proof: Follows immediately, for there is exactly one $\overline{v}$ for each vertex v. ■

Corollary 5.3.4 *If f is a statically hazard-free function of $n > 1$ variables, then each face of the n-cube of dimension $n - 1$ contains at least one vertex where $f = 0$, and at least one where $f = 1$.*

Proof: Each face of dimension $n - 1$ contains half the points on the n-cube. If one such face consists entirely of zeroes (ones), the opposing face must consist entirely of ones (zeroes). The face containing all ones corresponds to a literal x, and hence the function f is isomorphic to the one-variable function x. ■

The set of functions satisfying the above theorem and corollaries is very small. Indeed, on the three-cube there are precisely two, one the complement of the other. One of these is shown in figure 5.5. Note that the walk $\{x\overline{y}\overline{z}, x\overline{y}z, \overline{x}\overline{y}z\}$ is non-monotone. Hence we conclude that no (non-trivial) function on the 3-cube is statically hazard-free, for this is the only possible statically hazard-free function on the 3-cube. In fact, this statement applied to the $n-$cube is true for every $n > 1$. We show this now.

Consider again the function in figure 5.5. Note, first, that every path beginning at $\vec{0}$ is monotone, and, second, that the function is *nondecreasing* in each variable; there is no place on the cube in which changing a variable from 0 to 1 changes the value of the function from 1 to 0. We now prove that every hazard-free function must be either nonincreasing or nondecreasing in each variable. The proof follows, but we state the intuition clearly here. If one considers the function as imposing a topography on the cube, then if we begin a walk at some vertex v, then v



$f = xz + yz + xy$

Figure 5.5: Only Function on the 3-cube Satisfying Corollaries 5.3.3-5.3.4

must be in a single basin of this topography (if $f(v) = 0$), or on a single plateau (if $f(v) = 1$).

Definition 5.3.4 *A function f is* **nondecreasing (nonincreasing)** *in a variable x_j iff changing x_j from 0 to 1 (1 to 0) does not change f from 1 to 0 (0 to 1). If f is either nonincreasing or nondecreasing in x_j, then f is said to be* **unate** *in x_j. Otherwise f is said to be* **binate** *in x_j. If f is nondecreasing (nonincreasing) in every variable, then f is said to be nondecreasing (nonincreasing). If f is either nondecreasing or nonincreasing in every variable, then f is said to be unate. Otherwise f is said to be binate.*

Theorem 5.3.5 *Let f be any function, and $v = (x_1, ..., x_n)$ be any vertex of the n-cube such that every valid sequence beginning at v is monotone. Let $f(v) = 0$. Then if $x_j = 0$ at v, f is nondecreasing in x_j, and if $x_j = 1$ at v, then f is nonincreasing in x_j. Similarly, if $f(v) = 1$, then if $x_j = 0$ at v, f is nonincreasing in x_j, and if $x_j = 1$ at v, then f is nondecreasing in x_j.*

Proof: We prove for the case $f(v) = 0$; the case $f(v) = 1$ follows by symmetry. Let $x_j = 0$ at v. If f is not nondecreasing in x_j, then there is some vertex $u = (.., x_j = 0, ...)$ such that $f(u) = 1$ and its neighbour vertex $u' = (.., x_j = 1, ...)$ such that $f(u') = 0$. There is a valid walk $\{v, ..., u, u', ..., \overline{v}\}$, and, since $f(v) = f(u') = 0, f(u) = 1$, $\{v, ..., u, u', ..., \overline{v}\}$ is non-monotone. Similarly, if $x_j = 1$ at v, then if f is not nonincreasing in x_j, then there is some vertex $u = (.., x_j = 1, ...)$ such that $f(u) = 1$ and its neighbour vertex $u' = (.., x_j = 0, ...)$ such that $f(u') = 0$. There is a valid walk $\{v, ..., u, u', ..., \overline{v}\}$, and, since $f(v) = f(u') = 0, f(u) = 1$, $\{v, ..., u, u', ..., \overline{v}\}$ is non-monotone. ■

Corollary 5.3.6 *If f is statically hazard-free, then f is nondecreasing or f nonincreasing in every variable.*

Proof: Immediate, since if $f(\vec{0}) = 0$, then by the theorem f is nondecreasing in every variable, or $f(\vec{0}) = 1$, in which case f is nonincreasing in every variable. ■

It is almost immediate now that there are no non-trivial statically hazard-free functions of greater than one variable. Consider vertices v and $\vec{0}$, $v \neq \vec{0}$, $f(v) = f(\vec{0})$, where every walk from either v or $\vec{0}$ is monotone. Note that f must remain nondecreasing (nonincreasing) on the cube obtained by rotating v into $\vec{0}$, which in turn implies that f is independent of at least some dimensions of the $n-$cube. We formalize this argument below, showing that v must be indistinguishable from $\vec{0}$.

We begin by demonstrating a sufficient condition for f to be independent of x.

Lemma 5.3.2 *If f is both nonincreasing and nondecreasing in some variable x, then f is independent of x*

Proof: Changing the value of x from 0 to 1 cannot change the value of the function either from 1 to 0 or from 0 to 1. Hence changing the value of x cannot change the value of the function, and done. ■

Lemma 5.3.3 *Let v be any vertex, $f(v) = f(\vec{0})$, s.t. every valid sequence $\{\vec{0}, ..., \vec{1}\}$ and every valid sequence $\{v, ..., \overline{v}\}$ is monotone. Then f is independent of every variable x_j in which v differs from $\vec{0}$.*

Proof: WLOG, $f(\vec{0}) = f(v) = 0$. By the previous theorem, since $x_j = 0$ at $\vec{0}$ f must be nondecreasing in x_j. However, since $x_j = 1$ at v, f must be nonincreasing in x_j. Hence f is independent of x_j. ■

Lemma 5.3.4 *Let v be any vertex, $f(v) = f(\vec{1})$, s.t. every valid sequence $\{\vec{1}, ..., \vec{0}\}$ and every valid sequence $\{v, ..., \overline{v}\}$ is monotone. Then f is independent of every variable x_j in which v differs from $\vec{1}$.*

Proof: Follows exactly the proof of lemma 5.3.3 ■

Theorem 5.3.7 *Let f be a non-trivial function of $n > 1$ variables. Then f is not statically hazard-free.*

Proof: Induction on n. If $n = 2$, follows by case analysis on the functions $xy, x \oplus y, x + y$ (the other 7 true two-variable functions are isomorphic to one of these three for this purpose). Suppose theorem holds for $n < N$. If $n = N$, consider the set of N-variable functions which are not isomorphic to $N - 1$ variable functions. If f is a statically-hazard-free function, then for every vertex v every valid sequence $\{v, ..., \overline{v}\}$ is monotone. In particular, every sequence $\{(1, 0, ..., 0), ..., (0, 1, .., 1)\}$ is monotone. By lemma 5.3.3, therefore, f is independent of x_0, contradicting the assumption that f was not isomorphic to a function of $N - 1$ variables. ■

Theorem 5.3.8 *Let f be a non-trivial function of all of its variables. Then there is at most one vertex v such that every walk commencing at v is monotone and $f(v) = 0$, and at most one vertex v' such that every walk commencing at v' is monotone, and $f(v') = 1$. Further, if such a vertex v exists, then v' exists and $v' = \overline{v}$.*

Proof: That at most one such v and one such v' exist is immediate from lemma 5.3.3, so all that must be shown is that if v exists, v' exists and $v' = \overline{v}$. For this, suppose v exists. By appropriate change of variables we can ensure that $v = \vec{0}$, and hence that the function is nondecreasing in all its variables. But immediately, then, we have that $f(\vec{1}) = 1$, and that every walk from $f(\vec{1})$ is monotone, and of course $\vec{1} = \overline{\vec{0}}$, and so done. ■

The preceding theorems show that if a function is not to have a hazard under evaluation, then the function must be unate, and, further, (after some rotation of the cube) the evaluation must begin at the state $\vec{0}$ or $\vec{1}$. The practice of setting each variable to a known state before evaluation is known as *precharging*.

Thus precharging and unateness are necessary for hazard avoidance. The question is, are they sufficient? The following theorem provides this final piece to the puzzle.

Theorem 5.3.9 *If f is nondecreasing, then every sequence $\{\vec{0}, ..., \vec{1}\}$ is monotone.*

Proof: f is nondecreasing. Let $P = \{\vec{0}, v_1, ..., v_i, ..., \vec{1}\}$ be any valid sequence. We must show it is monotone.

Let v_i be the first node such that $f(v_i) = 1$. Then we must show $f(v_j) = 1 \, \forall j > i$. We proceed by induction. For v_{i+1}, observe that the only difference between v_i and v_{i+1} is that some variable, say x_k, was changed from 0 to 1. This cannot change f to 0, since f is nondecreasing. Similarly, if $f(v_j) = 1 \, \forall i < j < N$, the only difference between v_N and v_{N-1} is that variable x_r for some r was changed from 0 to 1 and hence, $f(v_N) = f(v_{N-1}) = 1$, and done. ∎

In sum, in this section we have shown that the family of hazard-free circuits, under our initial assumptions, is identical to the family of precharged unate circuits. In the next two sections, we will explore two of our base assumptions, and show that relaxing either or both assumptions has no effect on the results, and in the second case that the assumption cannot be easily relaxed in a VLSI environment.

5.4 The Satisfiability Don't-Care Set and Restricted Cubes

The preceding arguments rested on assumption (5): any starting or ending vertex was permitted. In practice, this is not the case. If the input variables to a node are themselves functions of the primary input variables, then various combinations of the input variables (typically called *faces* or *cubes*) may not be possible *static* values of the input vertices, and if so, are not appropriate terminal vertices of a walk. The collection of such impossible cubes is generally known as the *Satisfiability Don't-Care (SDC) Set*[3]. Now, even for non-precharged logic one can view the set of *allowable* starting and terminal vertices to reside in $\overline{SDC}$. However, it would be an error to consider that all walks across a cube must avoid the SDC set, for the vertices within these cubes are only forbidden as *static* values; there is no reason to believe that the vertices within these cubes may not occur as transient values as the inputs change.

Nevertheless, certain cubes may be forbidden. Given any start vertex v(any vertex from which all walks are monotone), any cube of the SDC set containing the terminal vertex $\overline{v}$ will not be entered on any valid walk from v, since such a walk will not exit the cube of the SDC set and hence will not terminate at a statically valid vertex. We call these cubes the *restricted cubes* of vertex v, denoted $R(v)$.

Definition 5.4.1 *A cube c is a* **restricted vertex** *of v iff $c \subseteq SDC$ and $\overline{v} \in c$. The set of all restricted vertices of v is denoted $R(v)$.*

The essential point about the vertices contained within the Satisfiability Don't Care set is that the value of the function may be chosen arbitrarily on these points, since these values will never be realized statically. The implemented function, of course, is completely specified; there is a real, concrete value attached to each point on the n–cube. The concrete value is a matter of concern for those vertices inside the SDC set but outside $R(v)$, since these will be visited in transit. Within $R(v)$, however, we can choose values arbitrarily, since these vertices will never be visited. The strategy we use to prove results in this section is to demonstrate that a single "good" assignment of function values exists for vertices in $R(v)$, and that by making that assignment we do not impose any restrictions on the hazard freedom of the realized function. We then show that the results of the preceding sections hold for such functions; this in turn shows that redundancy information in the form of the SDC set does not yield any significant loosening of the precharged, unate requirements of the previous sections. The "good" assignment simply forces values in the unreachable parts of the cube ($R(v)$) to be the value opposite that of the function at v.

Theorem 5.4.1 *Let $v \notin SDC$ and every walk from v terminating outside $R(v)$ be monotone. If $f(u) = \overline{f(v)}$ for all $u \in R(v)$, then every walk $\{v, ..., \overline{v}\}$ is monotone.*

Proof: Note that every walk $\{v, ..., w, w', ..., \overline{v}\}$ is composed of a prefix $\{v, ..., w\}$ lying outside $R(v)$ and a suffix $\{w', ..., \overline{v}\}$ lying inside $R(v)$. Since $f(w') = ... = f(\overline{v}) = \overline{f(v)}$,

f changes value more than once over the walk iff it changes at least twice on the prefix $\{v, ..., w\}$. But $\{v, ..., w\}$ is monotone, and done. ■

Definition 5.4.2 *The function $C_{f,v}$ obtained by setting $C_{f,v}(u) = f(u)$ for all u outside $R(v)$, and $C_{f,v}(u) = \overline{f(v)}$ for all $u \in R(v)$ is called the* **Completion** *of f with respect to v.*

We show that $C_{f,v}$ is monotone under evaluation beginning at v iff f is. Following this, we use the results of the previous section to draw conclusions about f.

Theorem 5.4.2 *Every walk $\{v, ..., \bar{v}\}$ is monotone under $C_{f,v}$ iff every walk $\{v, ..., w\}$ is monotone under f for each w outside $R(v)$.*

Proof: If $\{v, ..., \bar{v}\}$ is monotone under $C_{f,v}$, then some w is the last vertex on the walk outside $R(v)$. $\{v, ..., w\}$ is monotone under $C_{f,v}$, and for each u in the walk $\{v, ..., w\}$, u is outside $R(v)$ and so $C_{f,v}(u) = f(u)$, and so $\{v, ..., w\}$ is monotone under f. For the converse, note that every walk from v terminating in w under f (and so under $C_{f,v}$ is monotone, and for each $u \in R(v)$ we have $C_{f,v}(u) = \bar{C}_{f,v}(v)$. Hence $C_{f,v}$ meets the premises of f for the preceding theorem, and hence every walk is monotone under $C_{f,v}$. ■

Corollary 5.4.3 *Every walk $\{v, ..., w\}$ is monotone under f for each w outside $R(v)$ iff $C_{f,v}$ is unate in every variable.*

Proof: Immediate from theorem 5.3.5 and the fact that $C_{f,v}$ is a completely specified function. ■

The matter of precharging remains. This follows immediately from lemma 5.3.3.

Theorem 5.4.4 *Assume that $\vec{0}$ outside SDC. Let v be any vertex outside SDC, $f(v) = f(\vec{0})$, s.t. every valid sequence $\{\vec{0}, ..., \vec{1}\}$ and every valid sequence $\{v, ..., \bar{v}\}$ is monotone in $C_{f,v}$. Then f is independent of every variable x_j in which v differs from $\vec{0}$.*

Proof: Given two start vertices v and u, with $f(u) = f(v)$, $C_{f,u}$ and $C_{f,v}$ have the unateness properties specified by theorem 5.3.5. These properties may vary, since $C_{f,u}$ and $C_{f,v}$ need not be identical functions over the set $R(v) \cup R(u)$. However, let u and v differ in some x_j. Then $C_{f,u}$ is nondecreasing in x_j, and hence f is nondecreasing in x_j over the care set; similarly, $C_{f,v}$ is nonincreasing in x_j, and so f is nonincreasing in x_j over the care set. Hence $C_{f,u}$ and $C_{f,v}$ differ in the unateness of x_j, and f is independent of x_j over the care set and so is independent of x_j. ■

Thus the results of this section show that the (unrealistic) assumption that no input states were forbidden had no effect on the major result of the section 5.3.

5.5 Ordering The Inputs

If assumption (5) (that all input vertices are possible) does not affect the hazard-free status of the function, we should examine the assumption that the order in which the inputs change is arbitrary. Clearly the results of the previous two sections are heavily dependent upon this assumption. The objectives of this section are

1. to demonstrate that this assumption (4) is reasonable under most circumstances;, and

2. to demonstrate that assumption (4) has no effect on the major results of this paper.

In an abstract timing environment, each function is treated as a node in a directed acyclic graph. The edges represent the connections between functions; each edge is given a positive weight, which represents the delay between the time that the edge source switches value and the time that the value reaches the edge sink.

The value at either the source or the sink at any time is a function both of the initial state of the edge and of the current partial evaluation of the function at its source. We have been representing this partial evaluation as a walk along the function at the source of the edge, and ignoring the mechanism by which the variables to that function changed value. Of course, these input variables are represented by other edges in the graph, and the changes in their value reflect the changes in state of those edges. These edges in general change value in response to changes in the function at the edge source. Tracing back these changes in function, or *events*, we arrive naturally at the concept of events travelling down paths from the primary inputs. Since each edge has a weight, the delay down any path is simply the sum of the edge weights. If the edge weights model delay precisely, then one can time the arrival of an event at some node. Further, one can adjust the arrival time of an event at a node by adding nodes (called *static delay buffers*) to the graph at appropriate points along the path. The purpose of these nodes is merely to delay the arrival of signals at a point.

In practice, delays, even those generated by static delay buffers, are not known so precisely. Recall that delays of the various edges are dependent upon the precise sizes of the capacitors and resistors that make up the physical circuit, which in turn are heavily dependent upon the precise physical layout of the circuit in silicon, upon variations in the various doping steps of the fabrication process, and finally upon thermal and electrical factors in the operating environment. Through careful measurement of the mask-level design, one can obtain the needed information concerning the physical layout, but parameters of the fabrication line and operating environment can only be given probabilistically. As a result the edge weights are typically given as a *range* of figures, and the maximum of the range is taken as the edge weight. Further, the edge weights cannot be taken to scale uniformly throughout the circuit; it is entirely possible that some instance of the circuit will have some edges operating at or near their minimum delay values and others operating at or near their maximum delay values.

Hence when one considers an abstract circuit, one is in fact considering a *family* of circuits. The distinguishing characteristic of the circuit under analysis is that it is the *slowest* in the family. As a result, a partial order on the inputs of a gate is invalid unless it can be shown to hold for *every* member of the family. We examine the conditions under which this may be shown.

Definition 5.5.1 *The circuit obtained by assigning the maximum value to every edge is called the* **slow** *circuit, and the circuit obtained by assigning the minimum value to* *every edge is called the* **fast** *circuit.*

A node has settled to its final value only when its last event has propagated through the node. In chapter 2 we showed that this can in general be bounded above by the length of the longest viable path terminating at that node. The upper bound on a node settling is therefore the length of the longest viable path in the network in the slow circuit. A lower bound on a node settling is the length of the shortest path in the fast network. Since the condition on imposing the order x switches later than y is that the upper bound on y is less than the lower bound on x, we have:

Theorem 5.5.1 *An input x to a function f can be said to switch earlier than y if the longest viable path terminating in x in the slow network is shorter than the shortest path terminating in y in the fast network.*

If the delay of each edge in the fast network is taken to be 0, we have:

Theorem 5.5.2 *If the minimum edge weight for every edge is 0, then no partial order exists on the inputs to any function*

For the remainder of this section, we presume that the minimum weight of the range at each edge is 0. Under these circumstances, one can prove the following theorem.

Theorem 5.5.3 *If two events e_1 and e_2, arrive at node f_m, it cannot be determined which arrived first.*

Proof: If e_1 and e_2 travel down disjoint paths to f_m, either can be made to arrive before the other by reducing the delays along the relevant path to 0. If the two paths conjoin, then the event which arrives first at the junction arrives first at f_m, but the paths have disjoint prefixes and hence by the first case the event which arrives first at the junction cannot be predicted. If the two paths diverge, and then reconverge at f_m, then the shorter of the two disjoint reconvergent segments carries the first event, but either of the reconvergent segments can be made shorter by reduction of edge delay along the chosen piece to 0. These are all the cases, and so done. ∎

Even though one can order the inputs on a node if one has nonzero minimum delays for each edge, it is not clear that this loosens the restrictions on hazards. The fact that these restrictions still apply is a consequence of the two final theorems.

Now, clearly if a total order doesn't help us in loosening the precharged/unate restrictions, then no partial order will. Let us consider the case where x_i switches before x_j for $j > i$. We show in this case that every hazard-free function must be both unate and have the starting vertex fixed.

Assume a total variable order and consider the case of the three dimensional cube. Any attempt to construct a hazard-free function on this 3-dimensional cube with a 0 at both $\vec{0}$ and $\vec{1}$ fails, even though only one valid walk from $\vec{0}$ need be considered, due to the variable order. However, the fact of the variable order puts no restrictions on either the start or terminal vertices of a walk, merely on the internal structure of the walk once initial and terminal vertices have been selected. Since every walk of distance n involves $n+1$ points, a distance n walk originating and terminating on vertices where the function is 0 gives $n+1$ points where the function is 0. From the $n+1$ given 0 points on the cube derived from the walk, and from the fact that one could construct a single walk from any point to any point, we were able to fill the 3-dimensional cube with zeroes at each vertex. It developed that this is true of every cube, a fact we prove now.

Theorem 5.5.4 *Let f be a hazard-free function on the N-cube, with a total order on the arrival of the input variables. Let $f \doteq 0$ on any two points distance N apart on this cube. Then $f = 0$ on the entire cube.*

Proof: Without loss of generality, we assume that the variables change in the order $x_1, ..., x_n$; i.e., x_i switches before x_j iff $i < j$. We show by induction on n. The case is trivial for either the 0 or 1 dimensional cube, so suppose this is true for all cubes of dimension $< N$. For the case $n = N$, without loss of generality assume $f(\vec{0}(= \overline{x_1}, ..., \overline{x_N})) = f(\vec{1}(= x_1, ..., x_N)) = 0$. Now, the walk from $\vec{0}$ to $\vec{1}$ therefore induces a 0 on every point along the walk, and these points include the point $u = (1, ..., 1, 0)$. $\vec{0}$ and u form extrema points of the $N-1$ dimensional cube $\overline{x_N}$, and so by induction $f = 0$ everywhere on this cube. Hence $f = 0$ at the point $u' = (1, 0, ..., 0)$. Now,

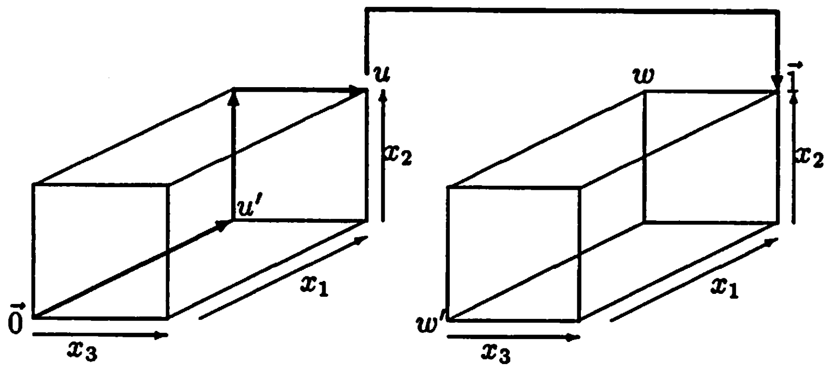


Figure 5.6: Illustration of theorem proof on 4-dimensional cube

u' and $\vec{1}$ form extrema points of the $N-1$ dimensional cube x_1, and hence $f = 0$ everywhere on this cube. Hence $f = 0$ at the point $w = (1, ..., 1, 0, 1)$, and so $\vec{0}$ and w form extrema points of the cube $\overline{x_{N-1}}$, and so $f = 0$ everywhere on this cube, and in particular at $w' = (0, ..., 0, 1)$. Finally, w' and $\vec{1}$ form extrema points of the cube x_N, and so $f = 0$ everywhere on the cube x_N. Now, $f = 0$ everywhere on $\overline{x_N}$ as well, so $f = 0$ everywhere on the entire cube, and done. ■

An illustration of the proof on the four-dimensional cube is given in figure 5.6.

With this in hand, we can immediately prove as a corollary the analogue to theorem 5.3.5

Theorem 5.5.5 *Let f be any function and the total order in which the inputs switch is given. $v = (x_1, ..., x_n)$ is any vertex of the n-cube such that every valid sequence beginning in v is monotone. Let $f(v) = 0$. Then if $x_j = 0$ at v, f is nondecreasing in x_j, and if $x_j = 1$ at v, then f is nonincreasing in x_j. Similarly, if $f(v) = 1$, then if $x_j = 0$ at v, f is nonincreasing in x_j, and if $x_j = 1$ at v, then f is nondecreasing in x_j.*

Proof: We prove for the case $f(v) = 0$, $x_j = 0$ at v only. If f is *not* nondecreasing in x_j, then there is some point u with $f(u) = 1$, $x_j = 0$ at u, with a neighbour u', $x_j = 1$ at u', $f(u') = 0$. But now v, u' are extrema of a cube, and $f = 0$ at both extrema. Hence $f = 0$ on the entire cube. But $f(u) = 1$, and u is a point of this cube, contradiction. ■

Lemma 5.3.3 follows this theorem with exactly the proof with which it follows theorem 5.3.5, which suffices to prove the point of this section: imposing any order on the variables does not relax the requirement that the functions be precharged and unate in order to be hazard-free.

The final question that naturally arises is whether a denial of both assumptions together is sufficient to loosen any of the restrictions imposed in the general case, given that neither separately is sufficient. We will not repeat the arguments of the previous section here. However, if we assume both a total order on the inputs *and* the existence of forbidden states, we can use the results of this section to argue that the function must be unate exclusive of the don't care set, and then use the completion construction of the previous section to argue that precharging is required.

Chapter 6

Timing Analysis in Hazard-Free Networks

6.1 Introduction

Chapters 2 and 3 centered around algorithms which found the longest path down which an event could propagate in a combinational network. Such a path is called the *true critical path*, and is of great interest in timing verification.

Due to the uncertainties in the actual delays of the various circuit elements mentioned in the above chapters, we argued that true critical path procedures must not only report the longest true path, but must also report a path whose length is no less than the length of the true critical path in all the *faster* networks in the family (since such a faster network will be the network actually fabricated and in use). This robustness constraint was called the *monotone speedup property*: one could not lengthen the critical path of the network by speeding up some of its components.

Recall from chapter 2 that in general networks simply tracing the set of paths which actually propagate events is not a robust procedure, in the sense of monotone speedup, since one can construct networks in which reducing the delays of some circuit elements causes signals to propagate down longer paths than the longest true path in the original network. Indeed, one can construct networks with *no* full true paths in the original network but have arbitrarily long true paths in some faster network. This is a particularly unfortunate state of affairs, since it ensures that any correct, robust criterion for true path tracing will be *loose* in the sense that it is possible that some paths down which events cannot propagate will be reported as true. Hence one is guaranteed to overestimate the critical delay in a network. Indeed, one can construct circuits in which the difference between the critical delay that one obtains by tracing the true paths and the critical delay that one *must* report is arbitrarily large.

It would be pleasant if there were an identifiable class of circuits for which the true longest path delay criterion were robust, and as the reader has certainly guessed by now, hazard-free circuits are such a class. It is the objective of this chapter to prove this, and to derive an algorithm which returns the length of the true critical path.

6.2 Dynamic Sensitization is Robust on Precharge-Unate Networks

We begin this discussion by observing that the results of the previous sections are assumed, and hence when we speak of hazard-free networks we also mean precharged, unate circuits, since we now know these concepts are equivalent. Hence at every time $t \geq 0$, each node f is either in its initial precharged state, or its evaluation state; the two, of course, might be the same. If one considers a function evaluation as a walk along the cube, as above, then a change in value of the function occurs in response to a change in value of one of its inputs; we call such a change in value an *event*, and say that the event has propagated from the input to the output of the function. It is natural to extend this notion of propagation from input to output to propagation along a whole sequence of nodes, $\{f_0, ..., f_m\}$, where each f_i is an input to f_{i+1}.

Recall from chapter 2 that for a node f_i to respond to an event on f_{i+1}, the other inputs (also called the *side inputs*) to f_{i+1} must be at the appropriate values when the event propagates through f_i in order to propagate through f_{i+1}; for example, if f_{i+1} is an OR or NOR gate, the side inputs must all hold the value 0 in order for the value of f_i to determine the value of f_{i+1}, and if f_{i+1} is an AND or NAND gate, the side inputs must all hold the value 1. In general, one can show that these values form a satisfying set to a logic function, the *boolean difference* of f_{i+1} with respect to f_i, and is denoted $\frac{\partial f_{i+1}}{\partial f_i}$.

In chapter 2, it was presumed that a set of inputs (called an *input vector* or an *input cube* c_1 was applied to the primary inputs at $t = -\infty$ and determined the initial state of the wires in the network. A vector c_2 was then applied at $t = 0$ and permitted to propagate through the network. Under these circumstances, it was shown that:

Theorem 6.2.1 *A path $\{f_0, ..., f_m\}$ is sensitizable by primary input cubes c_1, c_2 (an event propagates under c_1, c_2) iff $\frac{\partial f_i}{\partial f_{i-1}}(c_1, c_2, \tau_{i-1}) = 1$*

Now, the cube c_1 was present in that formulation since the cube c_1 determined the state of the network at $t = 0$. Now, however, the initial state of the network is determined by precharging, and so we may dispense with the vector c_1, rename c_2 as simply c, and write:

Theorem 6.2.2 *A path $\{f_0, ..., f_m\}$ is sensitizable by a primary input cubes c iff $\frac{\partial f_i}{\partial f_{i-1}}(c, \tau_{i-1}) = 1$*

Our task is to show that the dynamic sensitization criterion is robust on hazard-free networks. Such a network may be presumed to consist exclusively of AND, NAND, OR and NOR gates; more complex gates may be macroexpanded into subnetworks composed of these, and the set of dynamically sensitizable paths not decreased by the macroexpansion. Now, note for each such gate the sensitization conditions are either a 1 on all side inputs (for AND and NAND), or a 0 on all side inputs (for OR or NOR). Now, we presume that every input to one such gate is precharged to either 0 or 1; indeed, this is a requirement for the gate to be hazard-free. In general, then, either each input to a gate is precharged to its sensitizing value, or each input is precharged to its desensitizing value. In the former case, the first input to change state propagates, since each other input is still at its sensitizing value, and in the latter case, the last input to change state propagates, since each other input must have already changed to its sensitizing value. Informally, then, if we speed the network up, neither the first nor the last event on an input will be slowed down, and hence the event that propagates will not be slowed down. We formalize these notions below.

Lemma 6.2.1 *Let N be a hazard-free network. Then for each node f, input cube c, the set of dynamically sensitizable paths under c through f are all of the same length.*

Proof: This is immediate. Each dynamically sensitizable path through f propagates an event. But there is at most one event at f, and hence each separate dynamically sensitized path must propagate its event through f at the same time. ■

Definition 6.2.1 *We denote the time that an event propagates under c through f as $t(f,c)$. If we are discussing the time that the event arrives at f in the context of a specific network N, we denote this as $t_N(f,c)$.*

This leads immediately and naturally into the speedup, or robustness, theorem. Suppose we speed network N up into N' by reducing some internal delays. What we'd like to prove is that the longest dynamically sensitizable path in N' is no longer than the longest such path in N; in other words, $t_N(f,c) \geq t_{N'}(f,c)$.

Theorem 6.2.3 *Let N be any network, N' be any network obtained from N by reducing delays at some nodes. Then for each f, $t_{N'}(f,c) \leq t_N(f,c)$*

Note the proof of this theorem gives us the result we want, for the delay reported by a dynamic timing analyzer on the original circuit under this sensitization criterion is the max over all primary outputs F_j and cubes c of the $t(F_j, c)$.

Proof: Induction on the level of f, $\delta(f)$. For f a primary input, trivial. Assume true for all nodes of lesser level than f, and so, in particular, assume the result for the inputs g_i of f. Now, for each input g_i of f, we have $t_{N'}(g_i,c) \leq t_N(g_i,c)$. Now, since we have restricted our attention to AND, NAND, OR and NOR gates we may write:

$$t(f,c) = w(f) + \begin{cases} \min(t(g_i,c)) & g_i\text{'s are precharged to the sensitizing value} \\ \max(t(g_i,c)) & \text{otherwise} \end{cases}$$

without loss of generality, we presume the first case, and since $\min(t_{N'}(g_i, c)) \leq \min(t_N(g_i, c))$ and $w_{N'}(f) \leq w_N(f)$ we have that $t_{N'}(f, c) \leq t_N(f, c)$, as required. The reader can easily verify that the proof for the other case follows. ■

6.3 The Dynamic Sensitization Function

We now have the proof, and all that we desire is an algorithm to compute the longest dynamically sensitizable path in such a network. Fortunately, we can adopt the dynamic programming procedure developed previously to calculate this. First we must derive a logic function λ_P with the property that:

$$\lambda_P(c) = 1 \text{ iff } P \text{ is sensitized by } c$$

In general, we wish to write a series of equations similar to (2.2)-(2.4). The first equation, the analogue to (2.2), we can write immediately:

$$\lambda_P = \prod_{i=0}^{m} \lambda_P^{f_i} \tag{6.1}$$

Recall that $\mathcal{P}_{g,t}$ denoted the set of paths of length $\geq t$ terminating in g. It is also convenient to denote the set of paths of length *exactly* t terminating in g; denote these as $\mathcal{R}_{g,t}$.

The equation for $\lambda_P^{f_i}$ still must be derived.

We first consider DOMINO networks only; this suffices to encompass NORA networks as well, since DOMINO and NORA networks are isomorphic through the translation of appendix D. Note that for DOMINO networks, it suffices to consider only the set of rising events at a node. Note further that since DOMINO networks consist only of noninverting gates, a rising edge through a fanin of a node produces a rising edge at the output of the node (we neglect primary inputs for the moment).

Consider the generic picture of the pulldown network of a precharged-unate function, presented in figure 6.1.

In this diagram, q consists of the pulldown network feeding through f_{i-1} and r consists of the network which does not feed through f_{i-1}. Each path through a subnetwork corresponds to a cube of the subnetwork.

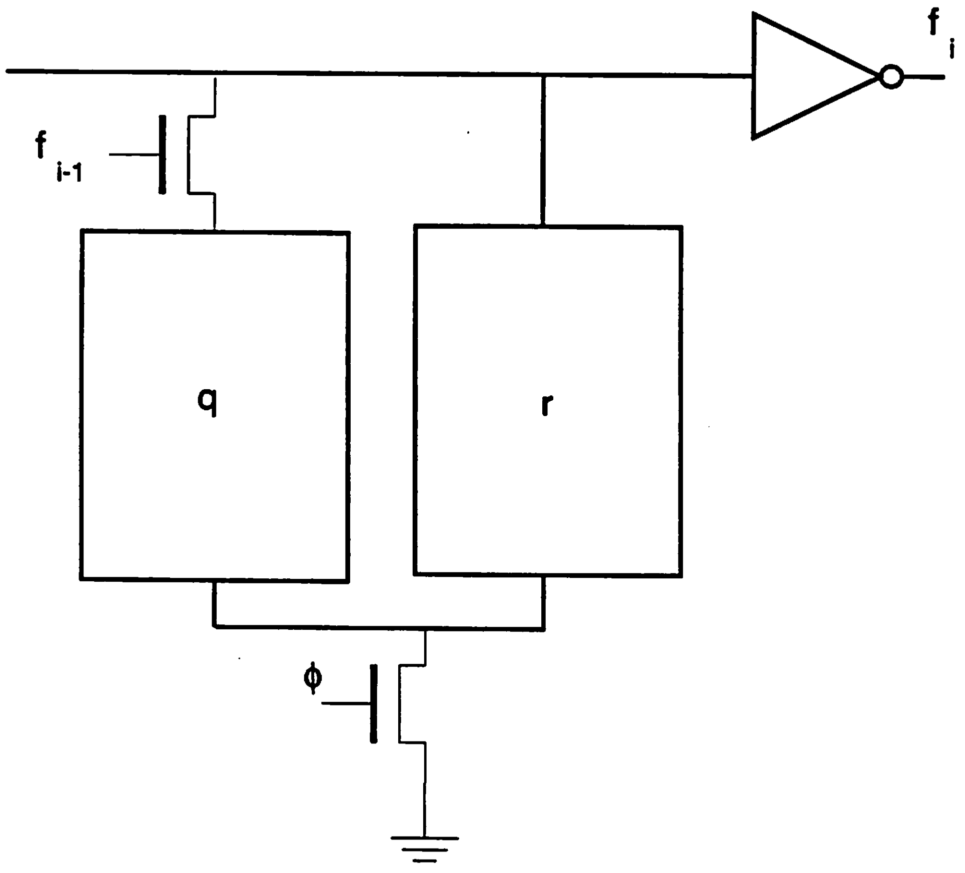


Figure 6.1: Generic Precharged-Unate Gate

The boolean difference $\frac{\partial f_i}{\partial f_{i-1}}$ may be written, in terms of q and r,

$$(q + r) \oplus r$$

which simplifies to:

$$q\overline{r}$$

or, in other words, q is conducting and r is nonconducting. This is also the condition under which a rising edge on f_{i-1} forces a rising edge on f_i.

Denote as $\lambda_1^{f,t}$ the condition that there is a rising edge on f at $T \geq t$. Denote as $\lambda_2^{f,t}$ the condition that there is a rising edge on f at $t \leq T$. Since every sensitizable path produces a rising edge, we have:

$$\lambda_1^{f,t} = \sum_{Q \in \mathcal{P}_{f,t}} \lambda_Q \tag{6.2}$$

We can then write the expression for $\lambda_2^{f,t}$ in terms of $\lambda_1^{f,t}$, by noting that there can be only one event on f for each input combination. The condition on $\lambda_1^{f,t}$ is that this event occurs no earlier than t, and hence $\lambda_1^{f,t}\lambda_2^{f,t}$ is equal to the dynamic sensitization functions of the paths terminating in f of length precisely t:

$$\lambda_2^{f,t}\lambda_1^{f,t} = \sum_{Q \in \mathcal{R}_{g,t}} \lambda_Q$$

further, $\lambda_1^{f,t} + \lambda_2^{f,t} = f$, since f must rise at some time if $f = 1$. Hence we must have:

$$\lambda_2^{f,t} = f(\overline{\lambda_1^{f,t}} + \sum_{Q \in \mathcal{P}_{g,t}} \lambda_Q) \tag{6.3}$$

We are now in a position to determine when a rising edge on f_{i-1} forces a rising edge on f_i. Basically, we must have q conducting at τ_{i-1}, r nonconducting *before* τ_{i-1}. Now, for q to be conducting, we must have that some path through q is conducting. We may write

$$q = c_1 + c_2 + ... + c_n$$

where each c_j is a path through q and a cube in the sum-of-products expression for q. The identity between these two must be exact, and can always be arranged through the appropriate choice of sum-of-products expression for q.

The requirement that q be conducting is therefore that some path c_j be conducting, i.e., must be *identically* 1. Further, since the precharge state of every variable is nonconducting, we must have that every variable g in c_j rise and arrive no later than τ_{i-1}. This requirement is expressed in the condition $g\lambda_2^{g,\tau_{i-1}}$ i.e.

$$\prod_{g \in c_j} g\lambda_2^{g,\tau_{i-1}}$$

Since $c_j = \prod_{g \in c_j} g$, the condition for c_j to be conducting is:

$$c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i - 1}$$

and since the condition for $q = c_1 + ... + c_n$ to be conducting is that some c_j be conducting, we have that the condition for q to be conducting is:

$$\sum_{c_j \in q} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i - 1} \tag{6.4}$$

The equation for r to be nonconducting is somewhat simpler. Every cube $c_k \in r$ must be nonconducting. Hence for some transistor controlled by g in c_k, the final value of g must be nonconducting (0), or g must be late and so be at its precharged, nonconducting value. This condition can be expressed:

$$\sum_{g \in c_k} (\lambda_1^{g,\tau_i - 1} + \overline{g})$$

and the condition for r to be nonconducting is that this be true for every cube:

$$\prod_{c_k \in r} (\sum_{g \in c_k} \lambda_1^{g,\tau_i - 1} + \overline{g}) \tag{6.5}$$

Putting (6.4) and (6.5) together, we get:

$$\lambda_P^{f_i} = \left(\sum_{c_j \in q} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i - 1} \right) \left(\prod_{c_k \in r} (\sum_{g \in c_k} \lambda_1^{g,\tau_i - 1} + \overline{g}) \right) \tag{6.6}$$

In sum, we define:

Definition 6.3.1 *The* dynamic sensitization function *(also* sensitization set*) of a path $P = \{f_0, ..., f_m\}$ in a DOMINO or NORA network is defined as:*

$$\lambda_P = \prod_{i=0}^{m} \lambda_P^{f_i} \tag{6.7}$$

where

$$\lambda_P^{f_i} = \left(\sum_{c_j \in q} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i - 1} \right) \left(\prod_{c_k \in r} (\sum_{g \in c_k} \lambda_1^{g,\tau_i - 1} + \overline{g}) \right) \tag{6.8}$$

and

$$\lambda_1^{g,t} = \sum_{Q \in \mathcal{P}_{g,t}} \lambda_Q \tag{6.9}$$

and

$$\lambda_2^{g,t} = f\left(\overline{\lambda_1^{g,t}} + \sum_{Q \in \mathcal{R}_{g,t}} \lambda_Q\right) \tag{6.10}$$

where, as before, $f_i = qf_{i-1} + r$.

The case of primary inputs should be treated carefully. Primary inputs can be treated simply as DOMINO nodes, with the case that:

$$\lambda_1^{g,t} = g \ \ \forall t \geq 0$$

and

$$\lambda_2^{g,t} = \begin{cases} g & t = 0 \\ 0 & t > 0 \end{cases}$$

This captures the behaviour that there is a rising edge on g iff g is true, and that g arrives at $t = 0$ for all g. This implies that the two phases of each primary input are treated as separate variables for the purpose of analysis.

We have immediately:

Theorem 6.3.1 *A path* $P = \{f_0, ..., f_m\}$ *in a precharged/unate network* N *is dynamically sensitized by input cube* c *iff* $\lambda_P(c) = 1$.

Proof:

$\Longleftarrow$ P dynamically sensitized by c. Induction on $\delta(f_m)$. Trivial if $\delta(f_m) = 0$. Assume for $\delta(f_m) < N$. Now, if P is dynamically sensitized by c, then we have f_m and f_{m-1} as in figure 6.1, and q, r defined as in the figure and subsequent discussion. By induction, we have that for each $g \in FI(f_m)$, and for each dynamically sensitizable path Q terminating in g, Q is dynamically sensitized by c iff $c \subseteq \lambda_Q$. Now, since there is a rising edge on f_m in response to a rising edge on f_{m-1}, we must have that there is a conducting path through q at τ_{m-1}. Hence for some cube c_j in the sum-of-products expression for q we must have that c_j is on at τ_{m-1}. In this case, the static value for each $g \in c_j$ is 1, and, further, g must have achieved its final value no later than τ_{m-1};

i.e., some path Q of length $\leq \tau_{m-1}$ terminating in g must be dynamically sensitized by c. By induction, therefore, $c \subseteq \lambda_2^{g,\tau_{m-1}}$, for $\lambda_2^{g,\tau_{m-1}}$ is the sum of the dynamic sensitization functions for such paths. Hence c satisfies

$$\sum_{c_j \in q} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_{m-1}}$$

Further, r is nonconducting at τ_{m-1}, i.e., $r = 0$ at τ_{m-1}. There are two cases. In the first case, the static value of r is 0. In this case the static value of every cube $c_k \in r$ is 0, i.e., for each cube c_k there is some $g \in c_k$ such that $\overline{g}$ is true, and we have that

$$\prod_{c_k \in r} \sum_{g \in c_k} \overline{g}$$

is satisfied. Otherwise, for some c_k, the static value is 1, i.e. every literal has its static value as 1. If every $g \in c_k$ has achieved its final value before τ_{m-1}, then a rising edge on f_m has occurred before τ_m, i.e., we have that c dynamically sensitizes a path through f_m of length $< \tau_m$. But we know that every path through f_m sensitized by c must be of the same length, and P is a path of length τ_m sensitized by c through f_m, contradiction. Hence for some $g \in c_k$, g settles to its final value at or after τ_{m-1}, i.e., there is a path Q of length $\geq \tau_{m-1}$ terminating in g sensitized by c. By induction, $c \subseteq \lambda_Q$. Now, $\lambda_Q \subseteq \lambda^{g,\tau_{m-1}}$, hence $c \subseteq \lambda^{g,\tau_{m-1}}$. Hence $c \subseteq \sum_{g \in c_k} \lambda^{g,\tau_{m-1}} + \overline{g}$ for each c_k, and hence is contained in the product. Hence $c \subseteq \lambda_P^{f_m}$. By induction, $c \subseteq \lambda_P^{f_i}$ for every $i < m$, and so $c \subseteq \lambda_P$.

$\Longrightarrow c \subseteq \lambda_P$. Induction on $\delta(f_m)$. For $\delta(f_m) = 0$, trivial. Assume for $\delta(f_m) < N$. If $\delta(f_m) = N$, we have by induction that $\{f_0, ..., f_{m-1}\}$ is sensitized by c. We must show that a rising event propagates from f_{m-1} to f_m under c. Write $f_m = qf_{m-1} + r$, as usual. Since $c \subseteq \lambda_P^{f_m}$, we have for some $c_j \in q$, c satisfies the right-hand side of (6.8), and so must satisfy:

$$c_j \prod_{g \in c_j} \lambda_2^{g,\tau_{m-1}}$$

By induction, the final value of each $g \in c_j$ is 1, and g arrives no later than τ_{m-1}. Hence when f_{m-1} toggles to 1, there is a conducting path through f_{m-1} and c_j. Hence if there is not a rising edge at f_m in response to a rising edge on f_{m-1}, there must be a conducting path through r before τ_{m-1}. But in this case, for some cube c_k in r, we must have that each g in c_k is turned on by c and, further, arrives before τ_{m-1}. By induction, therefore, we must have that $c \subseteq g$, and, further, that $c \subseteq \overline{\lambda_1^{g,\tau_{m-1}}}$ (since the one sensitizable path to g under c was of length $< \tau_{m-1}$. But then (6.5) is unsatisfied by c, and so too must be (6.8), and hence $\lambda_p^{f_m}$ is unsatisfied, contradiction. ■

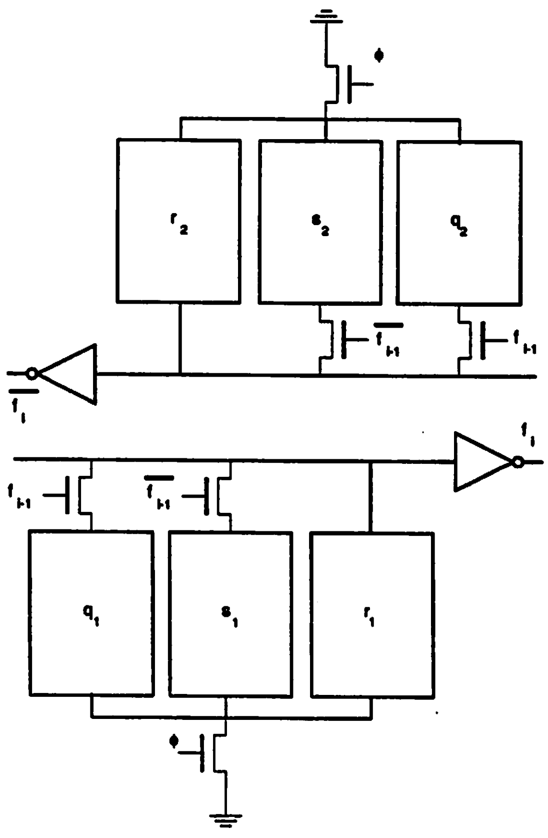


Figure 6.2: Generic DCVS Gate

Sensitization for DCVS circuits must be handled more carefully. A generic DCVS circuit can be mathematically analyzed as two DOMINO gates back-to-back, as in the diagram in figure 6.2. (In practice, the two pulldown networks of the two gates are often combined, for space reasons). Once again, the only cases that need be analyzed are rising edges on the inputs to a gate producing rising edges on the outputs of a gate. For such networks, equations (6.7) and (6.9)-(6.10) remain unchanged. Equation (6.8) splits into four equations, corresponding to the cases:

(i) a rising edge on f_{i-1} produces a rising edge on f_i;

(ii) a rising edge on f_{i-1} produces a rising edge on $\overline{f_i}$;

(iii) a rising edge on $\overline{f_{i-1}}$ produces a rising edge on f_i; and

(iv) a rising edge on $\overline{f_{i-1}}$ produces a rising edge on $\overline{f_i}$.

We analyze case *(i)* explicitly; the others are exactly analogous. For a rising edge on f_{i-1} to produce a rising edge on f_i, we must have that at τ_{i-1} there is a conducting path through q_1 and no conducting path through r_1. The other blocks may be neglected, since the presence of a rising edge on f_{i-1} assures us that there will be no conducting path to ground through $\overline{f_{i-1}}$ (and hence the possible presence of conducting paths through s_1 and s_2 is irrelevant). Similarly, the presence of a conducting path through q_1 ensures that there is no conducting path through either q_2 or r_2, since the mutual presence of such conducting paths through q_1 and either q_2 or r_2 would imply simultaneous rising edges on both f_i and $\overline{f_i}$, forbidden. Hence, note that (6.8) fits this situation exactly, with q_1 substituted for q and r_1 substituted for r.

$$\lambda_P^{f_i} = \left(\sum_{c_j \in q_1} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i-1} \right) \left(\prod_{c_k \in r_1} (\sum_{g \in c_k} \lambda_1^{g,\tau_i-1} + \overline{g}) \right) \tag{6.11}$$

The equation for case *(ii)* is:

$$\lambda_P^{f_i} = \left(\sum_{c_j \in q_2} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i-1} \right) \left(\prod_{c_k \in r_2} (\sum_{g \in c_k} \lambda_1^{g,\tau_i-1} + \overline{g}) \right) \tag{6.12}$$

The equation for case *(iii)* is:

$$\lambda_P^{f_i} = \left(\sum_{c_j \in s_1} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i-1} \right) \left(\prod_{c_k \in r_1} (\sum_{g \in c_k} \lambda_1^{g,\tau_i-1} + \overline{g}) \right) \tag{6.13}$$

The equation for case *(iv)* is:

$$\lambda_P^{f_i} = \left(\sum_{c_j \in 2_2} c_j \prod_{g \in c_j} \lambda_2^{g,\tau_i-1} \right) \left(\prod_{c_k \in r_2} (\sum_{g \in c_k} \lambda_1^{g,\tau_i-1} + \overline{g}) \right) \tag{6.14}$$

6.4 Algorithms

We now turn to algorithms. Again, the DOMINO case largely suffices, and only a few remarks need be made to cover the DCVS case.

As in the viability procedure, the basic algorithm is a dynamic programming procedure adapted from the best-first path tracing algorithm. The difference between $\psi_P^{f_i}$ and $\lambda_P^{f_i}$ requires that here both $\lambda_1^{f,t}$ and $\lambda_2^{f,t}$ be maintained for every node f.

Note that $\lambda_1^{f,t}$ is the analogue to $\psi^{f,t}$, and hence can be kept up to date in the same manner that $\psi^{f,t}$ was in the initial version of the viability procedure. To keep $\lambda_2^{f,t}$ up to date we need a little extra bookkeeping.

At the connection of node g into node f, we keep not one but two variables; f[g].λ_α and f[g].λ_β, and a variable f[g].t. The semantics of the variables are that f[g].λ_α holds the sensitization functions of all paths extending through g to f of length $>$ f[g].t, while f[g].λ_β holds the sensitization functions of all paths extending through g to f of length $=$ f[g].t. Hence

$$\lambda_1^{g,t} = f[g].\lambda_\alpha + f[g].\lambda_\beta$$

and

$$\lambda_2^{g,t} = \overline{f[g].\lambda_\alpha}$$

If we adapt the procedure of figure 3.15 to this problem, we have the procedure of figure 6.3.

The priority queue in this procedure is ordered by the relation $\succ$ developed in definition 3.3.1. The only difference between the two procedures is the name of the sensitization function (λ instead of ψ) and some details of the mechanism by which the dynamic programming information is computed. The order in which paths are examined is unchanged, and hence the scheduling theorems concerning algorithm 3.15 hold; in particular, lemmas 3.3.1-3.3.3 hold for this procedure. On this basis, we must derive the mechanism by which the dynamic programming information is updated.

From the scheduling theorems, it is clear that when we are extending a path

```
find_longest_true_path(){
    Initialize queue to primary inputs of the circuit
    while(((path, g) <- pop(queue)) $\neq$ nil) {
        k is the last node of path;
        update_lambda(g, k, path);
        $\lambda$ <- sensitization_function(path, g)
        if($\lambda \not\equiv 0$) {
            new_path <- {path, g};
            if(g is an output) return new_path;
            $\lambda$(new_path) <- $\lambda$;
            foreach extension np <- {new_path, h} of new_path
                if($E$(np) = $E$(new_path))
                    insert np on queue
        }
        if every extension ext of path s.t.
            $E$(ext) = $E$(new_path) has been explored {
            ext1 is next best extension of path
            push ext1 on queue;
            foreach extension ext2 of path with $E$(ext1) = $E$(ext2)
                push ext2 on queue;
        }
    }
}
```

Figure 6.3: Dynamic Programming Procedure to Find the Longest Sensitizable Path

P from f_{i-1} through f_i, it must be the case that, for each $g \in S(f_i, P)$:

$$f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta = \sum_{Q \in P_1(g,P,f_i)} \lambda_Q + \sum_{Q \in P_2(g,P,f_i)} \lambda_Q$$

where $P_1(g, P, f_i)$ is the set of paths through g and f_i that have already been explored and $P_2(g, P, f_i)$ is the set of paths through g and f_i on the queue and of maximal esperance, for these are the set of paths which are of length $\geq d(P)$. Further, from lemmas 3.3.2 and 3.3.3, we know that the set of paths in $P_2(g, P, f_i)$ is a superset of the paths through g to f_i of length $= d(P)$. Hence we must have:

$$f_i[g].\lambda_\alpha = \sum_{Q \in P_1(g,P,f_i), d(Q) > d(P)} \lambda_Q$$

and

$$f_i[g].\lambda_\beta = \sum_{Q \in P_1(g,P,f_i), d(Q) = d(P)} \lambda_Q + \sum_{Q \in P_2(g,P,f_i)} \lambda_Q$$

Further, those paths in $P_1(g,P,f_i)$ of length $d(Q) = d(P)$ must be the paths of minimal length in $P_1(g,P,f_i)$, and also must be the last explored. The variable $f_i[g].t$ is used to record this minimal length. As a new path Q is traced through g to f_i, it moves from $P_2(g,P,f_i)$ to $P_1(g,P,f_i)$, and must be a path of minimal length in $P_1(g,P,f_i)$. Hence the length of Q is checked; if it is *less* than $f_i[g].t$, we have a new lower bound length on $P_2(g,P,f_i)$, and we set:

$$\begin{aligned} f_i[g].t &= |Q| \\ f_i[g].\lambda_\alpha &= f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta \\ f_i[g].\lambda_\beta &= \lambda_Q \end{aligned}$$

otherwise, $|Q| = f_i[g].t$, we set

$$f_i[g].\lambda_\beta = f_i[g].\lambda_\beta + \lambda_Q$$

The code for `update_lambda` is given in figure 6.4.

`update_lambda` suffices to partition the sensitization functions of the paths $Q \in P_1(g,P,f_i)$. There remains the matter of assigning the sensitization functions of the paths $Q \in P_2(g,P,f_i)$. As in the procedure of figure 6.5, this is done in the procedure which computes the sensitization function.

Theorem 6.4.1 `sensitization_function`$(P, f_i) = \lambda_P^{f_i}$

Proof: Induction on $\delta(f_i)$. For the base case, f_i is a primary input, and $\lambda_P^{f_i} = 1$ for f_i a primary input. Assume for $\delta(f_i) < L$. Now, for $\delta(f_i) = L$ the theorem holds if we can show that:

$$f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta = \lambda_1^{g,\tau_i - 1}$$

```
update_lambda($f_i$, $g$, $Q$) {
    if ($d(Q) = f_i[g].t$)
        $f_i[g].\lambda_\beta \leftarrow f_i[g].\lambda_\beta + \lambda_Q$;
    else {
        $f_i[g].\lambda_\alpha \leftarrow f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta$
        $f_i[g].t \leftarrow d(Q)$;
        $f_i[g].\lambda_\beta \leftarrow \lambda_Q$;
    }
}
```

Figure 6.4: Procedure Updating Dynamic Programming Information

for the general case, and, further, that:

$$g\overline{f_i[g].\lambda_\alpha} = \lambda_2^{g,\tau_{i-1}}$$

For the first:

$$\lambda_1^{g,\tau_{i-1}} = \sum_{d(Q)\geq\tau_{i-1}} \lambda_Q$$

where Q is a partial path ending in g. Now, if $d(Q) > \tau_{i-1}$, or if the level of the last node of Q is less than that of the last node of P, then $\{Q, f_i\} \succ \{P, f_i\}$ by lemma 3.3.1, and hence has been examined previously by the algorithm. Since $\delta(g) < L$, by the induction hypothesis for each such path Q λ_Q was correctly calculated. Further, as each such Q was popped off the queue for extension through f_i, λ_Q was added into $f_i[g].\lambda_\beta$. Now, either it remained in $f_i[g].\lambda_\beta$ or was later transferred to $f_i[g].\lambda_\alpha$ and hence $f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta \supseteq \lambda_Q$ for all such paths Q. There remains the case where Q and P are incomparable under $\succ$. Now, if Q has been previously examined by the procedure, then λ_Q was added into $f_i[g].\lambda_\beta$ and hence $f_i[g].\lambda_\beta + f_i[g].\lambda_\alpha \supseteq \lambda_Q$. If not, by lemma 3.3.2, these are precisely the set of cases where $\{Q, f_i\}$ is on the queue and maximal under the order. These are the paths directly summed into $f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta$ in `sensitization_function`. Hence $f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta \supseteq \lambda_Q$ for

```
sensitization_function(path, g) {
    if g is a primary input return 1;
    k is the last node of path;
    sense_fn ← 0;
    esp ← $E$({path, g});
    foreach path p on queue of esperance esp {
        if p ends in j,g
            update_lambda(g, j, p);
    }
    foreach side input j to g {
        if g[j].t > d(path) {
            $g[j].\lambda_\alpha \leftarrow g[j].\lambda_\alpha + g[j].\lambda_\beta$;
            $g[j].\lambda_\beta \leftarrow 0$;
            g[j].t = d(path);
        }
    }   g = $qk + r$;
    foreach cube $c_i \in q$ {
        sense ← $c_i \prod_{k_1 \in c_i} \overline{g[k_1].\lambda_\alpha}$;
        sense_fn ← sense_fn + sense;
    }
    foreach cube $c_i \in r$
        sense_fn ← sense_fn * $\sum_{g \in c_i} (\overline{g} + g[k_1].\lambda_\alpha + g[k_1].\lambda_\beta)$;
    return sense_fn;
}
```

Figure 6.5: Sensitization Function for Dynamic Programming Procedure

each such path Q. For equality, all we must show is that no path Q has had its sensitization function λ_Q summed in incorrectly into $f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta$. This can only occur if the algorithm has examined some path Q before P with $d(Q) < d(P)$. But then $E(\{P, f_i\}) > E(\{Q, f_i\})$, contradiction.

To show that:

$$\overline{f_i[g].\lambda_\alpha} = \lambda_2^{g,\tau_{i-1}}$$

all we must show is that:

$$f_i[g].\lambda_\alpha = \sum_{\{Q,f_i\}>\tau_{i-1}} \lambda_Q$$

We already know that:

$$f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta = \sum_{d(Q)\geq\tau_{i-1}} \lambda_Q$$

By inspection of the code for the routine update_lambda, it is easy to see that $f_i[g].\lambda_\beta$ is the sum of the functions λ_Q for those Q of minimal length, and $f_i[g].\lambda_\alpha$ is the sum of the functions λ_Q for those Q of strictly greater than minimal length. Now, either the minimal length is τ_{i-1} or it is greater than τ_{i-1}. In the former case, the statement is proved, and in the latter the line

```
if g[j].t > d(path)
```

of procedure 6.5 is triggered, and we have that

$$\begin{aligned} f_i[g].\lambda_\alpha &= f_i[g].\lambda_\alpha + f_i[g].\lambda_\beta \\ f_i[g].\lambda_\beta &= 0 \end{aligned}$$

and hence

$$g\overline{f_i[g].\lambda_\alpha} = \lambda_2^{g,\tau_{i-1}}$$

and we are done. ∎

For DCVS, the same algorithm works; the sensitization functions for f_i and $\overline{f_i}$ must be separately maintained. Note, however, that since a rising edge on f_{i-1} can produce a rising edge on *either* f_i or $\overline{f_i}$, each iteration potentially adds two new paths to the queue, and hence the number of paths potentially grows very large. The bound in terms of the number of long false and true paths remains the same, and so the nominal complexity of the algorithm for DCVS remains as it is for DOMINO. However, the fact that a potentially larger number of long paths exists in a DCVS circuit should indicate that timing verification on such circuits may be in general somewhat more time-consuming. This effect may be somewhat mitigated by the fact that, in general, a DCVS circuit may require only half as many gates to realize the same function as a DOMINO circuit. Only experiment can answer these questions satisfactorily.

6.5 Conclusion

In this chapter, we have demonstrated that every dynamic sensitization is a robust, correct criterion on precharge-unate circuits and that the dynamic programming algorithm developed in chapter 3 can be modified to compute the critical paths of these circuits. These facts are a potential attraction of this design style, and in particular to the DCVS technology. These facts also mean that circuits designed with this style are less vulnerable to certain classes of stuck-at-fault, and hence may be considered, at least in this sense, somewhat more reliable than full static MOS.

Appendix A

Complexity Results

A.1 A Brief Introduction to the Theory of Polynomial Reducibility

The infant science that we today call Computer Science can be fairly said to have three recurrent, dominant themes: *abstraction*, *consistency* (often, incorrectly, called *correctness*) and *efficiency*. It is this last consideration which most strongly differs Computer Science from most other branches of mathematics. Other branches of mathematics are concerned with abstraction and consistency, but virtually never with efficiency.

Efficiency, in turn, has two sides, algorithm design and complexity. Algorithm design (also known as "upper bound theory") is concerned with finding good algorithms to solve a problem, and with characterizing the number of operations that those algorithms perform to solve an instance of a problem; complexity, or "lower bound theory" is concerned with demonstrating that any algorithm restricted to a class of operations (e.g., binary comparisons for sorting and searching) must require some number of those operations to solve an instance of a given problem. Both parts of the theory quote performance as a function of the problem size; e.g. $O(n \log n)$ binary comparisons are required to sort n items.

Lower bound theory is far less developed than upper-bound theory. While some problems (e.g. sorting n items, searching an ordered set of n items for a given item) have well-defined lower bounds that are "sharp" (i.e., there is a known algorithm that performs as well as the lower bound), these are very much the exception rather than the rule. Further, the mathematical techniques used to prove lower bounds are still very much in their infancy.

$$\begin{tikzcd}
\mathbf{P} \arrow[d] & & \mathbf{P\ Solution} \\
\mathbf{Q} \arrow[rr] & & \mathbf{Q\ Solution} \arrow[u]
\end{tikzcd}$$

Figure A.1: Generic Problem Transformation

One technique that has gained much currency over the past 15 years is the use of *reducibility*, or of *problem transformations*. The use of a problem transformation is shown graphically in figure A.1. Broadly, algorithms are found to transform an instance of problem P into an instance of problem Q, and the solution of the instance of problem Q into the solution of the instance of problem P; if a lower bound for problem P is known, a lower bound for problem Q can then be derived in terms of the lower bound for problem P and the cost of the transformation, since the use of the transformation and an algorithm for Q is an algorithm for P.

In the early 1970's, the use of problem transformation was to formalize research in a class of problems for which neither any good algorithm nor lower bound was known. The paradigmatic problem for this class was the **travelling salesman** problem: given a set of n cities, and, for each pair of cities i, j an integer cost w_{ij} associated with a transit from city i to city j, and an integer K, is there a tour of the n cities, in which each city is visited only once, such that the total cost of the tour is $\leq K$?

The interesting thing about travelling salesman (TS) is that, given a proposed tour, one can easily validate whether or not the proposed tour is in fact a tour and one can easily determine whether its total cost is $\leq K$; in fact, one can do so in time linear in the number of cities. However, discovering a solution is *believed* to be much harder. Currently, the best-known algorithm is still believed to be exponential in the number of cities.

In the early 1970's Cook observed [21] that the convolution product, and sum of polynomials all produced polynomials; hence, referring to the diagram in figure A.1, if the transformation from P into Q and from Q's solution to P's solution were polynomial[1], then the existence of a polynomial-time algorithm for Q would imply the existence of a polynomial-time algorithm for P. Conversely, the existence of an exponential-time lower bound for P would imply an exponential-time lower bound for Q.

Cook then considered the class of problem whose solution could be verified in polynomial time (e.g., travelling salesman), a class we now call $\mathcal{NP}$. He demonstrated that every problem in $\mathcal{NP}$ could be polynomially transformed to the problem of determining whether or not there was a satisfying assignment to a logic formula in conjunctive normal form; a problem call SAT. Hence SAT was the "hardest" problem in $\mathcal{NP}$, in the sense that a polynomial-time solution for SAT would imply a polynomial-time solution for any problem in $\mathcal{NP}$, and. conversely, that an exponential-time lower bound for some problem in $\mathcal{NP}$ would imply an exponential-time solution for SAT. Cook further speculated that other problems in $\mathcal{NP}$ might be as hard as SAT, i.e., there might be problems in $\mathcal{NP}$ into which SAT could be polynomially transformed.

In 1972 Karp [43] presented a fairly comprehensive set of such problems. Since then, a fairly rich set of such problems has been compiled. In 1979, the basic text on the area [28] presented a list of 320 problems known to be $\mathcal{NP}$-complete; i.e. as hard as SAT and polynomially-verifiable.

It is clear that the class of problems which may be solved in polynomial time

[1]Both in the sense that the transformation from P to Q takes polynomial time and in the sense that an instance of P of size n produces an instance of Q of size $p(n)$ for polynomial p

($\mathcal{P}$) is a subset of the set of problems which may be *verified* in polynomial time ($\mathcal{NP}$), i.e., it is clear that $\mathcal{P} \subseteq \mathcal{NP}$. It is not known whether this containment is proper, though it is widely believed to be the case. If the containment is in fact proper, then all the $\mathcal{NP}$-complete problems must be super-polynomial in complexity.

$\mathcal{NP}$-complete problems are not the last word in complexity. There is a large class of problems *known* to be intractable – that is, a class of problems for which any algorithm must take exponential time. A further interesting class are those problems polynomially reducible to an $\mathcal{NP}$-complete problem but whose verification procedures are not known to be in $\mathcal{P}$ (and, in general, are known not to be in $\mathcal{P}$ unless $\mathcal{P}= \mathcal{NP}$). Procedures polynomially reducible to SAT are called $\mathcal{NP}$-hard: they are as hard as any problem in $\mathcal{NP}$. $\mathcal{NP}$-hard problems which are also known to be polynomially verifiable are called $\mathcal{NP}$-complete.

Proofs of $\mathcal{NP}$-hardness of a problem are in some sense lower bound results: they demonstrate that a lower bound on the problem is polynomially related to the maximum lower bound of the problems in $\mathcal{NP}$. Similarly, a proof of membership in $\mathcal{NP}$ is an upper bound result: it shows that an upper bound for this problem is polynomially related to an upper bound on SAT. With this in mind, we proceed to the complexity results.

In the remainder of this appendix, we will often be using the phrase "polynomial time" or "linear time". Of course, either phrase is meaningless without a referent: polynomial means polynomial in *what*? The size measure we will be using for instances is the sum, over all the nodes, of the number of literals appearing in the disjoint normal form of each node.

A.2 Sensitizability of the Longest Path is $\mathcal{NP}$-hard

We go now to our basic complexity result, which shows that all the problems associated with path sensitization are $\mathcal{NP}$-hard; i.e., that this problem is not significantly easier than any problem in $\mathcal{NP}$. The specific subproblem that we choose to show is $\mathcal{NP}$-hard is **Long Path Sensitization:** given a network N, is the longest path through N sensitizable? Since this is obviously a special case of the general false path problems we have been addressing, and since the fact that a generalization of an $\mathcal{NP}$-hard problem is another $\mathcal{NP}$-hard problem, showing this result suffices to show that all the problems of significance that we are addressing are $\mathcal{NP}$-hard.

In order to show that the problem **Long Path Sensitization** is $\mathcal{NP}$-hard, we need an $\mathcal{NP}$-hard problem which we can transform into this problem. We choose the well-known $\mathcal{NP}$-complete problem **3SAT**

3SAT is SAT restricted to three literals per clause; i.e., given a formula:

$$(u_{11} + u_{12} + u_{13})(u_{21} + u_{22} + u_{23})...(u_{m1} + u_{m2} + u_{m3}) \tag{A.1}$$

where each u_{ij} is a literal of some boolean variable $\{x_1, ..., x_k\}$, is there an assignment to the variables $\{x_1, ..., x_k\}$ such that formula A.1 is satisfied, i.e., evaluates to 1?

3SAT was shown to be $\mathcal{NP}$-complete, and hence also $\mathcal{NP}$-hard, in [43]; it is one of the original classic $\mathcal{NP}$-complete problems.

We now transform 3SAT into Longest Path Sensitization.

Problem A.2.1 [Longest Path Sensitization] **Instance:** A network N, with a unique longest path P.
Question: Is the unique longest path $P \in N$ sensitizable?

This will suffice to show that all problems investigated in this thesis are $\mathcal{NP}$-hard, since we will show that this is a special case of each such problem.

Theorem A.2.1 *LPS is $\mathcal{NP}$-hard*

Proof: Reduction from 3SAT. Given an instance of 3SAT:

$$(u_{11} + u_{12} + u_{13})(u_{21} + u_{22} + u_{23})...(u_{n1} + u_{n2} + u_{n3})$$

where each u_{ij} is a literal drawn from the set $\{x_0, ..., x_k, \overline{x_0}, ..., \overline{x_k}\}$, we construct the circuit shown in figure A.2. If the delay on each gate is 1, and the delay on the static

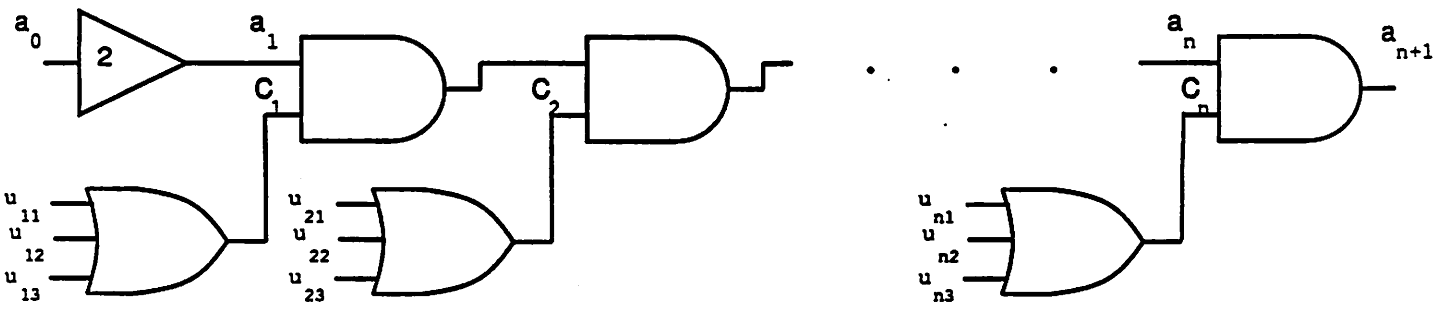


Figure A.2: LPS Transformation from 3SAT

delay buffer is 2 as marked, then the longest path through this graph is $\{a_0, ..., a_{n+1}\}$. Since the longest path is unique, we have that its dynamic sensitization function is:

$$\prod_{i=1}^{n+1} \frac{\partial a_i}{\partial a_{i-1}}$$

Further, we have:

$$\frac{\partial a_i}{\partial a_{i-1}} = a_{i a_{i-1}} \oplus a_{i \overline{a_{i-1}}} = C_i \oplus 0 = C_i = (u_{i1} + u_{i2} + u_{i3})$$

Hence the path is sensitizable iff the instance of 3SAT is satisfiable. Further, the transformation is obviously polynomial (in fact, linear). Since this was an arbitrary instance of an $\mathcal{NP}$-hard problem, LPS is $\mathcal{NP}$-hard. ∎

We now use this basic result to demonstrate that longest statically sensitizable path is $\mathcal{NP}$-complete, as is longest viable path and longest dynamically sensitizable path.

A.3 Longest Statically Sensitizable Path is $\mathcal{NP}$-complete

Problem A.3.1 [Longest Statically Sensitizable Path] **Instance:** A network N, real K.

Question: Is there a path $P \in N$, input minterm c, P statically sensitized by c, $d(P) \geq K$?

Theorem A.3.1 *LSSP is $\mathcal{NP}$-hard.*

Proof: Immediate from the fact that LPS is $\mathcal{NP}$-hard, since a network with a unique longest path of length K_1 has a statically sensitizable path of length K_1 iff the longest path in such a graph is sensitizable. ■

Further, LSSP clearly $\in \mathcal{NP}$, since, given an input minterm c and a path $P = \{f_0, ..., f_m\}$, one can easily verify in linear time that $d(P) \geq K$ and that $\frac{\partial f_i}{\partial f_{i-1}}$ is statically satisfied for every i by direct simulation. Hence LSSP is $\mathcal{NP}$-complete.

A.4 Longest Dynamically Sensitizable Path is $\mathcal{NP}$-complete

Problem A.4.1 [Longest Dynamically Sensitizable Path] **Instance:** A network N, real K.
Question: Is there a path $P \in N$, and input minterms c_1, c_2, P statically sensitized by c_1 and c_2, $d(P) \geq K$?

Theorem A.4.1 *LDSP is $\mathcal{NP}$-hard.*

Proof: Immediate from the fact that LPS is $\mathcal{NP}$-hard, since a network with a unique longest path of length K_1 has a dynamically sensitizable path of length K_1 iff the longest path in such a graph is statically sensitizable. ■

Further, LDSP clearly $\in \mathcal{NP}$, since, given an input minterm c and a path $P = \{f_0, ..., f_m\}$, one can easily verify in linear time that $d(P) \geq K$ and that $\frac{\partial f_i}{\partial f_{i-1}}$ is dynamically satisfied at τ_{i-1} for every i by direct simulation. Hence LDSP is $\mathcal{NP}$-complete.

A.5 Longest Viable Path is $\mathcal{NP}$-complete

This result is quite surprising. LVP is obviously $\mathcal{NP}$-hard, since LPS is a special case of LVP. However, few (including the author) would have been prepared to believe that a polynomial verification procedure existed for LVP. The proof that such a verification procedure exists is interesting, as well; generally, proofs of this nature involve the construction of an algorithm, which is then proved correct and polynomial time. The proof herein, however, merely demonstrates that such a procedure exists, and does not give details of the construction.

Problem A.5.1 [Longest Viable Path] **Instance:** A network N, real K.
Question: Is there a path P, minterm c, such that P is viable under c and $d(P) \geq K$?

Problem A.5.2 [Longest Viable Path Verification] **Instance:** A network N, minterm c, node g, real K.
Question: Is there a path $P = \{g_0, ..., g\}$, $d(P) \geq K$, viable under c?

Lemma A.5.1 *LVP Verification is* $\in \mathcal{P}$.

Proof: Induction on $\delta(g)$. If $\delta(g) = 0$, then trivial, for every path consisting only of a primary input is viable under any minterm c, and is of well-defined length. Assume for $\delta(g) < L$. If $\delta(g) = L$, then there is such a path $P = \{g_0, ..., g_r, g\}$ iff

1. $d(\{g_0, ..., g_r\}) \geq K - w(g)$ and
2. $\{g_0, ..., g_r\}$ is viable under c and
3. For each input $g_j \neq g_r$ of g, either c sets g_j to its sensitizing value or g_j terminates a path, viable under c, of length at least $d(\{g_0, ..., g_r\})$

Now, by induction, for each input y_j of g, we can determine whether there is a path of length $\geq K - w(g)$ viable under c terminating in y_j, and we can do so in polynomial time. There are only a polynomial number of inputs to g, and hence in polynomial time one can determine the existence of the set V of inputs to g with the property that, for each $y_j \in V$, y_j terminates a path of length $\geq K - w(g)$ viable under c. A viable path under c terminating in g of length at least K therefore exists iff, for the $y_j \in V$ terminating a path of minimal length, we have that c satisfies $\mathcal{S}_{V-y_j}\frac{\partial g}{\partial y_j}$. Now, by the viable fork lemma, if c satisfies $\mathcal{S}_{V-y_j}\frac{\partial g}{\partial y_j}$ for some y_j it does so for each y_j, and hence one can choose the y_j to test arbitrarily. This test can certainly be done in polynomial time. ∎

Theorem A.5.1 *LVP* $\in \mathcal{NP}$

Proof: Suppose we are given a path $P = \{f_0, ..., f_m\}$, minterm c. We can easily verify that $d(P) \geq K$, so all we must do is demonstrate that it is polynomial to verify that $c \in \psi_P$. For this, all we must do is show that it is polynomial to verify that $c \in \psi_P^{f_i}$ for every i, for there are only a polynomial number of such functions. Now, if $c \in \psi_P^{f_i}$, c must satisfy some term of:

$$\sum_{U \subseteq S(f_i,P)} (\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \psi^{g,\tau_i-1}$$

Now, let V be the unique maximum subset of $S(f_i, P)$ such that $c \in \psi^{g,\tau_i-1}$ for each $g \in V$. By lemma A.5.1 the fact that $c \in \psi^{g,\tau_i-1}$ can be determined in polynomial time for each g, and hence the determination of V is polynomial by the boundedness of $S(f_i, P)$. It is easy to demonstrate that c satisfies

$$\sum_{U \subseteq S(f_i,P)} (\mathcal{S}_U \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in U} \psi^{g,\tau_i-1}$$

iff c satisfies

$$(\mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}}) \prod_{g \in V} \psi^{g,\tau_i-1}$$

and hence iff c satisfies

$$\mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}}$$

The calculation of $\mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}}$ is easily made in polynomial time, and, further, the determination that c satisfies $\mathcal{S}_V \frac{\partial f_i}{\partial f_{i-1}}$ is easily made in polynomial time by direct simulation. ∎

This theorem, together with the theorems which demonstrate that LVP is $\mathcal{NP}$-hard, demonstrates that LVP is $\mathcal{NP}$-complete.

Appendix B

A Family of Operators

The Boolean difference and the smoothing operator, explored earlier, can be thought of as two members of a family of operators involving the cofactors. Each of these operators reduces the dimension of the space by one variable, but the semantics of the operators vary. The character of this family can be divined by examining the formulae for the boolean difference:

$$\frac{\partial f}{\partial x} = f_x \oplus f_{\overline{x}}$$

and of the smoothing operator:

$$S_x f = f_x + f_{\overline{x}}$$

The hint here is that for each of the 16 two-variable boolean functions, there should be a separate cofactor operator. The purpose of this appendix is to enumerate these operators, describe their function and their interrelationship. This taxonomy is not particularly useful, but it does serve to beautify science.

The sixteen dyadic boolean functions are as described in table B.1. these functions correspond to operators as described in table B.2, letting f_x stand for x and $f_{\overline{x}}$ stand for y Now, it is relatively clear through an examination of these operators that these operators are related in a fairly rich way. In particular, consider the *duality* relation: an operator O is the dual of an operator O' iff $O_x f = O'_x \overline{f}$. Clearly duality is a symmetric relationship. We can write the duality table in table B.3, omitting trivial operators:

x	y	0	1	$\overline{x}$	$\overline{y}$	xy	$x+y$	$\overline{xy}$	$\overline{x+y}$	$x \geq y$	$x > y$	$x \leq y$	$x < y$	$x \overline{\oplus} y$	$x \oplus y$
0	0	0	1	1	1	0	0	1	1	1	0	1	0	1	0
0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	1
1	0	0	1	0	1	0	1	1	0	1	1	0	0	0	1
1	1	0	1	0	0	1	1	0	0	1	0	1	0	1	0

Table B.1: Dyadic Boolean Functions

Function	Operator	Semantic
x	f_x	f evaluated at $x = 1$
y	$f_{\overline{x}}$	f evaluated at $x = 0$
$\overline{x}$	$\overline{f_x}$	$\overline{f}$ evaluated at $x = 1$
$\overline{y}$	$\overline{f_{\overline{x}}}$	$\overline{f}$ evaluated at $x = 0$
0	0	0
1	1	1
xy	$\mathcal{C}_x f = f_x f_{\overline{x}}$	$f = 1$ for each value of x
$x + y$	$\mathcal{S}_x f = f_x + f_{\overline{x}}$	$f = 1$ for some value of x
$\overline{xy}$	$\overline{\mathcal{S}_x f}$	$f = 1$ for no value of x
$\overline{x+y}$	$\overline{\mathcal{C}_x f}$	$f = 0$ for some value of x
$x \geq y$	$\mathcal{I}_x f$	Set of points where f is monotone increasing in x
$x > y$	$\mathcal{IP}_x f$	Set of points where f is strictly monotone increasing in x
$x < y$	$\mathcal{DP}_x f$	Set of points where f is strictly monotone decreasing in x
$x \leq y$	$\mathcal{D}_x f$	Set of points where f is monotone decreasing in x
$x \oplus y$	$\frac{\partial f}{\partial x}$	Set of points where f is determined by x
$x \overline{\oplus} y$	$\overline{\frac{\partial f}{\partial x}}$	Set of points where f is independent of x

Table B.2: Dyadic Boolean Functions and their Corresponding Operators

Operator	Dual
$\mathcal{C}_x f$	$\mathcal{S}_x f$
$\mathcal{I}_x f$	$\mathcal{D}_x f$
$\mathcal{IP}_x f$	$\mathcal{DP}_x f$
$\frac{\partial f}{\partial x}$	$\frac{\partial f}{\partial x}$
$\overline{\frac{\partial f}{\partial x}}$	$\overline{\frac{\partial f}{\partial x}}$

Table B.3: Duality Table of Operators

Note that only eight of the 16 operators are mentioned in the duality table. This follows from the observation that the operators 0, 1, f_x, $\overline{f_x}$, $f_{\overline{x}}$, $\overline{f_{\overline{x}}}$ are trivial, and that the duality properties of the operators $\overline{S_x f}$ and $\overline{C_x f}$ are adequately captured elsewhere in the table.

Appendix C

Fast Procedures for Computing Dataflow Sets

C.1 Introduction

In computing whether paths are true by some sensitization criterion, we assert values of nodes in a multi-level network and then determine whether some input vector can **justify** the assertions that we make. If such a vector exists, we say that the multi-level function implicitly expressed by these assertions is **satisfiable**. In previous appendices, we have seen that in general this is a hard problem.

Now, it is clear that any function which requires both that y and $\bar{y}$ be true for some variable y cannot be satisfiable; such a function is said to have an **explicit incompatibility**. Hence one approach to the satisfiability problem is simply to determine whether a function has an explicit incompatibility, and delare it unsatisfiable only if it does. Note that this is an inexact procedure: a function may not have an explicit incompatibility but still be unsatisfiable. Hence this is a *biased* SAT test: all satisfiable functions are reported as satisfiable, but some unsatisfiable functions may be reported as satisfiable. However, referring to the approximation spectrum in 4.1, it is clear that this is a *positively-biased SAT test*, which, as we reported there, is a safe approximation to SAT: using this in a false-path detection algorithm will not result in a true path being reported as false. This appendix details a fast ($O(n^4)$)

procedure for computing such a safe approximation. It computes the implications of any assertion.

Dataflow computations for optimizing Boolean logic networks are well known [15, 7, 76, 35]. Such analysis computes inferences of the form $y_i = b_i \Rightarrow y_j = b_j$ for nodes y_i, y_j and b_i, b_j in $\{0,1\}$. In the literature, the set $\mathcal{F}_{ij}(x)$ is used to represent the set of nodes y_k s.t. y_k is set to j when x is set to i.

Now, a polynomial algorithm to compute these sets exactly for general networks obviously implies a polynomial solution to the co-$\mathcal{NP}$-complete problem of tautology (setting any primary input to either 1 or 0 sets the output x to 1 iff the network is tautologous, and hence $x \in \mathcal{F}_{11}(y_k) \cap \mathcal{F}_{01}(y_k) \forall y_k$, where y_k is a primary input iff the network is tautologous).

For this reason, the sets $\mathcal{F}_{ij}$ are not computed. Berman, Trevillyan and Joyner [7] have defined, on networks of NOR gates, the interesting subsets $C_{ij}(x) \subseteq \mathcal{F}_{ij}(x)$. These are defined by the rules:

$$\begin{aligned}
&y \in C_{10}(x) \quad \text{if} \quad \exists t \in C_{11}(x) \text{ and } y \text{ is a fanout of } t && \text{(C.1)}\\
&y \in C_{10}(x) \quad \text{if} \quad \exists t \in C_{11}(x) \text{ and } y \text{ is a fanin of } t && \text{(C.2)}\\
&y \in C_{10}(x) \quad \text{if} \quad \exists t \in C_{10}(x) \text{ and } y \text{ is the only fanin of } t \text{ not in } C_{10}(x) && \text{(C.3)}\\
&y \in C_{10}(x) \quad \text{if} \quad \exists t \in C_{11}(x) \text{ and } y \in C_{10}(t) && \text{(C.4)}\\
&y \in C_{10}(x) \quad \text{if} \quad \exists t \in C_{10}(x) \text{ and } y \in C_{00}(t) && \text{(C.5)}\\
&y \in C_{10}(x) \quad \text{if} \quad x \in C_{10}(y) && \text{(C.6)}
\end{aligned}$$

The subsets are computed by finding the least fixed point of the recurrence relations given by these rules and the analogous rules for C_{11}, C_{00}, and C_{01}. Initially, $C_{11}(x) = x$, $C_{00}(x) = \overline{x}$, and $C_{01}(x) = C_{10}(x) = \emptyset$ for all x.

The four rules are generally self-explanatory. The first rule (the *fanout* rule) captures the fact that in a network of nor gates, setting $t = 1$ sets all fanouts of t to 0. The second *fanin* rule is derived from the observation that if t is set to 1, then all of its fanins are set to 0. The third (another fanin) rule captures the fact that if $t = 0$, at least one of its inputs must be 1. Rules 4 and 5 transitively close the sets.

Rule 6 captures the well-known rule of deduction $x \Rightarrow \overline{y}$ is equivalent to $y \Rightarrow \overline{x}$

While this approach has demonstrated some power, nevertheless some improvements are desirable. First, it is tedious to develop new rules for each sort of gate, and for combinations of gates. Second, one wishes an explicit algorithm for the computation of these sets, with some hope that it is efficient.

This appendix develops a fast procedure to compute these sets, and a generalization to all boolean networks.

C.2 Terminology

Recall from chapter 1 that a product of literals is called a *cube*. A cube may also be viewed as a *set* of literals; the cube xyz is equivalent to the set $\{x, y, z\}$. This equivalence of sets and cubes permits us to regard the sets $C_{ij}(x)$ as cubes $C_{ij}(x)$. This permits a new, generalized approach to the derivation of C-sets.

In the derivation, we will be using the cofactor notation a great deal; we remind the reader here that f_c refers to the function f evaluated on the space defined by the cube c.

C.3 The New Approach

The traditional division of the dataflow implications into four sets is an artifact of the traditional nor- or nand-gate formulation. In fact, for each x, the sets of interest are the sets of literal values implied by choosing either $x = 1$ or $x = 0$. By using both phases, one can take the union of $C_{10}(x)$ and $C_{11}(x)$ as the set C_x (the set of values implied by $x = 1$), and the union of $C_{00}(x)$ and $C_{01}(x)$ as the set $C_{\overline{x}}$ (the set of values implied by $x = 0$).

We define C_x ($C_{\overline{x}}$) as the fixed points of the set sequence C_x^i ($C_{\overline{x}}^i$), where $C_x^0 = \{x\}$, $C_{\overline{x}}^0 = \{\overline{x}\}$, and C_x^{n+1} is obtained from C_x^n by the relations:

$$y \in C_x^n \quad \text{if} \quad y_{C_x^{n-1}} = 1 \tag{C.7}$$

$$y \in C_x^n \quad \text{if} \quad \exists t \in C_x^{n-1} \text{ and } t_{C_x^{n-1}} = yf \text{ some } f \tag{C.8}$$

$$y \in C_x^n \quad \text{if} \quad \exists \bar{t} \in C_x^{n-1} \text{ and } t_{C_x^{n-1}} = \bar{y} + f \text{ some } f \tag{C.9}$$

$$y \in C_x^n \quad \text{if} \quad \exists t \in C_x^{n-1} \text{ and } y \in C_t^{n-1} \tag{C.10}$$

$$y \in C_x^n \quad \text{if} \quad \bar{x} \in C_{\bar{y}}^n \tag{C.11}$$

And, of course, the symmetries obtained by substituting $\bar{y}$, and/or $\bar{t}$ for t, and/or $\bar{x}$ for x. Relation (C.7) is the usual fanout rule; relations (C.8) and (C.9) capture the fanin rule; and relation (C.10) is transitive closure. Relation (C.11) is the contrapositive. Note that these rules simplify to the well-known nor-gate rules for networks consisting only of nor gates.

Lemma C.3.1 *Let $y(\bar{y}) \in C_x$, C_x is obtained as a fixed point of relations (C.7)-(C.11). Then y is set to 1 (0) whenever x is set to 1, and the similar observation holds for $C_{\bar{x}}$.*

Proof: WLOG, we consider y only in the positive phase, and x in the positive phase; the other three cases follow by symmetry. We prove by induction on n, the level at which y is added to the set C_x (that is, $y \in C_x^n - C_x^{n-1}$. The base case is trivial (x is clearly set to 1 when x is set to 1), so suppose the statement holds for all $z \in C_w^{n-1}$, $\forall w$. Now, y is added at n, and must be added by one of (C.7)-(C.11). If by (C.7), then $y_{C_x^n} = 1$, and since the settings are correct in C_x^n by the inductive assumption, y is certainly set to 1 when x is set to 1. If by (C.8), then $\exists t \in C_x^{n-1}$ and $t_{C_x^{n-1}} = yf$. By the inductive assumption t is set to 1 when x is set to 1, and $t_{C_x^{n-1}} = yf$, hence for $t = 1$ we must have $y = 1$, whence we must set y to 1. If by (C.9), then $\exists \bar{t} \in C_x^{n-1}$ s.t. $t_{C_x^{n-1}} = \bar{y} + f$. For $t = 0$, as required, we must have $\bar{y} = 0$, whence y must be set to 1. If by (C.10), then we have $w = 1$ from $x = 1$ and $y = 1$ from $w = 1$, whence $x = 1$ implies $y = 1$. If by (C.11), we have that $y = 0 \Rightarrow x = 0$. Hence if $x = 1$ we must have $y = 1$, for if $y = 0$ then $x = 0$, and so done. ■

We now turn to the computation of the sets. The details of this computation are important for the efficiency of the algorithm.

C.4 Computations

The preliminary observation that we make before we begin the algorithms is the duality between cubes and sets. Using this duality, we can store the sets C_x^n and $C_{\bar{x}}^n$ as cubes, and use the logic operations of cofactoring for node evaluation and boolean AND to find the union of two sets.

In this and subsequent code, it is important to understand precisely the difference between a *variable* and a *literal*. Strictly speaking, a literal is the instance of a variable in either of its phase; it may be thought of as a pair (variable, phase), where phase is in {0,1}. By abuse of notation, a literal is generally represented by the appropriate variable in its positive phase; thus is brevity the enemy of precision.

We keep to this convention here. Except where explicitly noted, all arguments to the functions developed below are literals, and hence may be in either phase, though they will always appear, by convention, in positive phase. Further, the sets C_x^n, which will be represented by the variables C_x, are indexed by literals and not variables.

C.4.1 Basic Algorithms

The two fundamental procedures are the fanout and fanin evaluation procedures. The former attempts to discover nodes that are set to a constant under the cofactoring operation; the latter attempts to discover nodes which have non-trivial cube factors under the cofactoring operation.

The fanout evaluation procedure returns 0, 1, or 2, according as to whether y is set to 0, 1, or neither by the cofactoring process. y in this code is a node, not a literal.

```
evaluate_node(y, x)
{
    df_cube = $C_x$;
    eval_node = cofactor(y, df_cube);
    if(eval_node == 1) return 1;
    else if(eval_node == 0) return 0;
    else return 2;
}
```

The fanin procedure returns the common cube dividing the cofactored cube. The process is relatively straightforward: the literal y is known to be set when the literal x is set. For the moment, assume that the phase of y is positive. If y is set to a product of a cube and some function by the cofactoring process (in other words, when the cubes of the cofactored node have a non-trivial intersection), then the literals of cube must be set appropriately to set cube to 1.

Now consider the case where y is in its negative phase. Hence we must have $\overline{y} = 1$ under C_x, and using the fact that $\overline{(f_c)} = \overline{f}_c$, we can run the described algorithm on $\overline{y}$

```
evaluate_fanin(y, x)
{
    if(y is in positive phase) eval_node = $y_{C_x}$;
    else eval_node = $\overline{y}_{C_x}$;
    fanin_cube = get_common_cube(eval_node);
    return common_cube;
}
```

C.4.2 Transitivity

The results of the preceding section suffice to capture the fanin and fanout rules, respectively. There remains the matter of transitive closure. Immediate transitivity can be guaranteed by rewriting the fanout and fanin rules as follows:

$$C_x^{n+1} \supseteq C_y^n \quad \text{if} \quad y_{C_x^n} = 1 \tag{C.12}$$

$$C_x^{n+1} \supseteq C_y^n \quad \text{if} \quad \exists t \in C_x^n \text{ and } t_{C_x^n} = yf \text{ some } f \tag{C.13}$$

(with the usual symmetries). The contrapositive rule is rewritten:

$$C_x^{n+1} \supseteq C_y^n \quad \text{if} \quad \overline{x} \in C_{\overline{y}}^n \tag{C.14}$$

Put bluntly, the entire set of literals C_y^n is included in C_x^{n+1}, rather than simply y. The reader can easily verify that this is correct as an immediate consequence of the previously-given transitive closure rules. This is accomplished by the following code:

```
merge_df_cubes(x, y)
{
    if($C_x \supseteq C_y$) return 0;
    else {
        $C_x = C_x \cup C_y$;
        return 1;
    }
}
```

Note that merge_df_cubes returns 1 only if $C_x^{n+1} \neq C_x^n$.

Let us quickly consider the matter of the contrapositive. This can be handled most naturally by merging $C_{\overline{x}}$ into $C_{\overline{y}}$ whenever C_y is merged into C_x, as implied by (C.14).

There remains the matter of further transitivity. Certainly C_x^{n+1} is transitively closed by the operations given above. However, if x appears in the positive phase in C_z^n for some z, then the transitivity closure rules require that $C_z^{n+2} \supseteq C_x^{n+1}$. Now, if $C_x^n = C_x^{n+1}$, this is assured inductively. The difficulty arises if C_x^{n+1} has changed. In this case, we must propagate its change to all of the dataflow cubes where x appears in the positive phase. This is done by maintaining sets of cubes D_x and $D_{\overline{x}}$ for each node x, where D_x is the set of cubes C_y containing x, and $D_{\overline{x}}$ the cubes C_z containing $\overline{x}$.

```
propagate_dataflow(x)
{
    foreach $y \in D_x$ {
        merge_df_cubes(y, x);
        merge_df_cubes($\overline{x}, \overline{y}$);
    }
}
```

The maintenance of the sets D_z requires a change to merge_df_cubes; for if z is in C_y^n and is not in C_x^n, then C_x^{n+1} must be added to D_z.

```
merge_df_cubes(x, y)
{
    if($\mathcal{C}_x \supseteq \mathcal{C}_y$) return 0;
    else {
        $\mathcal{C}_x = \mathcal{C}_x \cup \mathcal{C}_y$;
        foreach $z \in \mathcal{C}_y$
            $D_z = D_z \cup \{\mathcal{C}_x\}$
        return 1;
    }
}
```

C.4.3 When $\mathcal{C}_x = 0$

There remains the matter of the interpretation of zeroing one of the C_x^i. This occurs iff, for some variable y, $y \in C_x^i$ and $\overline{y} \in C_x^i$; hence $x = 1 \Rightarrow y = 1$ and $y = 0$. This is impossible, hence if $C_x^i = 0$, then x cannot be set to 1. Since x cannot be set to 1, then all of the other variable settings which imply $x = 1$ are also impossible, and hence their corresponding cubes must be set to 0. These are the cubes contained in the transitive closure of D_x.

```
propagate_zero(x)
{
    stack = tfo_collect(x);
    while((z = pop_stack(stack)) != nil)
        $C_z = 0$;
}
```

The transitive-fanout collection procedure is a standard graph traversal through the edges implied by the sets D_y. propagate_zero is called from propagate_dataflow if the propagated cube is ever found to be 0.

C.4.4 Evaluation Algorithms

There remains the question of evaluation. Literals may be added to a C set either through propagation or evaluation. Propagation has been adequately covered above. We turn to evaluation. The core of the evaluation strategy is in the following lemma and definition.

Definition C.4.1 *If some variable y is set to 1 (0) by either the routine* evaluate_node *or the routine* evaluate_fanin*, under the implications of some dataflow cube C_x^n, we say that y is* **implied** *by C_x^n.*

Now, clearly, at each iteration of the evaluation algorithm, we only wish to examine the variables which *may* be implied by C_x^n. We isolate these *potential implicants* of C_x^n by the following lemma.

Lemma C.4.1 *Let y be an implicant of C_x^n, y is* not *an implicant of C_x^{n-1}. Then y is a fanin or a fanout of some literal in $C_x^n - C_x^{n-1}$, or a fanin of some literal in C_x^n which has a fanin in $C_x^n - C_x^{n-1}$.*

Proof: y is an implicant of C_x^n only if it is set to some value under C_x^n by evaluate_node or by evaluate_fanin. In the former case, it must be a fanout of some node in C_x^n, since cofactoring by a cube only may set the values of fanouts of that cube. Further, if y is not a fanout of some literal in $C_x^n - C_x^{n-1}$ $y_{C_x^n} = y_{C_x^{n-1}}$, a contradiction since y is not an implicant of C_x^{n-1}. In the latter case, then there is some literal $z \in C_x^n$ s.t. $z_{C_x^n} = yf$, some f (or $\overline{z} \in C_x^n, z_{C_x^n} = \overline{y} + f$. Now, since y not an implicant of C_x^{n-1}, either z not in C_x^{n-1} in which case $z \in C_x^n - C_x^{n-1}$, or $z_{C_x^{n-1}} \neq z_{C_x^n}$, in which case at least one fanin of z in $C_x^n - C_x^{n-1}$. ∎

With this in hand, we can proceed with the evaluation algorithm. The previous lemma suggests an event-driven approach. We maintain a stack of records, each record containing a dataflow cube and a literal newly added to the cube. At each iteration, one such record is popped off the stack and the potential new implicants of the dataflow cube are examined. These are, by lemma, the fanouts of the new literal, the fanins of the new literal, and the other fanins of the fanouts of the new literal. This is captured by the following code:

```
while(($\mathcal{C}_x$, y) = pop_stack(evaluation_stack)) {
    if $\mathcal{C}_x = 0$ continue;
    foreach fanout w of y {
        if((w has a literal w1 $\in \mathcal{C}_x$) { (w1 is either $w$ or $\overline{w}$)
           new_cube = evaluate_fanin(w1, x);
           foreach literal z in new_cube
               merge_df_cubes(x, z);
               merge_df_cubes($\overline{z}$,$\overline{x}$);
        }
        phase = evaluate_node(w, x);
        if(phase != 2){
           w1 is the literal suggested by w and phase;
           merge_df_cubes(x, w1);
           merge_df_cubes($\overline{w1}$,$\overline{x}$);
        }
    }
    new_cube = evaluate_fanin(y, x);
    foreach literal z in new_cube
        merge_df_cubes(x, z);
        merge_df_cubes($\overline{z}$,$\overline{x}$);
    propagate_dataflow(x);
```

}

`merge_df_cubes` is modified to add new elements to this stack, as literals are added to the dataflow cubes. The stack is initially a set of records of the form `(Cx, x)` for each node `x`. The algorithm terminates when the stack is empty.

C.5 Correctness

We now turn to a proof of correctness of the algorithms given above. The correctness of `evaluate_node`, `evaluate_fanin`, `merge_df_cubes`, `propagate_dataflow`, and `propagate_zero` is evident, or has been adequately treated above. We now establish the correctness of the package.

Lemma C.5.1 *Let $x \in C_y$. Then either $C_y \supseteq C_x$ or C_x is on the stack.*

Proof: We construct a loop invariance argument. Clearly on the 0th iteration the statement holds. Suppose it holds through k iterations. Now suppose $x \in C_y$ on the $k+1$st iteration. If the statement of the lemma does not hold through this iteration, then either C_x was popped off the stack, or C_x grew and C_y did not grow to contain it, or C_x was added to C_y and not all of C_x was added to C_y. In the first case, at the end of the iteration `propagate_dataflow(x)` was called. Since $C_y \in D_x$, when `propagate_dataflow(x)` C_y is updated to contain C_x. In the second case, when C_x grows so that it no longer is contained C_y, C_x is shoved on the stack. In the third case, there is some C_z, $z \neq x$ such that $x \in C_z$, $C_x \not\subseteq C_z$ was added to C_y. But hence, on the kth iteration, we must have had C_x on the stack by the invariance assumption. ■

Corollary C.5.1 *At the completion of the algorithm, if $x \in C_y$, $C_y \supseteq C_x$.*

Theorem C.5.2 *Let C_x be the fixed points of relations (C.7)-(C.11). Then the final value of $C_x = C_x \; \forall x$*

Proof: $C_x \supseteq C_x$. We construct an inductive argument on n, and show that $C_x \supseteq C_x^n$ for every n. Since $C_x = C_x^n$ for some n, this gives the result. Clearly $C_x \supseteq C_x^0 (= \{x\})$.

Assume that $C_x \supseteq C_x^n \; \forall n \leq N$. Let $y \in C_x^{N+1}$. Now, either $y \in C_x^N$, and done, or y was added by one of relations (C.7)-(C.11). If by (C.7), then $y_{C_x^N} = 1$. Since $C_x \supseteq C_x^N$ by assumption, we must have that $y_{C_x} = 1$ for all iterations through the algorithm after the last element of $fanins(y) \cap C_x^N$ (call this z) was added to C_x. However, once z was added to C_x, then (z, C_x) was pushed on the evaluation stack by `merge_df_cubes`. When it was subsequently popped, all the fanouts of z were examined, including y. Since $y_{C_x} = 1$ for that value of C_x, C_y is merged with C_x. Since $y \in C_y$, we are done for this case. The cases of (C.8)-(C.9) are shown by similar arguments. For the case where y is added by (C.10), let t be the literal in the statement of the relation (C.10). By assumption $t \in C_x$ and $y \in C_t$. Now, either t is added to C_x after y is added to C_t, in which case y is added to C_x when C_t is merged into C_x in the main loop of the algorithm, or y is added to C_t after t is added to C_x, in which case $C_x \in D_t$ when y is added to C_t, and so y is added to C_x when C_t is merged into C_x in `propagate_dataflow`. For the case where y was added by (C.11), then on some iteration $\overline{x}$ was added to $C_{\overline{y}}$ (by induction, since $\overline{x} \in C_{\overline{y}}^N \subseteq C_{\overline{y}}$), and $C_{\overline{y}}$ was added to $D_{\overline{x}}$, and $(\overline{x}, C_{\overline{y}})$ was pushed on the stack. When it is popped, `propagate_dataflow(`$\overline{y}$`)` will merge $C_{\overline{x}}$ into $C_{\overline{y}}$, and then C_y into C_x, adding y to C_x, and done.

$C_x \subseteq C_x$. We show this by a loop invariance argument. Clearly $C_x \subseteq C_x$ through 0 iterations of the loop. Suppose that $C_x \subseteq C_x$ through k iterations of the loop; we must show that $C_x \subseteq C_x$ through the $k+1$st iteration. Suppose that y, C_x is the pair popped off the evaluation stack at the $k+1$st iteration. Now, if at the end of the $k+1$st iteration there is a z such that $C_z \not\subseteq C_z$, we have two cases. Case a, $z = x$. In this case, either `evaluate_node` or `evaluate_fanin` returned an incorrect result, or some cube C_y was merged into C_x and $C_y \not\subseteq C_y$. The correctness of `evaluate_node` and `evaluate_fanin` has been established, and $C_y \subseteq C_y$ by the invariance assumption, so the proof holds if $z = x$. If $z \neq x$, we have two cases. Either C_z was changed by the action of `propagate_dataflow`, and the item incorrectly added to C_z was in C_x But we must have then that $C_z \in D_x$, i.e., that $x \in C_z$. Since $C_x \subseteq C_x$ through the $k+1$st iteration from above, and since C_z is correct and contains x through the kth iteration, we must have that $C_x \subseteq C_z$. In the second case, $z = \overline{y}$, where y is the literal popped off the stack with C_x, and we know from (C.11) and (C.10) that $C_x \supseteq C_y \Rightarrow C_{\overline{y}} \supseteq C_{\overline{x}}$, and so done. [1] ∎

C.6 Complexity Analysis

There are two separate analyses: zero propagation and evaluation. Each literal can have its dataflow cube zeroed at most once, and so the cost of zeroing the cubes is $O(n)$, where n is the number of cubes in the network, plus the cost of traversing the transitive fanout of each node in the the dataflow graph, which is bounded above by the number of edges of this graph. This in turn is the total number of implications discovered, m, which is bounded above by $O(n^2)$ but should in the usual case be $O(n)$. Hence we argue that this is $O(n + m)$.

If we denote the maximum number of fanins of the nodes in the network as f_1, and the maximum number of fanouts as f_2, and the maximum number of cubes as c, and the maximum number of implicants in any dataflow cube (the maximum size of any $|C_x|$) as d_1, then we see that `evaluate_node(x, y)` and `evaluate_fanin(y, x)` are both $O(cf_1 + d_1)$. Clearly f_1, f_2, and d_1 are all $\leq n$, but in general this provides a very loose bound. Similarly, if we denote the maximum size of any D_x as d_2, this is bounded above by n but is in general small. Dataflow propagation is $O(d_1 d_2)$. Simple cube merging is $O(d_1)$. Since we have a new implication for each iteration through the loop, we have at most $O(m)$ iterations. There are $O(f_2)$ evaluations in each loop, and one propagation. Hence the total cost of finding m implications is bounded above by $O((cf_1 f_2 + f_2 d_1 + d_2 d_1)m)$. Note that each of these quantities should be in general small, and so we expect the average-case running time to be linear in n, though the worst-case running time, obtained in the case of a flat network (a network where every node is a primary input or a primary output), is $O(n^4)$. Of course, for such a network, one would hardly wish for dataflow analysis.

[1] A careful reader will note that this argument is not quite valid, since both C_x and C_z can change during the $k+1$st iteration in the backward evaluation loop. However, it is easy to extend this argument by arguing that one of the C variables must be the *first* to violate the containment condition; these break down to the two cases above, and are dispatched in the fashion shown; the invalid argument given above has been retained, since it is somewhat clearer than the similar, valid argument.

C.7 Efficiency

A loose lower bound for the problem is clearly $\Omega(m)$, which is considerably less than the $O((cf_1f_2 + f_2d_1 + d_2d_1)m)$ upper bound derived in the previous section. Both are loose to some degree, but that the Θ bound is likely to be closer to the upper than the lower bound. However, further research is required to derive a tighter lower bound for this problem.

It is easy to see that the algorithms derived here are more efficient than a naive implementation suggested by the recurrence relations:

```
while(changing)
    changing = 0;
    foreach node $x$
        $C'_x = C_x$;
        foreach $y \in C_x$
            $C'_x = C'_x \cup C_y$;
            foreach fanout $z$ of $y$
                if $z_{C_x} = 1$ $C'_x = C'_x \cup C_z$;
                if $z_{C_x} = 0$ $C'_x = C'_x \cup C_{\overline{z}}$;
            if $y_{C_x} = zf$ $C'_x = C'_x \cup C_z$;
        if($C'_x \neq C_x$) changing = 1;
    foreach node $x$
        $C_x = C'_x$;
```

There are potentially $O(m)$ iterations through the outer loop, since one can have as many as one iteration per implication. There are clearly $O(n)$ iterations of the second loop. There are $O(d_1)$ iterations of the third loop. The union operation is also $O(d_1)$. There are $O(f_2)$ fanouts of y, and each evaluation is $O(cf_1 + d_1)$. This implementation is clearly $O(mnd_1f_2(cf_1 + d_1))$, or $O(n^6)$ in the worst case.

C.8 Sparse Matrix Implementation

When these algorithms were implemented in MisII, it was found that the implementation was excessively slow. Given a CPU time limit of one hour on a Vax 8800, only the circuit C17 of the iscas benchmarks completed, in roughly 10 CPU minutes. Profiling revealed that the vast majority of time was spent in the low-level cofactor routines, which ideally should be $O(cf_1 + f_1 + d_1)$, but which may be $O(cf_1 d_1)$ in an implementation not designed with this application in mind. As a result, a sparse matrix implementation was done. In this implementation, an index is assigned to each literal. The row of the matrix corresponding to x represents C_x and the column corresponding to x represents D_x. The cofactoring process is simulated in a straightforward manner. Though the complexity is left unchanged by this implementation, the actual running time of the algorithms are substantially reduced.

C.9 An Improvement

A consistency equation may be considered. If $x \Rightarrow z$, $y \Rightarrow \overline{z}$, it must follow that $x \Rightarrow \overline{y}$ and $y \Rightarrow \overline{x}$, for if x and y, then z and $\overline{z}$, absurd. This gives the equation:

$$x \in C_{\overline{y}}^{n} \quad \text{if} \quad \exists z \in C_{x}^{n-1} \text{ and } \overline{z} \in C_{y}^{n-1} \tag{C.15}$$

This equation can actually be deduced from (C.10) and (C.11), and as a result is embedded in the procedure discussed so far. However, there is an interesting corollary. When $C_x = 0$, then $x = 1$ is impossible: x is stuck at 0. Hence *every* node may imply $x = 0$. We write:

$$x \in C_{y}^{n} \quad \text{if} \quad C_{\overline{x}}^{n-1} = 0 \tag{C.16}$$

This may be incorporated in the algorithm by modifying propagate_zero as follows:

```
propagate_zero(x)
{
    stack = tfo_collect(x);
    while((z = pop_stack(stack)) != nil) {
        $C_z = 0$;
        foreach literal y ≠ z
            $C_y = C_y \cup C_{\overline{z}}$
    }
}
```

Note this does not affect the complexity of the algorithm.

C.10 Results

The algorithms were implemented and tested on the well-known ISCAS benchmarks. The number of implications discovered is given, as well as the number of dataflow cubes that went to 0. This latter number is of very great interest, for the literals corresponding to these cubes cannot occur and hence the corresponding variable can be set to the opposite value in the circuit (for example, if $C_y = 0$, then every occurrence of y can be replaced by the constant 0; if $C_{\overline{y}} = 0$, then y can be replaced by the constant 1. If both C_y and $C_{\overline{y}} = 0$, the circuit is trivial. Run times are given in seconds on a Vax 8650.

Results and Times for the ISCAS Benchmarks

Circuit	Implicants	Zeroed Dataflow Cubes	Seconds
C1355	26400	0	445.1
C17	25	0	0.4
C1908	35951	0	354.9
C2670	47514	3	614.0
C3540			
C432	1826	0	18.8
C499	6040	0	56.4
C5315	70750	1	535.2
C6288	16070	17	158.9
C7552			
C880	5071	0	32.9

C.11 Extensions

C.11.1 Extending Arbitrary Cubes

The relations (C.7)-(C.11) may apply to any sets of asserted literals; the sets which begin with the initial assertion of a single literal is an interesting seed subcase. In general, however, we are interested in the behaviour of a circuit in response to a *set* C of asserted literals, not merely a single literal, and there is no reason to restrict C to satisfy the condition $C \subseteq C_x$ for some x. The problem is this: given some initial cube C^0 we wish to find some maximal cube of implications from the set of assertions contained in C^0.

Now, clearly one method is to take the union of the data flow sets implied by the cube:

$$C = \bigcup_{x \in C^0} C_x \tag{C.17}$$

with of course the caveat:

$$C = 0 \text{ if } \exists x \in C^0 \text{ such that } C_x = 0 \tag{C.18}$$

where C_x is, as before, the fixed point of relations (C.7)-(C.11). However, we can in general do somewhat better if $C^0 \nsubseteq C_x$ for some x. The union relation merely reflects the dataflow propagation of the C_x sets into the new cube. However, other implications may be derived from analogues to (C.7)-(C.9). From this we derive the equations:

$$y \in C^n \quad \text{if} \quad y_{C^{n-1}} = 1 \tag{C.19}$$

$$y \in C^n \quad \text{if} \quad \exists t \in C^{n-1} \text{ and } t_{C^{n-1}} = yf \text{ some } f \tag{C.20}$$

$$y \in C^n \quad \text{if} \quad \exists \overline{t} \in C^{n-1} \text{ and } t_{C^{n-1}} = \overline{y} + f \text{ some } f \tag{C.21}$$

$$y \in C^n \quad \text{if} \quad \exists t \in C^{n-1} \text{ and } y \in C_t \tag{C.22}$$

which are fairly clearly analogues of equations (C.7)-(C.10). An analogue to equation (C.11) is not required, since the contrapositive is only closed under the action of individual literals and hence is closed by the existing sets C_x.

Theorem C.11.1 *Consider any C^0. Let C be obtained as a fixed point of relations (C.19)-(C.22). Then if $y(\bar{y}) \in C$, y is set to $1(0)$ when the cube C^0 is set.*

Proof: Attach a node $\eta = C^0$ to the network, where η does not fan out. Observe that the relations (C.19)-(C.22) are identical to those obtained by relations (C.7)-(C.11) for such a new node, and apply the results of Lemma C.3.1. ■

The code to implement this is quite straightforward. One merely adds an extra row to the sparse matrix developed above, and implements a modified version of the main loop developed above; this modified version simply removes extraneous code relating to dataflow propagation of the cube C, whose dataflow implications clearly do not fan out.

```
while(($C$, y) = pop_stack(evaluation_stack)) {
    if $C_x = 0$ return 0;
    foreach fanout w of y {
        if((w has a literal w1 $\in C$) { (w1 is either $w$ or $\overline{w}$)
           new_cube = evaluate_fanin(w1, x);
           foreach literal z in new_cube
               merge $C_{\ddagger}$ into $C$;
        }
        phase = evaluate_node(w, x);
        if(phase != 2){
           w1 is the literal suggested by w and phase;
           merge $C_{w1}$ into $C$;
        }
    }
    new_cube = evaluate_fanin(y, $C$);
    foreach literal z in new_cube
        merge $C_{\ddagger}$ into $C$;
}
```

C.11.2 The Fanout Care Set and the Test Function

For some applications, (e.g., testing), one not only wishes the circumstances under which some literal may be set, but also the circumstances under which the setting of that variable may propagate to the output. This function, known variously as the *fanout care* condition or the *boolean difference*, may be written, for each variable x:

$$\sum_i f^i_x \oplus f^i_{\bar{x}} \tag{C.23}$$

and the implications which this required may be found by assigning the node η of the previous section to this value and performing the implications in the manner above.

Appendix D

Precharged, Unate Circuits

CMOS circuitry comes in two basic flavours: dynamic and static. Static circuitry, also called *restoring* logic, is fully complementary. Each gate consists of a *pulldown* network of NMOS transistors connecting the gate output to ground, and a *pullup* network of PMOS transistors connecting the output to power, or, as it is better known, V_{dd}. The pullup network is the dual of the pulldown network; hence exactly one of the two networks is conducting at any time, and so the output is affirmatively driven to its logic value. Hence a static gate is responsive to values on its inputs at all times; in logical terms, the gate may be thought of as an ideal logic switch.

Static CMOS gates obviously involve some redundancy at the transistor level, since either network is sufficient to compute the logic function. In the very early days of CMOS design (the late 1970's and early 1980's), a fairly dubious line of reasoning held that this wasted area, due to the large separation required between the p- and n-regions in CMOS technology[1].

The immediate idea was to eliminate one of the two transistor networks; the problem, therefore, was how to drive the logic to the appropriate value when the missing transistor network was putatively active. In 1982, Krambeck, Lee and Law [50] proposed to use the capacitive properties of the gate terminal of the MOS switch and of interconnect to realize its function; i.e., the fact that the wires and gate

[1]this separation is required to avoid an electrical problem known as *latchup*; this problem is not germane to the subject of this thesis–for details, see [81]

terminals of MOS transistors act as storage elements when disconnected from either power or ground.

Consider the case where the pullup network is eliminated. In this case, the gate cannot be driven to a logic 1 when required, though it can still be driven to a logic 0. Krambeck and his co-workers realized that in the case where a static gate would be driven to 1, the gate missing the pullup network would be disconnected from either power or ground and so would retain the value left on it. Hence if this value was 1, the gate would compute its correct value in this case. In the case where the pulldown network was active, the gate would be set to 0 as usual.

This design leads to the picture of a two-phase design. During the first or *precharging* phase, the pulldown network is disabled and the output of the gate is connected to power, so the storage element inherent in MOS logic is set to logic 1. During the second or *evaluation* phase, the network is enabled and the connection to power disabled.

The enabling and disabling of the relevant connections is fairly easy, as is shown in figure D.1. In this figure, a single PMOS transistor, controlled by φ, runs between power and the output. Similarly, a single NMOS transistor, also controlled by φ, runs between ground and the pulldown network.

Now, when φ is low, the PMOS transistor is conducting and the NMOS transistor is not; hence the output is connected to power and the pulldown network is disconnected from ground, i.e., is disabled – φ is low on the precharge phase. When φ is high, the PMOS transistor is not conducting and the output is isolated from power, and the pulldown network is connected to ground – i.e., the pulldown network is enabled.

Note that there is only a finite amount of charge stored on the output of any gate; hence, it is critical that the pulldown network only be conducting during the phase if its final state is conducting; otherwise, during the period it is conducting, the output can be driven to ground; this is a fatal error if the final value is 1, as it is if the pulldown network is nonconducting.

This has three immediate consequences. First, the initial state of the pulldown network must be non-conducting – all the transistors that may change during

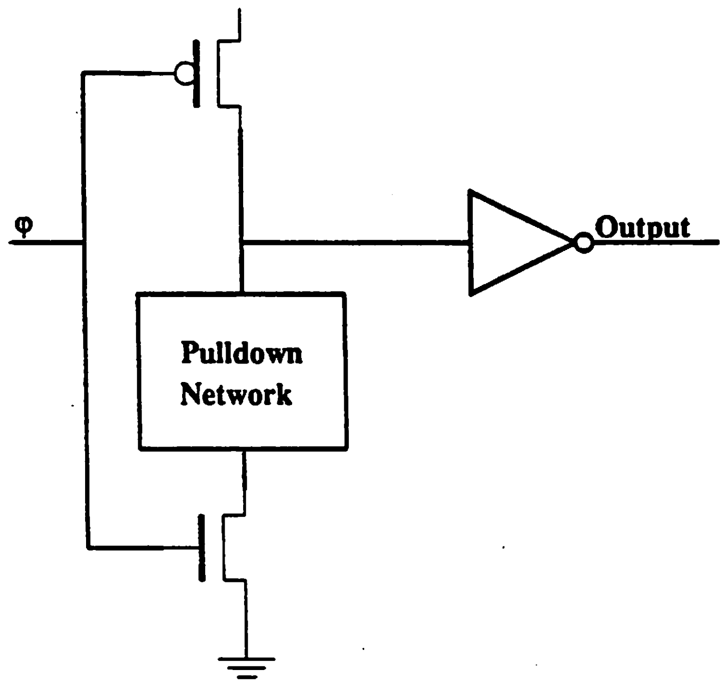


Figure D.1: Generic Dynamic Gate

evaluation must be initially off. Further, no transistor may turn on during evaluation unless its final value is on; i.e., the gate must be hazard-free to function correctly. Finally, since all transistors that may change must be precharged to their off state, and since the primary inputs may not be precharged, the evaluation phase may not begin until the primary inputs have all reached their stable value.

The hazard-free property is assured by construction. For the other, note that NMOS transistors are conducting for logic 1, nonconducting for logic 0. Hence the inputs to a gate must be precharged to 0. However, the precharge state of this gate is 1; if the inputs are gates like it, this will not do.

In Krambeck et. al.'s work, this is handled by attaching a static inverter to each gate; the gate itself fans out only to the static inverter, which is the real output of the gate. The gate is therefore precharged to 0 (since the precharge state of the inverter is obviously the inverse of that of the gate proper), and the gate can only undergo $0 \rightarrow 1$ transitions during evaluation. If this is done, this logic is called DOMINO. Now, since the basic MOS gate is inverting, the basic DOMINO gate is non-inverting; for the pulldown network will in general realize the complement of some (small) AND/OR network; the static inverter forces the DOMINO gate to realize the complement of the pulldown network, i.e. the AND/OR network. Note that adjustments need be made to the logic to realize an arbitrary function, since only the primary inputs of the circuit may appear in inverted form. However, it is easy to transform an arbitrary network into a network consisting of only AND and OR gates, where the primary inputs appear in both inverted and noninverted form.

We illustrate the basic properties of DOMINO logic with a simple example, shown in figure D.2. This gate realizes the function ab when φ is high.

It eventually developed that DOMINO logic saved little area, at least when laid out with an automated synthesis tool [38]. However the same work also showed that DOMINO logic switched about 30% faster than static CMOS.

The lack of inverting devices was a difficulty with DOMINO logic, since the technique used to transform an arbitrary network into a network of only AND and OR gates could potentially double the gate count, resulting in a circuit unacceptably large. Another approach, was devised by Goncalves and De Man and detailed in [31] and [27]. In their circuit style, called NORA, *only* inverting gates were permitted. NORA circuits consist of alternating *n*- and *p-type* gates. An *n-type* gate is simply a DOMINO gate *sans* the static inverter. A *p-type* gate is a gate consisting of a pullup rather than a pulldown network. The precharge transistor on a p-type gate is an NMOS transistor; the evaluate transistor is a PMOS transistor. Since the PMOS transistor is active when its control is at logic 0, if the precharge and evaluate transistors on the n-type gate are controlled by φ, the corresponding transistors on the p-type gate is controlled by $\overline{\varphi}$. Static inverters are classed as p-type gates if they are fed by an n-type gate, and n-type gates if they are fed by a p-type gate. Hence one may say that the output of a p-type gate is precharged to logic 0, and the output of an n-type gate is precharged to logic 1. Further, the same rule that input

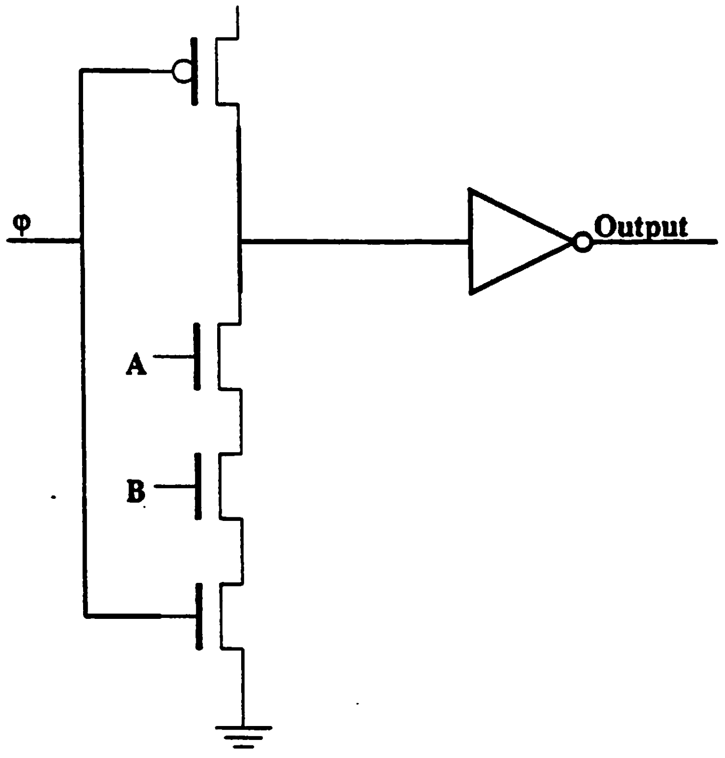


Figure D.2: DOMINO AND Gate

transistors must be precharged to their off state applies to NORA logic; the off-state for p-type gate inputs is logic 1, the off-state for n-type gate inputs is logic 0; this argument leads to the conclusion that n-type gate outputs can feed only p-type gates, and vice-versa.

Note that any NORA circuit can be transformed into an equivalent DOMINO circuit by adding inverters to the output of each gate and moving the pullup trees in p-type gates to the pulldown n-type well.

An example of a generic NORA network is shown in figure D.3.

A disadvantage of NORA circuits is that p-type transistors are typically slower to switch than n-type, due to decreased electron mobility in the p-well. Further,

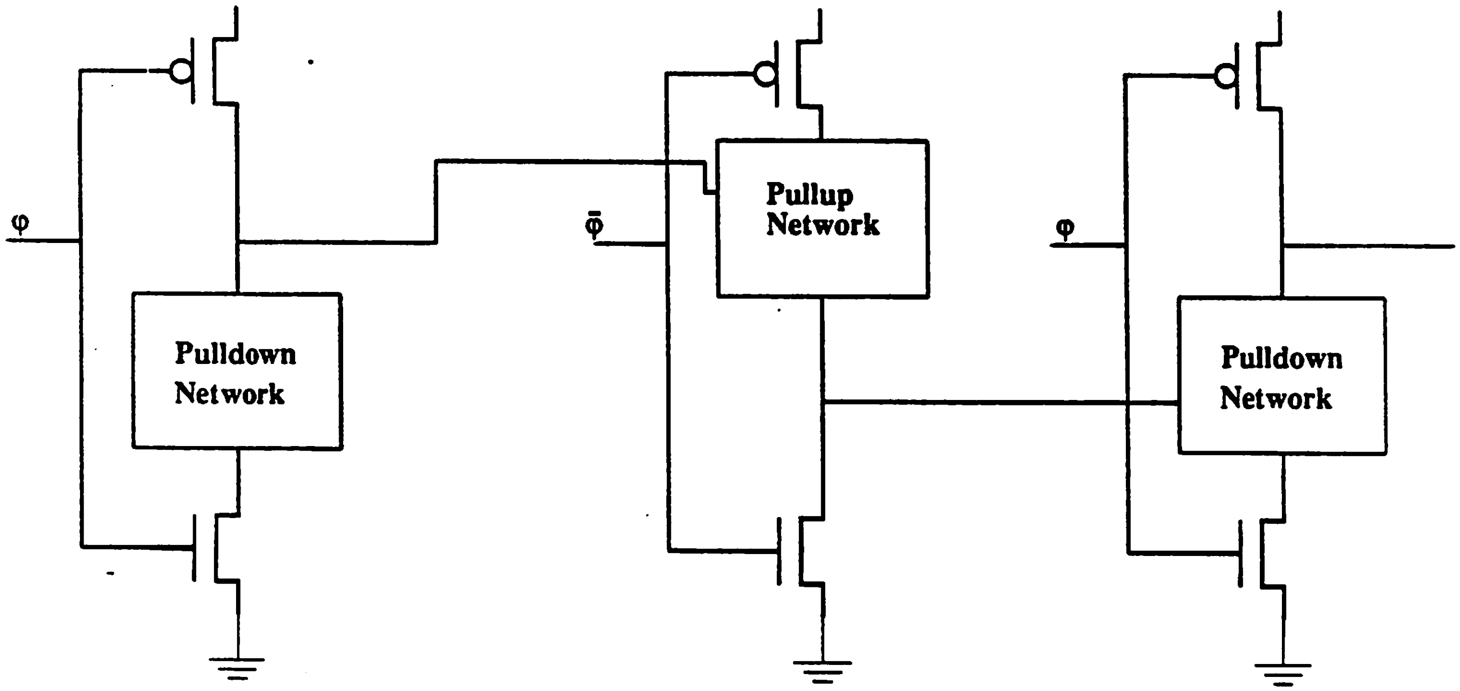


Figure D.3: Generic NORA Gate

the fact that p-type gates can only feed n-type gates and vice-versa makes NORA circuits very hard to design.

A final type of precharged circuit was introduced by Heller et al [36]. In this form of logic, called *Differential Cascode Voltage Switch* or, more simply, DCVS, the pulldown trees for both a function and its complement are implemented; a DCVS gate is simply two merged DOMINO gates back-to-back. This form of logic has the advantage that both complemented and uncomplemented logic is available, without NORA's slow p-type gates and difficulty. However, naive implementations of DCVS require as many transistors as full static; further, both each signal and its complement must be routed throughout the network in every DCVS implementation. These two factors indicate that the resulting circuit must be approximately twice the area of a static implementation; however, the flexibility of the DCVS design style and clever physical design have combined to offset this penalty[84].

In general, one can do somewhat better than naive implementations of DCVS. It was realized that DCVS circuits mapped nicely onto Bryant's graph-based representations of Boolean functions, called *Boolean Decision Diagrams* or *BDDs*. A BDD is a rooted, binary, directed acyclic graph with two leaves, where each non-leaf node is labelled with the name of a variable, and each edge is labelled either 1 or 0. The leaf nodes are labelled 1 and 0.

A BDD is intended to model the computation of a boolean function, given assignments of the variables, as a traversal of the graph. At each node, the edge labelled i is followed when the variable that labels that node is set to i. The value of the function is the label on the leaf at which the variable terminates.

Consider the edge labelled 1 originating in node labelled x to be a transistor that is active when $x = 1$; this clearly corresponds to an NMOS transistor controlled by x. Similarly, the edge labelled 0 corresponds to an NMOS transistor controlled by $\bar{x}$. In this case, there is a path from the node labelled 0 to the root iff f evaluates to 0, and a path from 1 to the root iff f evaluates to 1, i.e., $\bar{f}$ evaluates to 0. If the root is then directly connected to ground, it is clear that the function is correctly implemented (that is, there is a connecting path from ground to the appropriate node) if the node labelled 1 is replaced by a node labelled $\bar{f}$, and the node labelled 0 is replaced by a node labelled f. The derivation of this connection tree is shown in figure D.4. The full DCVS implementation, with the appropriate precharging logic, is shown in figure D.5. Note that, as with DOMINO implementations, the DCVS nodes must all be precharged low.

Note that each form of logic shown here has the property that the resulting gate can only change from 1 to 0 or from 0 to 1. Such gates are called *unate*. From the discussion above, it is clear that this is an inherent property of dynamic logic.

Interestingly, though the DCVS gate is unate is may realize any boolean function; e.g, the function $xz + \bar{x}yz$ is easily realizable by a DCVS gate. The reason that this is possible is that the literals x and $\bar{x}$ are carried on independent wires and hence are indistinguishable at the gate level from independent variables. The central

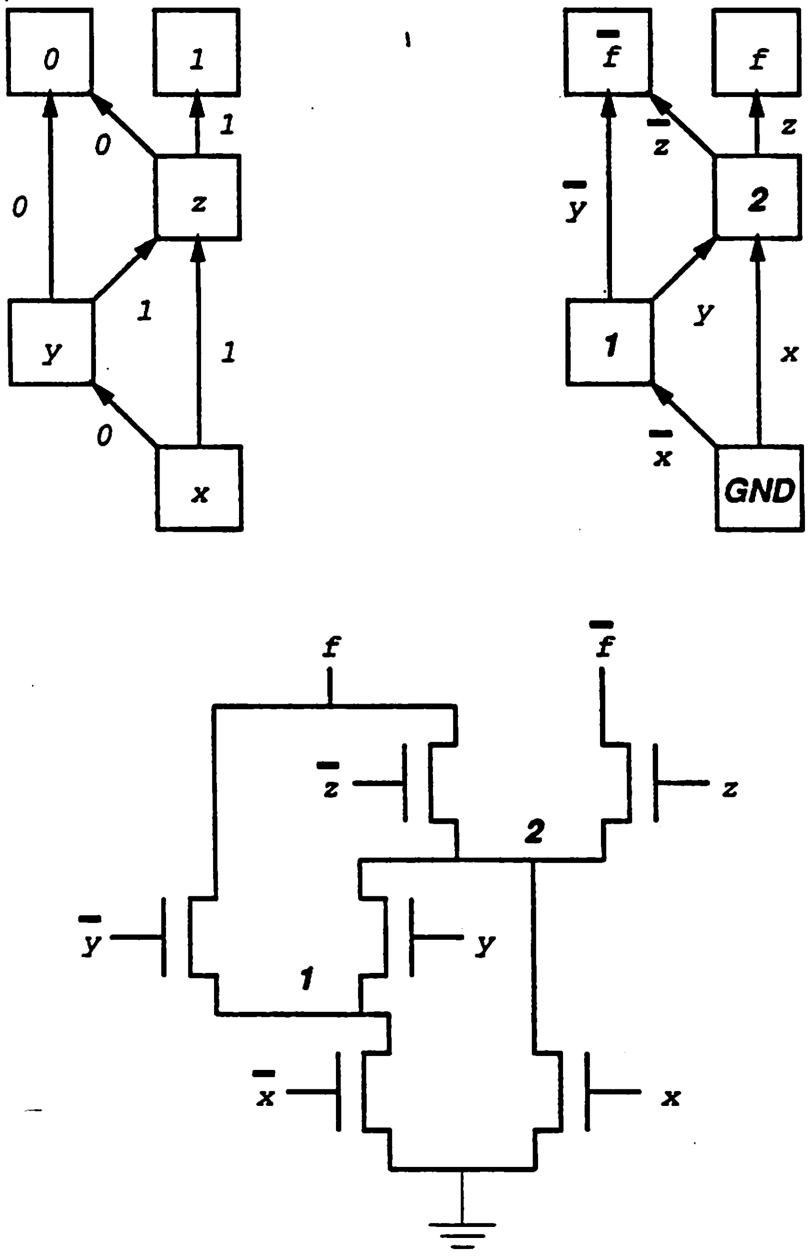


Figure D.4: BDD and DCVS Representation of $f = xz + \bar{x}yz$

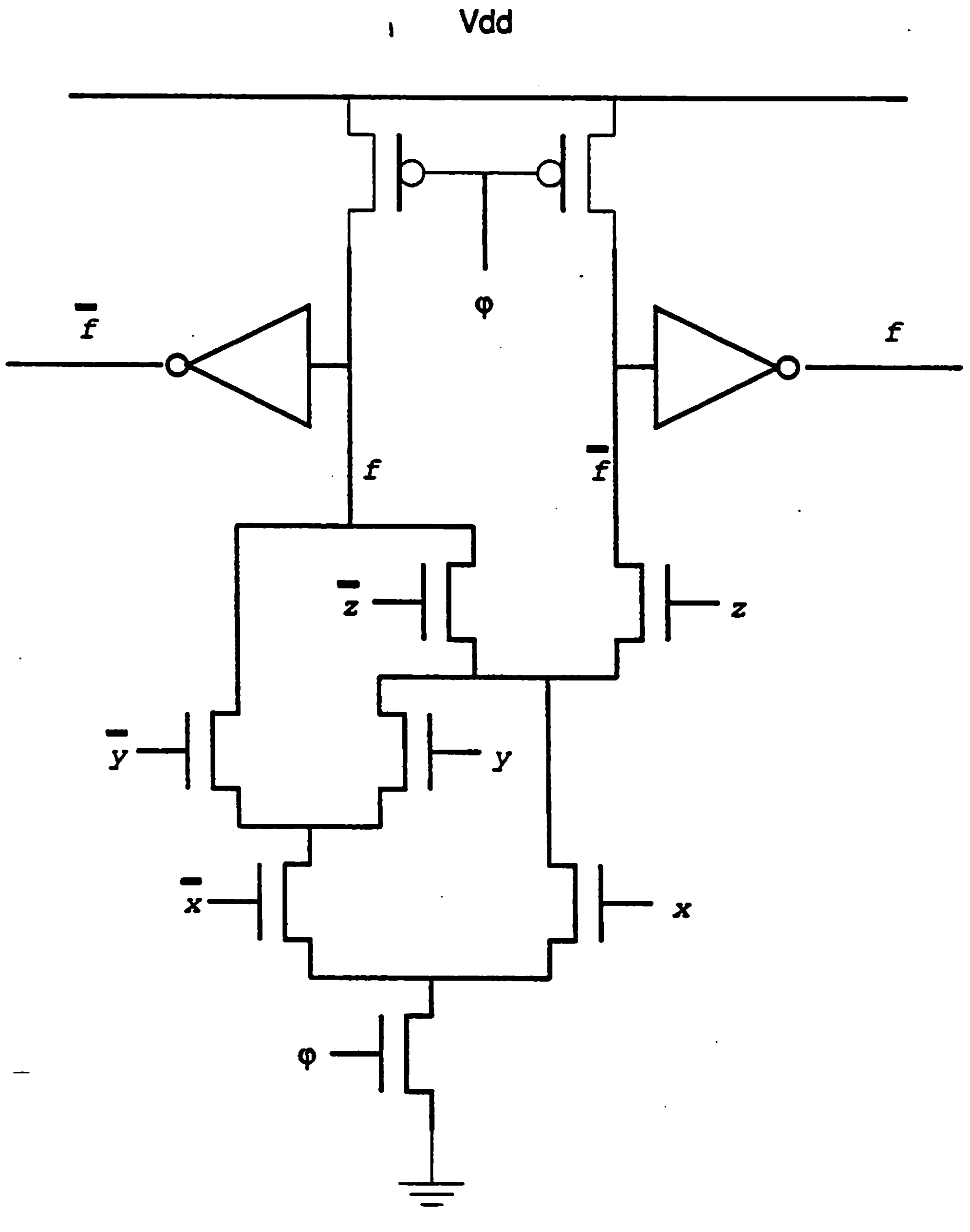


Figure D.5: Full DCVS Implementation $f = xz + \bar{x}yz$

attraction of DCVS technology is that it combines the strong timing properties of DOMINO demonstrated in this thesis and the full power of the static boolean gate.

Bibliography

[1] S. B. Akers. On a Theory of Boolean Functions. *J. SIAM*, 1959.

[2] K. A. Bartlett, D. G. Bostick, G. D. Hachtel, R. M. Jacoby, M. R. Lightner, P. H. Moceyunas, C. R. Morrison, and D. Ravenscroft. BOLD: A multi-level logic optimization system. In *IEEE International Conference on Computer-Aided Design*, 1987.

[3] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on CAD*, 1988.

[4] Romy L. Bauer, Jiayuan Fang, Antony P-C Ng, and Robert K. Brayton. XPSim: A MOS VLSI circuit simulator. In *IEEE International Conference on Computer-Aided Design*. 1988.

[5] Romy L. Bauer, Antony P-C Ng, Arvind Raghunathan, Mark W. Saake, and Clark D. Thompson. Simulating MOS VLSI circuits using SuperCrystal. In *VLSI '87*, 1987.

[6] J. Benkoski, E. Vanden Meesch, L. Claesen, and H. DeMan. Efficient algorithms for solving the false path problem in timing verification. In *IEEE International Conference on Computer-Aided Design*, 1987.

[7] L. Berman, L. Trevillyan, and W. Joyner. Global flow analysis in automated logic design. *IEEE Transactions on Computers*, January 1986.

[8] Daniel Brand. Redundancy and don't cares in logic synthesis. *IEEE Transactions on Computers*, October 1983.

[9] Daniel Brand. Personal communication, 1988.

[10] Daniel Brand and Vijay S. Iyengar. Timing analysis using functional analysis. Technical Report RC 11768, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 10598, 1986.

[11] Daniel Brand and Vijay S. Iyengar. Timing analysis using functional analysis. In *IEEE International Conference on Computer-Aided Design*, 1986.

[12] R. K. Brayton, E. Detjens, S. Krishna, T. Ma, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang R. Yung, and A. L. Sangiovanni-Vincentelli. Multiple level logic optimization system. In *IEEE International Conference on Computer-Aided Design*, 1986.

[13] R. K. Brayton, G. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[14] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. Wang. MIS: A multi-level logic synthesis system. *IEEE Transactions on CAD*, 1987.

[15] R. K. Brayton, F. Somenzi, and E. M. Sentovich. Don't cares and global flow analysis of boolean circuits. In *IEEE International Conference on Computer-Aided Design*, 1988.

[16] M. A. Breuer. The effects of races, delays, and delay faults on test generation. *IEEE Transactions on Computers*, 1974.

[17] M. A. Breuer and R. Lloyd Harrison. Procedures for eliminating static and dynamic hazards in test generation. *IEEE Transactions on Computers*, October 1974.

[18] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 1981.

[19] R. E. Bryant. MOSsim: A switch-level simulator for MOS LSI. In *Design Automation Conference*, 1981.

[20] James J. Cherry. PEARL: A CMOS timing analyzer. In *Design Automation Conference*, 1988.

[21] S. Cook. On the complexity of theorem-proving procedures. In *ACM Symposium on the Theory of Computing*, 1971.

[22] Ewald Detjens, Gary Gannot, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in mis. In *IEEE International Conference on Computer-Aided Design*, 1987.

[23] J. T. Deutsch and A. R. Newton. A multiprocessor implementation of relaxation-based electrical circuit simulation. In *Design Automation Conference*, 1984.

[24] David H. C. Du, Steve H. C. Yen, and S. Ghanta. On the general false path problem in timing analysis. In *Design Automation Conference*, 1989.

[25] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, March 1965.

[26] Jiayuan Fang. The approximate exponential function method for circuit simulation. Technical report, Electronics Research Laboratory, UC-Berkeley, 1987.

[27] V. Friedman and S. Liu. Dynamic logic CMOS circuits. *IEEE Journal of Solid State Circuits*, 1984.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[29] C. Thomas Glover and M. Ray Mercer. A method of delay fault test generation. In *Design Automation Conference*, 1988.

[30] Prabhakar Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, 1980.

[31] Nelson F. Goncalves and Hugo J. DeMan. NORA:a racefree dynamic CMOS technique for pipelined logic structures. *IEEE Journal of Solid State Circuits*, 1983.

[32] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[33] G. Hachtel, R. Jacoby, K. Keutzer, , and C. Morrison. On the relationship between area optimization and multifault testability of multilevel logic. In *International Workshop on Logic Synthesis*, 1989.

[34] G. Hachtel, R. Jacoby, and P. Moceyunas. On computing and approximating the observability don't-care set. In *International Workshop on Logic Synthesis*, 1989.

[35] G. Hachtel, R. Jacoby, P. Moceyunas, and C. Morrison. Performance enhancements in BOLD using "implications". In *IEEE International Conference on Computer-Aided Design*, 1988.

[36] L. G. Heller, W. R. Griffin, J. W. Davis, and N. G. Thoma. Cascode voltage switch logic: A differential CMOS logic family. In *IEEE International Solid State Circuits Conference*, 1984.

[37] Robert B. Hitchcock. Timing verification and the timing analysis program. In *Design Automation Conference*, 1982.

[38] Mark Hofmann. *Automated Synthesis of Multi-Level Logic in CMOS Technology*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1982.

[39] V. M. Hrapcenko. Depth and delay in a network. *Soviet Math. Dokl.*, 1978.

[40] N. Jouppi. TV: An nMOS timing analyzer. In *Third Caltech VLSI Conference*, 1983.

[41] N. Jouppi. Deriving signal flow direction in MOS VLSI. *IEEE Transactions on CAD*, july 1987.

[42] N. Jouppi. Timing analysis and performance improvement of MOS VLSI designs. *IEEE Transactions on CAD*, may 1987.

[43] R. M. Karp. Reducibility among combinatorial problems. In *ACM Symposium on the Theory of Computing*, 1971.

[44] Kurt Keutzer, Sharad Malik, and Alexander Saldanha. Is redundancy necessary to reduce delay? In *Submitted to Design Automation Conference*, 1990.

[45] Y. H. Kim. *Accurate Timing Verification for VLSI Designs.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1989.

[46] Y. H. Kim, S. H. Hwang, and A. R. Newton. Electrical-logic simulation and its application. *IEEE Transactions on CAD*, January 1989.

[47] T. W. Kirkpatrick and N. Clark. PERT as an aid to logic design. *IBM Journal of Research and Development*, 1966.

[48] T. W. Kirkpatrick and N. Clark. PERT as an aid to logic design. *IBM Journal of Research and Development*, 1966.

[49] D. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1973.

[50] R. H. Krambeck, C. M. Lee, and H-F. S. Law. High-speed compact circuits with CMOS. *IEEE Journal of Solid State Circuits*, 1982.

[51] Jean Davies Lesser and John J. Shedletsky. An experimental delay fault test generator for LSI logic. *IEEE Transactions on Computers*, 1980.

[52] Chin Jen Lin and Sudhakar M. Reddy. On delay fault testing in logic circuits. *IEEE Transactions on CAD*, 1987.

[76] L. Trevillyan and L. Berman. Improved logic optimization using global flow analysis. In *IEEE International Conference on Computer-Aided Design*, 1988.

[77] D. E. Wallace and C. H. Sequin. Plug-in timing models for an abstract timing verifier. In *Design Automation Conference*, 1986.

[78] D. E. Wallace and C. H. Sequin. ATV: An abstract timing verifier. In *Design Automation Conference*, 1988.

[79] D. M. Webber and A. L. Sangiovanni-Vincentelli. Circuit simulation on the connection machine. In *Design Automation Conference*, 1987.

[80] N. Weiner and A. L. Sangiovanni-Vincentelli. Timing analysis in a logic synthesis environment. In *Design Automation Conference*, 1989.

[81] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley, 1985.

[82] J. White and A. L. Sangiovanni-Vincentelli. *Relaxation-based Circuit Simulation.* Kluwer Academic Publishers, 1985.

[83] Steve H. C. Yen, David H. C. Du, and S. Ghanta. Efficient algorithms for extracting the *k* most critical paths in timing analysis. In *Design Automation Conference*, 1989.

[84] Ellen J. Yoffa and Peter S. Hauge. ACORN: A local customization approach to DCVS physical design. In *Design Automation Conference*, 1985.

[53] S. Malik, A. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *IEEE International Conference on Computer-Aided Design*, 1988.

[54] Hugo De Man. Personal communication, 1988.

[55] Patrick C. McGeer and Robert K. Brayton. Efficient algorithms for computing the longest viable path in a combinational network. In *Design Automation Conference*, 1989.

[56] Patrick C. McGeer and Robert K. Brayton. Provably correct critical paths. In *Decennial CalTech VLSI Conference*, 1989.

[57] Patrick C. McGeer and Robert K. Brayton. Hazard prevention in combinational circuits. In *Hawaii International Conference on the System Sciences*, 1990.

[58] Patrick C. McGeer and Robert K. Brayton. Timing analysis on precharged-unate networks. In *Submitted to Design Automation Conference*, 1990.

[59] Patrick C. McGeer, Robert K. Brayton, Richard L. Rudell, and Alberto L. Sangiovanni-Vincentelli. Extended stuck-fault testability for combinational networks. In *Submitted to MIT Conference on Advanced Research in VLSI*, 1990.

[60] Patrick C. McGeer, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Performance enhancement of combinational circuits through the introduction of τ-irredundant faults. In *Preparation*, 1990.

[61] T. M. McWilliams. Verifiction of timing constraints on large digital systems. In *Design Automation Conference*, 1980.

[62] L. W. Nagel. SPICE2: A computer program to simulate semiconductor circuits. Technical Report UCB/ERL M75/520, Electronics Research Lab, University of California at Berkeley, 1975.

[63] A. R. Newton and A. L. Sangiovanni-Vincentelli. Relaxation-based electrical simulation. *IEEE Transactions on Electronic Devices*, 1983.

[64] John K. Ousterhout. Crystal: A Timing Analyzer for nMOS VLSI Circuits. In *Third Caltech VLSI Conference*, 1983.

[65] John K. Ousterhout. Switch-level delay models for digital MOS VLSI. In *Design Automation Conference*, 1984.

[66] John K. Ousterhout. A switch-level timing verifier for digitial MOS VLSI. *IEEE Transactions on CAD*, July 1985.

[67] P. Penfield, Jr. and J. Rubinstein. Signal delay in RC tree networks. In *Design Automation Conference*, 1981.

[68] T. Quarles. *The* SPICE3 *Circuit Simulator*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1989.

[69] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM J. Res. Develop*, 1966.

[70] J. Rubinstein, P. Penfield, Jr., and M. A. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on CAD*, July 1983.

[71] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[72] Frederick F. Sellers, Jr., M. Y. Hsiao, and L. W. Bearnson. Analyzing errors with the Boolean difference. *IEEE Transactions on Computers*, 1968.

[73] C. E. Shannon. The synthesis of two-terminal switching function. *Bell System Technical Journal*, 1948.

[74] Kanwar Jit Singh, Albert R. Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *IEEE International Conference on Computer-Aided Design*, 1988.

[75] G. L. Smith. Model for delay faults based upon paths. In *International Test Conference*, 1985.