An Approach To Uncertainty In VLSI Design

Cheryl Lynn Harkness
Ph.D Dissertation

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-90-15
May 1991

AN APPROACH TO UNCERTAINTY IN VLSI DESIGN

by
Cheryl Lynn Harkness
B. S., Union College, 1984
Sc. M., Brown University, 1986

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1991

Vita

Cheryl Harkness was born in Troy, New York on August 3, 1962. She graduated summa cum laude and with departmental honors from Union College, Schenectady, New York in 1984 receiving a B.S. in Computer Science. Throughout her undergraduate career, Ms. Harkness spent her summers working first as a business applications programmer for an independent software firm and then as a summer intern with the VLSI Design Division at General Electric's Research and Development Center.

Ms. Harkness entered the graduate school at Brown University in September 1984, received her Sc.M. in Computer Science in May 1986, and was admitted to doctoral candidacy in May of the same year. While in graduate school, she has served as a research and teaching assistant for the Computer Science Department. In addition, she spent the summer of 1987 working on timing verification systems at Digital Equipment Corporation in Hudson, Massachusetts. With her advisor, she has published several technical reports on topics related to her thesis. She has co-authored a paper, entitled "Modeling Uncertainty in RC Timing Analysis", which was published in the 1989 *Proceedings of the IEEE International Conference on Computer-Aided Design* and was presented at the conference in November 1989.

Upon completion of her Ph.D. at Brown University, Ms. Harkness will join the Design Technology Center at Hewlett-Packard in Santa Clara, California. She has accepted a position as Software Development Engineer designing hierarchical simulation and verification tools. Her research interests include computer-aided design for VLSI, analysis of algorithms, computational geometry, and computer graphics. She is a member of Phi Beta Kappa, Sigma Xi, and the IEEE.

Abstract

Computer-aided design (CAD) tools take a circuit description and analyze the design to predict circuit behavior. Although these descriptions take many forms ranging from behavioral specifications to mask layouts, all of them contain bits of quantitative information about the circuit (e.g. physical dimensions, logic values, threshold voltages, parasitic capacitances). Often the precise values of these circuit characteristics cannot be determined before the layout is completed and/or the chip is fabricated. Even though the exact values are not known, estimates in the form of bounds, probability distributions, or average values can be obtained for these uncertain circuit parameters. Most current CAD tools employ only "expected" values in their calculations, returning one of several possible outcomes. A fundamental issue is how the circuit will behave with other combinations of possible values.

In this thesis, we present a general framework based on interval algebra for modeling uncertainty in VLSI. Our approach is to represent uncertain parameters as bounding intervals and to develop algebras for manipulating these ranges. We explore applications for this framework in several areas of VLSI design, including switch-level simulation, timing analysis, and placement. For each area, we create an application-specific interval algebra and incorporate this algebra within an existing algorithm for solving the problem. The result is a new interval version of the algorithm that uses the full range of values for each uncertain parameter in its computations and returns all possible solutions.

The different applications reveal the strengths and the weaknesses of the interval paradigm. Among its many advantages are its versatility, simplicity, and speed. Its main liability is that it may produce overly-conservative bounds for some applications. For these cases, we provide alternative interval-based techniques which improve the bounds, but require more computation time.

Acknowledgements

Although this document bears the name of a single author, it could not have been successfully completed without the help and assistance of many others. I would like to express my gratitude to all those who have contributed to this effort and made my stay here at Brown more enjoyable.

My advisor, Daniel Lopresti, deserves many thanks for his guidance and direction over the years. His suggestion that I look into interval arithmetic inspired this dissertation. His thoroughness as a reader and critic is also much appreciated. Through his meticulous efforts, this thesis was greatly improved.

I am indebted to my readers, John Savage and Gerard Baudet, for their many insightful comments. Special thanks to Gerard for sparking my interest in VLSI CAD, for advising me during my first years of graduate school, and for frequent visits back to Brown to check on my progress.

During my summer with Digital, Karem Sakallah introduced me to timing verification. His influence is evident in Chapters 3 through 5 of this thesis. I am also grateful to him for his advice and encouragement. My summer in industry was invaluable, not only for the experience gained, but also for the people that I met. Thanks to Darrylle, Sundar, Chandra, Karem, and Costas for a truly memorable summer.

Over the years at Brown, I have had the privilege of meeting many wonderful people. Chris Chiu, Phillip Hubbard, Richard Hughey, Richard Manning, Swaminathan Manohar, Alex and Susie Nacar, Scott Oaks, Robert Ravenscroft, Cathy Schevon, and John Stasko have all enriched my life with their friendship during these past six years. I am also grateful to Jim and Traudie Campbell for "adopting" me, for providing a haven away from academic life, and for making sure that I was always well-fed. Rob Ravenscroft warrants special thanks for his enduring friendship, for helping me through the tough times, and for sustaining me with some of the most delicious baked goodies

and culinary delights imaginable.

Ευχαριστῶ πάρα πολύ to Lucia Athanassaki for patiently teaching me Modern Greek. Learning Greek under her tutelage was a most pleasant and rewarding diversion. I only regret that I could not continue my lessons this last semester. Lucia, I wish you success with your thesis and career.

I am also thankful for the loving support of my significant other, Costas Spanos. I am indebted to him for his sound advice and inspiration, for his friendship and love, and for a constant supply of chocolate and teddy bears. Thanks also for two particularly unforgettable vacations. I don't know what I would have done without you.

The contribution of my family – brother, sister, parents, and grandparents – is immeasurable. Their support has meant more to me than they could ever imagine. Without my parents, Robert and Carol Harkness, none of this would have been possible. They have always stressed the importance of education and have encouraged their children to make the most of the opportunities before them. Throughout my long college and graduate career, they have sustained me with their unfailing love, encouragement, and financial assistance. This thesis is as much a product of their efforts as my own and I dedicate it to them.

CLH
Providence, RI

Contents

List of Tables

List of Figures

Chapter 1

Introduction

By analyzing a circuit description, VLSI design tools determine the features (i.e., functionality, speed, structure) of the resulting chip. Although circuit descriptions take many forms ranging from high-level behavioral specifications to mask layouts, all contain bits of quantitative information about the circuit (e.g., physical dimensions, logic values, threshold voltages, parasitic capacitances). As a result of postponed design decisions, manufacturing process disturbances, and many other factors, the precise values of these circuit parameters often cannot be determined in advance. For example, in top-down design, we begin with a purely functional specification of desired behavior and gradually refine this description to produce a structural implementation that realizes the given function. Throughout this process, we test portions of the design to analyze performance, to check functionality, or to explore alternatives. Because the design is incomplete at intermediate stages, all the implementation details needed for the analysis may not be known exactly. Even at the mask-level uncertainty exists, since the electrical properties of the circuit are subject to uncontrollable variations in the fabrication process. In these cases, a circuit parameter may assume any one of several possible values.

While the presence of uncertainty in VLSI design is widely recognized, most current computer-aided design (CAD) tools use only "expected" values in their calculations. They analyze one particular instantiation of the design, ignoring the other possibilities (See Figure 1.1). The most notable specific exceptions are found in timing verification, multiple-valued logics, and process simulation. Unfortunately, the only *general*

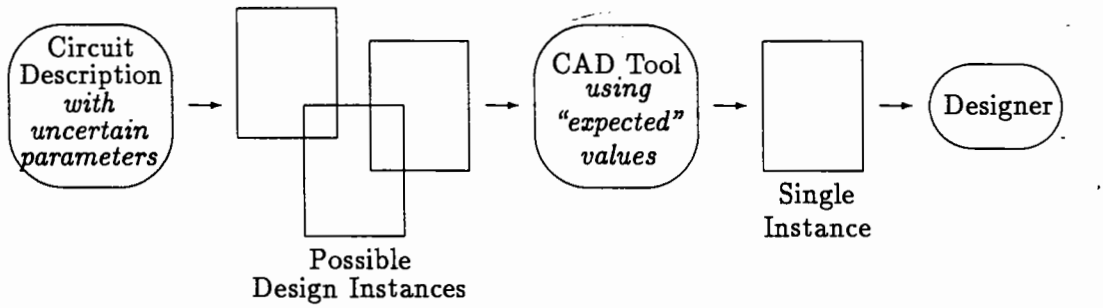


Figure 1.1: Current CAD Tools Ignore Uncertainty

method currently available for modeling parameter uncertainty is the computationally-expensive Monte Carlo technique. The goal of this research is to develop a general and efficient alternative to modeling uncertainty.

The major contribution of this thesis is a new theoretical framework based on interval algebra for modeling uncertainty in VLSI design. Our approach is to model uncertain parameters as intervals and to create interval algebras for manipulating these uncertain values. Once created, these algebras are embedded within existing CAD algorithms, thus creating new interval versions of the algorithms that compute bounds on the results when given interval inputs. We explore applications for this framework in several areas of VLSI design, including simulation, timing analysis, and placement. These applications not only illustrate the approach, but also reveal the strengths and weaknesses of the interval paradigm.

1.1 Uncertainty

Uncertainty arises when some physical aspect of the design (e.g., logic values, capacitances, resistances, physical dimensions, rise and fall times, wire length) needed to perform a phase of the analysis is not precisely known. *Webster's Dictionary* [Webs84, p. 744] gives the following three definitions for the word *uncertain*:

1. "Not yet determined or settled."

2. "Not capable of being known in advance."

3. "Likely to change: variable."

These definitions reveal three sources of uncertainty in VLSI design.

The first definition of an uncertain value is one that is "not yet determined or settled." In VLSI design, such uncertainty may result from delaying design decisions or exploring alternatives. In any circuit description, there are elements that can be implemented in more than one way and often there are many versions of a single part. One of the designer's problems is to choose the best implementation to realize his design. Figure 1.2 illustrates this principle showing two alternative implementations of an adder. Each different implementation will have its own unique set of characteristics (e.g., size, delay). If parts of the design are analyzed before all of these choices are made, we introduce some uncertainty about these characteristics. In top-down design, some implementation details may not be available when testing the circuit at an initial or intermediate stage. One example is estimating wire length or channel width while exploring placement alternatives. To evaluate placement quality, we may need to know wire length or channel width. Unfortunately, these values can only be precisely determined after full routing. Because routing is computationally expensive, it is not practical to perform this calculation for each possible placement. Instead, we use estimates of wire length to determine placement quality. Another example is preliminary behavioral simulation of a switch-level schematic before sizing all the transistors. To simulate this circuit, we need to know the relative strengths of the transistors, a function of transistor size. In this case, we must estimate transistor strengths.

Webster's second definition for uncertainty is "not capable of being known in advance." This type of uncertainty arises as a consequence of uncontrollable disturbances in the manufacturing process used to fabricate a chip. During fabrication, even slight aberrations in any of a multitude of process parameters (e.g., exposure time, temperature, gas pressure) produce variations in the electrical and physical characteristics of the resulting circuit (See Figure 1.3); thus, resistances and capacitances may vary from wafer to wafer and even from chip to chip on a single wafer. The circuit specifications given for each technology are statistical averages based on measurements taken from test runs of a technology. These values are thus only approximations to the real values. The only way to determine the actual values for each chip (and each device on the chip) is to measure them after fabrication, but in order to test designs and predict behavior before fabrication, we must estimate these transistor and interconnect characteristics.

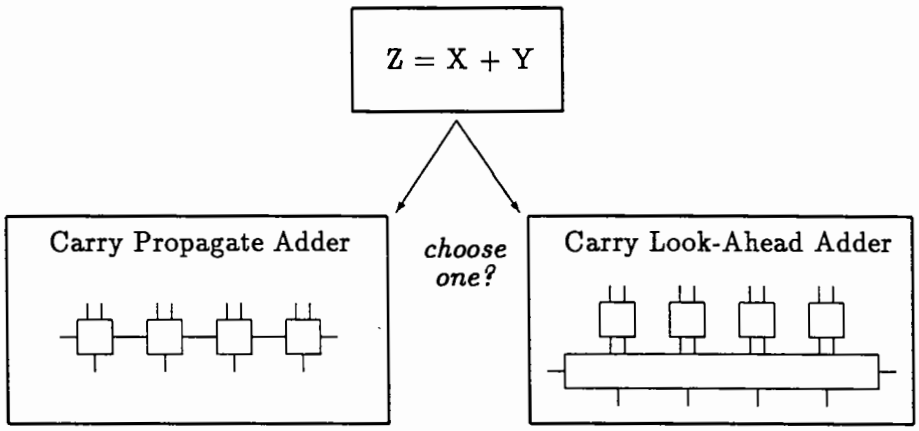


Figure 1.2: Choosing the Best Implementation

Figure 1.3: The Effects of Manufacturing Disturbances

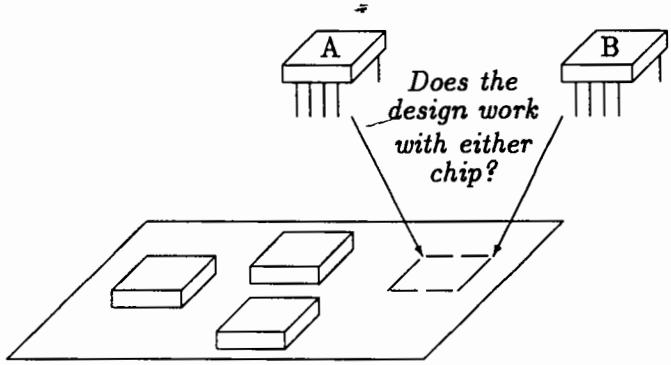


Figure 1.4: Designing for Flexibility

Electrical characteristics are not the only estimates that a designer commonly sees. In semi-custom design, predesigned standard cells are used as building blocks to create larger, more complex circuits. These cells too have estimated performance specifications. Even the specifications given for off-the-shelf parts are estimates. For example, the rise and fall times given for a chip are averages. The actual rise and fall times of each chip will lie within some acceptable tolerance of these given values.

The final definition of uncertain is "likely to change" or "variable." This type of uncertainty occurs when designing for flexibility. Under this scenario, the engineer designs a circuit with the express intention of being able to substitute functionally-equivalent pieces into the final design. As we stated earlier, each piece of a circuit description may be implemented in a variety of ways. But instead of choosing a single implementation for a portion of the circuit, the design engineer could build it so that several different implementations could be inserted into the design and it would still function correctly (See Figure 1.4). For example, consider designing a circuit board using off-the-shelf parts. Two vendors may manufacture chips with identical functionality, but perhaps slightly different operating speeds. For economic reasons, the designer may want to be able to use either chip in his/her final design. Because the implementations have slightly different operating characteristics, the values used in analysis will be uncertain.

These examples illustrate the prevalence of uncertain values in circuit design. As shown, uncertainty exists in all types of circuit descriptions from system-level specifications to technology-dependent layouts. In spite of this, most current design aids ignore

uncertainty and employ only single-valued estimates in their calculations. These programs analyze only one particular set of values and return a single estimate of behavior, but we require methods that test the full range of possible values for each parameter.

1.2 Motivation

What advantages does modeling uncertainty offer? Methods that account for parameter uncertainty are necessary for several reasons. They return more credible results, provide a more complete characterization of circuit behavior, and facilitate iterative improvement.

The first of these benefits is credibility. Using expected values in circuit analysis yields only one of many possible solutions. Unless the true values of all parameters match the average values chosen, the observed behavior of the circuit will differ from the predicted performance. In practice, the probability of exactly predicting the true result is very small. In contrast, approaches that model uncertainty return a range of potential behaviors for the circuit, instead of an exact solution. With these methods, it is more certain that the measured result will fall within the predicted range.

A more complete characterization of the circuit is the second advantage. Using the average values alone may yield an expected solution, but gives no indication of the range of possible solutions that can result. As stated in a recent article:

> Statistical analysis recognizes that components used in a design normally have a range of acceptable values, typically varying between 1 and 20 percent from the stated value. When a breadboard is built and tested, one possible combination of values is observed. Similarly, initial simulations use the exact values shown on the schematic. But the question is whether the design will continue to work satisfactorily with other possible combinations. [Hard88, p.39]

Analysis methods that incorporate uncertainty generate the full range of possible outputs, not just the "expected" result. Depending on the method employed, the range of solutions may be represented as bounds or as a statistical distribution. An added benefit of uncertain approaches is that they generate best- and worst-case bounds automatically.

The third advantage of such methods is that they facilitate efficient iterative design.

Chip design is an iterative process of creation, testing, and modification. Mistakes in the design can require costly re-evaluation. In describing the testing of the MIPS-X chip, Chow and Horowitz state:

> A major problem with the tool set was the amount of time it took to find a bug, fix it and rerun the simulation. During the end of the design, it would take over three CPU hours on a DEC Microvax II to generate and initialize a new simulation run after the layout was changed. [Chow88, pp. 104-5]

Armed with CAD tools that handle uncertain values, the design engineer can test portions of the circuit earlier and more completely. This allows him/her to locate and eliminate some errors much earlier in the design process, when changes are less costly.

Uncertain approaches not only allow the engineer to test the circuit earlier, but can also increase efficiency in other ways. With tools that support only single-valued estimates, the designer must repeat the analysis every time these values change. Bounding results, however, may be reusable. As long as the updated values or ranges still fall within the original ranges given for a variable, the results of previous computations are still valid. This is particularly useful when the source of uncertainty is delayed design decisions. In these cases, the bounds are typically refined as choices are made. Here we can benefit from the results of previous computations and may only have to re-evaluate a small subset of the alternatives, instead of repeating the entire analysis.

1.3 Background

While most current design aids ignore parameter variability, there are several noteworthy exceptions. Foremost among these is Hayes' contribution to multiple-valued logics. In [Haye86], he defines uncertainty as it pertains to discrete logic levels in digital circuit design and introduces a formalism for creating multiple-valued logics. He begins with a set S of n discrete logic levels and defines an uncertain variable (called a *u-value*) as the subset of S containing all possible logic levels that the unknown could assume. For example, if $S = \{0, 1, 2\}$, then $U = \{0, 1\}$ is a u-value over S. Given any two-input operator over the set S, this operator can be extended to handle u-values by applying it to each element of the u-value and taking the union of the results. For example, let S be the binary set $\{0, 1\}$ and consider the boolean operator AND. If X represents the u-value $\{0, 1\}$, then $AND(1, X) = AND(\{1\}, \{0, 1\}) = AND(1, 0) \cup AND(1, 1) =$

$\{0,1\} = X$. Hayes also presents extensions for creating and manipulating strings of u-values, called *p-values*. U-value operators can be further extended to handle p-values by applying the operator pairwise to corresponding vector elements and concatenating the results. For example, $AND(1X,01) = AND(1,0)\ AND(X,1) = 0X$. Using these basic principles, Hayes creates increasingly complex logics from the basic binary set $\{0,1\}$ and gives several examples of logics in common use today.

Hayes' treatment of uncertainty is complete and rigorous as it applies to multiple-valued logics. While his methods are well-suited to logic values, they are not practical as a general method for modeling uncertainty. One reason is that Hayes' operators are computed by the exhaustive enumeration of all possible combinations of operand values. Because digital logics typically consist of small, finite sets of discrete values, exhaustive enumeration is possible. For other applications, the uncertain parameters may be defined over an infinite set (such as $\Re$) and thus may have an infinite number of possible values. In such cases, Hayes' methods cannot be applied. Even so, some of his basic ideas still hold and we incorporate them in our solution.

A second application involving unknown logic values is presented by Bryant in [Brya84]. He describes a switch-level simulator, MOSSIM II, designed to handle unknown transistor state. A transistor may be in any one of three possible states: 0 (open or nonconducting), 1 (closed or conducting), or X (indeterminate). Bryant seeks an efficient method for handling indeterminate transistor state without having to evaluate all 2^k possible assignments of values to the k indeterminate-state transistors. To solve this problem, he presents a *signal flow analysis* algorithm and provides a proof of his method. Other switch-level simulators use similar methods to address the problem of unknown transistor state, including [Byrd85, Gecs87, Hajj87, Rama83a, Rama83b, Sund87]. In this thesis, we introduce another method for modeling uncertainty and apply this technique to create an extension of Bryant's algorithm that models uncertain transistor strength and node size as well.

Systems that model uncertainty also exist for timing verification of digital circuits. These systems may be divided into two classes: bounding and statistical approaches. McWilliams' timing verifier SCALD[McWi80] provides an example of the bounding approach. SCALD is a timing verifier for synchronous, sequential, digital circuits that uses the minimum and maximum propagation delays of circuit components (e.g., logic gates, flip flops) and wires to compute bounds on circuit delay.

Representing the statistical approach, Hitchcock presents a statistically-based timing verifier, TA[Hitc82]. In TA, the delay of each circuit component is represented by a mean value and a standard deviation. The delay of the circuit is represented as a statistical distribution calculated from the delays of the components. Mean arrival time, μ, is computed by summing the mean delays along the path taken and the standard deviation, σ, is found by applying standard convolutions. The longest path is defined as the path with the maximum $\mu + \beta\sigma$, where β is a constant specified by the user representing the confidence level. This model assumes that all distributions are Gaussian.

In [Wall86], Wallace and Sequin propose an abstract timing verifier which provides a standard user interface, but supports several different timing models. Among the delay models supported are a bounding model similar to SCALD and a statistical model patterned after TA. The framework presented by Wallace and Sequin is elegant and the delay models are more precisely defined.

In both bounding methods, the operators for computing bounds are often only intuitively defined. One of our objectives is to identify the underlying algebraic structure needed to compute bounds and to formally define these operators. In addition to computing delay, we also need to return the longest paths themselves. We discuss each of these issues in greater detail in this thesis.

Recent developments in statistical process simulation have focused attention on the need for CAD tools that model uncertainty. In describing their process simulator, Fabrics II [Nass84], Nassif and his colleagues state the following:

> For performance evaluation prior to IC manufacturing, designers usually employ a circuit simulator, such as SPICE. However, the accuracy of such a performance evaluation strongly depends on the accuracy of the device model parameters used in the simulation. In order to guarantee accuracy of circuit simulation, model parameters are extracted from measurements made on fabricated devices ... An alternative method for determining device parameters is to use a sequence of process and device simulators, such as Fabrics II, which contain physical models of fabrication steps and semiconductor devices. [Nass84, p. 41]

Fabrics II is a process simulator that produces statistical values for integrated circuit (IC) parameters. This system consists of two parts: a process simulator and a device

simulator. Given a description of the manufacturing steps including the process parameters (e.g., times and temperatures of the diffusions steps, doses and energies of the ion implantations) and disturbances (i.e., random variables with probability distributions which model disturbances such as mask misalignment and uneven absorption of dopant), the simulator computes a set of physical parameters for the process (e.g., impurity concentrations, sheet resistances, gradient voltages, and oxide thicknesses). From the physical parameters generated by the process simulator and the layout of a device, the device simulator produces statistical estimates of device parameters. These parameters include threshold voltage, intrinsic transconductance, the dimensions of the device, and intranodal and substrate zero-bias capacitances. When coupled with a circuit simulator, this statistical information may be used to sample the performance of an integrated circuit.

Others employ these device parameter estimates to perform circuit simulation. The first method is a novel bounding approach presented by Zukowski. In [Zuko86], he states:

> A circuit designer is often concerned with the performance of a circuit over a fairly wide range of inputs and circuit models. The uncertainties in the simulation, often arising from variations in fabrication processes, produce a range of circuit behaviors that can be captured in a bound. Since complete bounding simulators have not yet been available to circuit designers, they often use a technique called approximate worst case analysis to estimate the range of behaviors. In approximate worst case analysis, an exact simulator is used at least twice, with inputs that are at the extremes thought most likely to produce extreme behavior. One use of a bounding simulator is to replace this approximate procedure with a rigorous one. [Zuko86, p. 53]

To accomplish this goal, his algorithms use simplified circuit models whose behavior bounds that of the more accurate models. His basic strategy consists of three parts: devising efficient methods to compute bounds on the behavior of very simple circuits, transforming the more realistic models into these simple ones by exploiting the monotonic properties of the model, and using relaxation techniques to analyze the subcircuits. The basic elements of his simplified model are a lumped resistor, a purely resistive transistor, and a lumped capacitor. These are combined to create a *bounding*

circuit whose behavior bounds that of the original. The key characteristic of a bounding circuit is that its behavior is a monotonic function of input voltage and component behavior. To compute bounds on the behavior of the circuit, Zukowski's algorithm uses relaxation techniques to analyze the network equations that define behavior. It performs the relaxation twice – once to compute upper bounds and again to compute lower bounds. The monotonicity of the variables determines the values used in each computation.

Instead of creating simplified circuit models, our approach is to define algebras for manipulating bounds. The two methods are complementary and it may be possible to incorporate both within a single system. For example, with a bounding algebra, the relaxation phase would only be performed once to compute both upper and lower bounds.

The second electrical simulation method is more representative of current practice. In [Dive84], Divekar addresses the problem of statistical circuit simulation to determine circuit performance. He advocates using Monte Carlo simulation to model the effects of device parameter uncertainty. The Monte Carlo method [Sobo75] is a commonly used, statistically-based technique for handling uncertainty. Using this procedure, the circuit is simulated many times. Each time, the algorithm randomly selects a value for each variable from its distribution. The results of each trial are recorded and compiled to compute bounds or a statistical distribution of the results. While this method is effective, it is computationally-expensive. For compute-intensive applications, such as simulation, the time complexity of the basic algorithm prohibits large numbers of Monte Carlo trials; however, many trials are necessary to accurately determine circuit behavior. The goal of this thesis is to develop methods for modeling uncertainty which produce similar bounds in less time.

1.4 Bounding vs. Statistical Methods

These current approaches to modeling uncertainty can be divided into two general categories: bounding methods and statistical methods. In the bounding methods, parameter values are chosen at the minimum and maximum endpoints of their ranges to compute worst and best-case circuit behavior. In addition to worst-case bounds, statistical techniques return a probability distribution of the results, represented as a mean value and standard deviation. There are advantages and disadvantages to each

of these methods.

Proponents of statistical methods argue that their approach is more accurate. Bounding models return performance limits without any indication of the probability of their occurrence. If the probability of these worst-case conditions is small, then these results are overly conservative.

While this is certainly true, bounding techniques offer several advantages over statistical ones. First, they are simpler and more intuitive. Operations for manipulating bounds are easier to derive than those for statistical distributions. An arithmetic for combining ranges, called *interval arithmetic* [Kuli81, Moor79], has been developed. In this thesis, we employ the principles of this arithmetic to derive other operations needed in VLSI applications. For statistical distributions, no such algebras exist. Hitchcock does provide a rule for adding two Gaussian distributions; however, more complex operations are needed for other VLSI applications and we cannot assume that the distributions will always be normal. Currently, the only general methods available for statistical analysis are trial methods, such as Monte Carlo simulation.

The second advantage is speed. Bounding algorithms are inherently much faster than statistical methods. As demonstrated in this thesis, bounding methods are several orders of magnitude faster than Monte Carlo simulation.

Third, bounding methods are more general. Value ranges can be used to model types of uncertainty that are not suited to statistical measures. For instance, a mean value and standard deviation have little meaning for multiple-valued logics. Also, uncertainty that arises as a result of exploring design alternatives cannot be modeled as a statistical distribution.

1.5 Our Solution

For the above reasons, we have chosen a bounding approach based on interval analysis to model uncertainty. The principles of our method are derived from a field of mathematics known as *interval arithmetic* [Kuli81, Moor79], introduced in the 1960's to bound truncation error in digital computers. Building on the basic algebraic structures of this arithmetic, we create a formal framework for modeling uncertainty in VLSI design. By deriving application-specific interval algebras for manipulating uncertain values and incorporating them within existing algorithms for analyzing VLSI circuits, we create new interval versions of the algorithms that compute bounds on the results

when given interval inputs.

The first step of this methodology is to represent uncertain values as intervals. We begin with a few definitions. An interval $[a, b]$ over an ordered set of values S is defined as the set $\{x \mid x \in S, a \leq x \leq b\}$. A *degenerate* interval $[a, a]$ is an interval which contains only the one element a. Let I be the set of all intervals over the real numbers $\Re$; thus, $I \equiv \{[a, b] \mid a, b \in \Re\}$. In this thesis, we use two notations to denote intervals: we either represent an interval by its endpoints in square brackets (i.e., $[a, b]$) or as a variable with a hat (i.e., $\hat{x}$).

The second step is to define algebras for manipulating the intervals. Since each application needs different combinations of operators, these algebras are custom-tailored for each application. For example, we need interval arithmetic (i.e., addition, subtraction, etc.) to calculate the delay of a transistor network, but only minimum, maximum, and blocking operations to perform switch-level simulation. While the algebras are application-specific, the method for deriving their interval operators is the same for all applications. Each interval operator F is the logical extension of a real-valued operator f and is derived from it using the following formula [Moor79]:

$$F(\hat{x}, \hat{y}) \equiv \{f(x, y) \mid x \in \hat{x} \text{ and } y \in \hat{y}\}. \tag{1.1}$$

In other words, an interval operator is derived from its real counterpart by applying the non-interval operator to all possible combinations of the elements chosen from each interval and taking the union of the results. Using this technique, a rule is obtained for computing bounds on the result. For example, interval addition $\oplus$ is defined as follows: $[a, b] \oplus [c, d] = \{x + y \mid x \in [a, b] \text{ and } y \in [c, d]\} = [a + c, b + d]$.

Using this formula, the following interval arithmetic operations $\oplus$ (addition), $\ominus$ (subtraction), $\odot$ (multiplication), and $\oslash$ (division) over the set I are defined:

$$[a, b] \oplus [c, d] \equiv [a + c, b + d] \tag{1.2}$$

$$[a, b] \ominus [c, d] \equiv [a - d, b - c] \tag{1.3}$$

$$[a, b] \odot [c, d] \equiv [min(ac, ad, bc, bd), max(ac, ad, bc, bd)] \tag{1.4}$$

$$[a, b] \oslash [c, d] \equiv [a, b] \odot [1/c, 1/d] \quad \text{provided that } 0 \notin [c, d] \tag{1.5}$$

Similarly, we derive the following minimum (MIN) and maximum (MAX) operators:

$$MIN([a, b], [c, d]) \equiv [min(a, c), min(b, d)] \tag{1.6}$$

$$MAX([a, b], [c, d]) \equiv [max(a, c), max(b, d)] \tag{1.7}$$

In addition, there are several order relations which are useful:

$$[a,b] = [c,d] \iff a = c \text{ and } b = d \tag{1.8}$$

$$[a,b] < [c,d] \iff b < c \tag{1.9}$$

$$[a,b] > [c,d] \iff a > d \tag{1.10}$$

$$[a,b] \leq [c,d] \iff a \leq c \text{ and } b \leq d \tag{1.11}$$

$$[a,b] \geq [c,d] \iff a \geq c \text{ and } b \geq d \tag{1.12}$$

$$[a,b] \parallel [c,d] \iff [a,b] \subset [c,d] \text{ or } [c,d] \subset [a,b] \tag{1.13}$$

The last of these relations states that two intervals are incomparable ($\parallel$), if one is a proper subset of the other.

Over the set of real intervals I, these operations have a number of important properties. Interval addition and multiplication are both commutative and associative. The degenerate intervals $[0,0]$ and $[1,1]$ form the additive and multiplicative identities respectively. The distributive property

$$\hat{x} \odot (\hat{y} \oplus \hat{z}) = (\hat{x} \odot \hat{y}) \oplus (\hat{x} \odot \hat{z}) \text{ for all } \hat{x}, \hat{y}, \hat{z} \in I$$

holds only in the following special cases:

1. if $\hat{x}$ is a degenerate interval or

2. if $\hat{y} \odot \hat{z} > [0,0]$.

However, the subdistributivity property always holds; therefore,

$$\hat{x} \odot (\hat{y} \oplus \hat{z}) \subseteq (\hat{x} \odot \hat{y}) \oplus (\hat{x} \odot \hat{z}).$$

Non-degenerate intervals have no additive or multiplicative inverses; in particular, $\hat{x} \oslash \hat{x} \neq [1,1]$ unless $\hat{x}$ is degenerate. Another important property of interval arithmetic is *monotonic inclusion*: if $\hat{w}, \hat{x}, \hat{y}, \hat{z} \in I$ and $* \in \{\oplus, \ominus, \odot, \oslash\}$

$$\hat{w} \subseteq \hat{y} \text{ and } \hat{x} \subseteq \hat{z} \Longrightarrow \hat{w} * \hat{x} \subseteq \hat{y} * \hat{z}.$$

For the minimum and maximum operators, we must augment the set of real numbers $\Re$ to include ∞ and $-\infty$. Let I^+ denote the set of intervals over this augmented set. The interval minimum and maximum operators are both commutative and associative. $[\infty, \infty]$ and $[-\infty, -\infty]$ are the identity elements of MIN and MAX respectively. I^+

is closed with respect to either operator. Proofs of these properties are provided in Appendix A.

The final step of the method is to embed these interval operators within existing algorithms for analyzing VLSI circuits. This creates an interval version of the algorithm that computes bounds on the result of the computation when given interval input parameters.

1.6 Applications

In the remainder of this thesis, we present several examples chosen from various areas of VLSI analysis to illustrate the usefulness and elegance of the interval approach to uncertainty. These examples run the gamut from simulation to placement and are intended to demonstrate that interval analysis can be successfully incorporated into many different types of VLSI algorithms. Chapter 2 contains the first application, switch-level simulation. Here, we employ interval methods to simulate transistor networks with unknown transistor strengths and node sizes. The second application is timing analysis. This problem has two parts. In Chapters 3 and 4, we perform RC timing analysis to compute bounds on the delay of a transistor network given uncertain device characteristics. In Chapter 5, we combine these delays using critical path analysis to determine bounds on the delay of the circuit. The third application is placement. In Chapter 6, we compute an interval objective function and discuss how uncertain cost functions may be used when seeking minimum-cost placements. These three applications not only illustrate our interval methodology, but also reveal its strengths and weaknesses. In Chapter 7, we conclude by discussing the merits and limitations of the interval paradigm and suggesting other areas for which these same interval techniques may be applied.

Chapter 2

Switch-Level Simulation with Uncertain Signal Strength

The first application of our interval technique is switch-level simulation. Numerous techniques for performing rapid switch-level simulation have been developed in recent years [Brya80, Brya84, Byrd85, Gecs87, Rama83a, Sund87]. Without exception, these methods require detailed knowledge of circuit implementation. In particular, the relative conductances, or strengths, of all transistors must be specified in advance. For example, Figure 2.1 depicts a simple transistor network consisting of two transistors in series. Because the relative strengths of the two transistors are not specified, this circuit cannot be simulated, even though its output is undefined only when both transistors are conducting.

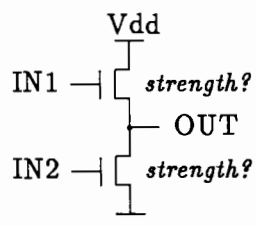


Figure 2.1: A Simple Transistor Network (strengths not specified)

While strength information can be derived from low-level circuit descriptions (e.g. device geometry), it is rarely present in the higher-level specifications used in today's advanced design tools. Uncertainty is inherent in top-down design, where all of the implementation details may not be available when testing the circuit at an intermediate design stage. Simulating a transistor network to verify its behavior before calculating the exact transistor size ratios is one such example.

In this chapter, we describe an extension to an existing switch-level simulator, MOSSIM II [Brya84], capable of handling uncertain transistor strengths. MOSSIM II assumes a certain underlying mathematical structure. We represent transistor strengths by ranges of values and create an interval extension of this algebraic structure to accommodate strength ranges. Our method is simple yet accurate, as we prove later in this chapter. The resulting interval switch-level simulation algorithm, Intssim, has a time complexity comparable to that of the original. In addition, our algorithm also models uncertain node sizes.

2.1 The Transistor Model

2.1.1 Traditional Approach

In switch-level simulation, a transistor is modeled as an ideal switch, in which the gate of the transistor controls the flow of current between the other two terminals (See Figure 2.2). The state of a transistor (i.e., whether the switch is *open* (nonconducting), *closed* (conducting), or *indeterminate*) is uniquely determined by the transistor type and the signal value at its gate. Table 2.1 displays the state information for the three transistor types most commonly used in VLSI technologies.

A network of transistors is modeled as a weighted, undirected graph [Brya84]. The vertices of this graph represent the source or drain terminals of the transistors in the network and the edges indicate the connectivity between these terminals. The vertices of the graph are divided into two mutually exclusive classes: inputs (which are direct connections to the voltage sources Vdd and Gnd) and storage nodes (everything else). Associated with each vertex are two attributes: signal value and node size. The first attribute is the current signal value of the node (0, 1, or X). The size of a node is a measure of its capacitance. Rather than using the actual capacitances themselves, switch-level simulators usually employ a set of discrete nodes sizes. If the vertex is an

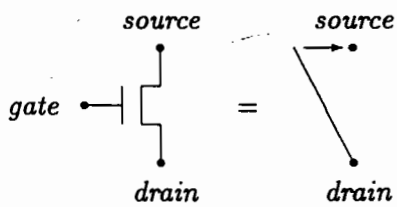


Figure 2.2: The Switch-Level Transistor Model

Gate value	Transistor type		
	n-type	p-type	d-type
0	open	closed	closed
1	closed	open	closed
X	indeterminate	indeterminate	closed

Table 2.1: Transistor State Table

input node, it is assigned size ω, indicating infinite capacitance. All other nodes are ranked by capacitance and assigned values from the set $\{\kappa_1,...,\kappa_{max}\}$.

Each edge in the graph corresponds to a transistor in the network and indicates its state (See Figure 2.3). A graph edge is drawn between two vertices in the graph, if the transistor linking that pair of nodes is *potentially* closed. There are two types of edges: *1-edges* for transistors in a *closed* state and *X-edges* for transistors in an *indeterminate* state. Associated with each edge is a strength attribute, which is a measure of the conductance of the transistor. Conductance is determined by the geometry of a transistor and, in particular, by the ratio of its length to its width. Instead of using the actual ratios in the simulation, switch-level algorithms employ a finite set of discrete strengths, like those used to specify node sizes. To determine these strengths, the transistors are ranked according to ratio size and transistor type and are assigned values, indicating relative conductance, from a set of discrete possibilities, $\{\gamma_1,...,\gamma_{max}\}$.

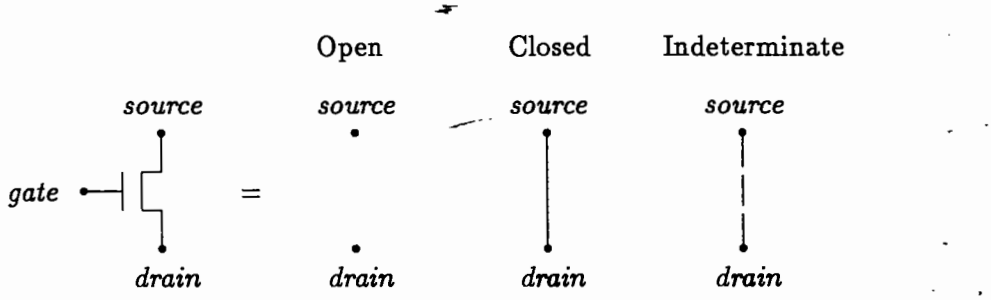


Figure 2.3: Transistor State and Graph Edges

2.1.2 Unknown Transistor Strength

The set of transistor strengths $T = \{\gamma_1, ..., \gamma_{max}\}$ is ordered such that $\gamma_1 < \gamma_2 < ... < \gamma_{max}$. A transistor with unknown or indeterminate strength may assume any number of consecutive strengths chosen from this set; therefore, we represent a transistor's strength by a range of these values or an interval. If the relative strength of a transistor is precisely known, then we represent it as a degenerate interval. Our extensions will manipulate degenerate and non-degenerate intervals with equal ease; thus, our extension subsumes current switch-level simulation algorithms.

2.1.3 An Example

Figure 2.4 depicts a transistor network containing unknown transistor strengths. All transistors in the network are given unique names and assigned interval strengths. From this network, the connectivity graph in Figure 2.5 is generated. Each graph vertex is labeled with a unique identifier followed by its signal value and size attributes in parentheses. For instance, vertex 0 represents the source terminal of transistor t_0. Since it is connected to the voltage source Vdd, this node has a signal value of 1 and an infinite capacitance ω. Likewise, vertex 4 represents the drain terminals of transistors t_2, t_3, and t_5, which are all connected to ground. This vertex thus has a signal value of 0 and an infinite capacitance ω. The rest of the vertices represent transistor terminals that are not directly connected to either Vdd or Gnd; therefore, these vertices are assigned an uncertain signal value X and a relatively weak capacitance κ_1. Each edge in the graph represents a conducting transistor and is labeled with its

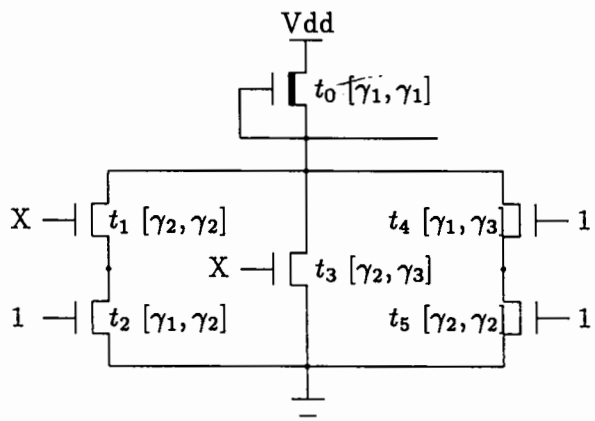


Figure 2.4: Transistor Network with Unknown Transistor Strengths

name and associated strength. 1-edges appear as solid lines and X-edges as dashed lines.

2.2 The Simulation Algorithm

To analyze a network containing transistors with unknown strength, we must effectively determine the steady-state value of the network under all possible strength assignments to each transistor of unknown strength. If the steady-state value of a node is the same regardless of the strengths assigned to these transistors, then that signal value is taken to be the steady-state value of the node. If, however, different strength assignments yield different signal values for the same node, then the steady-state value of the node is taken to be X.

Table 2.2 presents this analysis for the network in Figure 2.5. All combinations of possible strengths are assigned to the unknown-strength transistors in the network and the resulting steady-state is computed using the original MOSSIM II algorithm. Since the network must be analyzed for each possible strength combination, the basic simulation algorithm is executed $\prod_{i=1}^{k} g_i$ times, where k is the number of transistors with unknown strength and g_i is the number of different strengths that can be assigned to transistor i. This method has a worst-case time complexity of $O(g^k)$, where k is

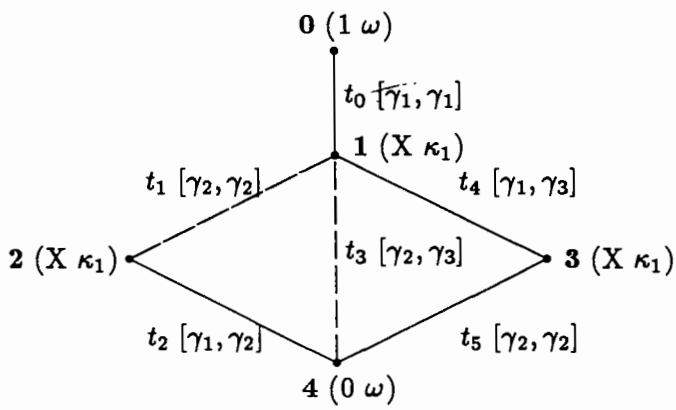


Figure 2.5: Connectivity Graph of Network with Unknown Transistor Strength

number of transistors with unknown strength and g is the maximum number of different transistor strengths. Clearly, a more efficient technique is needed. To remedy this problem, we create an interval version of MOSSIM II which more efficiently computes the steady-state value of the network.

2.2.1 The MOSSIM II Simulation Algorithm

We now review the MOSSIM II switch-level simulation algorithm [Brya84]. We begin with a few definitions. A *current path* through the transistor network is a path originating a signal source that passes through zero or more consecutive conducting transistors. The *signal value* of a path is the signal value of its source node. (e.g., Paths originating at Vdd have a signal value of 1. Paths originating at Gnd have a signal value of 0). The *strength* of a path is defined as the minimum of source node size (capacitance) and the strength (conductance) of the weakest transistor in the path. Bryant identifies three types of paths distinguished by their strengths: input paths, driven paths, and stored-charge paths. An input path consists of a signal node connected to Vdd or Gnd. Since they are directly connected to voltage sources, input paths are the strongest type of paths and have strength ω. Driven paths originate at Vdd or Gnd and pass through one or more transistors. The conductance of the transistors in the path determines the resulting strength of the signal; thus, driven paths have strengths in the range γ_1 to

Strength			Steady-state				
t_2	t_3	t_4	0	1	2	3	4
γ_1	γ_2	γ_1	1	X	X	0	0
γ_1	γ_2	γ_2	1	0	0	0	0
γ_1	γ_2	γ_3	1	0	0	0	0
γ_1	γ_3	γ_1	1	X	X	0	0
γ_1	γ_3	γ_2	1	0	0	0	0
γ_1	γ_3	γ_3	1	0	0	0	0
γ_2	γ_2	γ_1	1	X	0	0	0
γ_2	γ_2	γ_2	1	0	0	0	0
γ_2	γ_2	γ_3	1	0	0	0	0
γ_2	γ_3	γ_1	1	X	0	0	0
γ_2	γ_3	γ_2	1	0	0	0	0
γ_2	γ_3	γ_3	1	0	0	0	0
Final values			1	X	X	0	0

Table 2.2: Combinational Analysis of Network containing Unknown Strengths

γ_{max}. Stored-charge paths originate at storage nodes and pass through zero or more transistors. These paths provide the weakest signals, because the source of the signal is a finite reservoir of charge isolated from a voltage source. Such paths have strengths ranging from κ_1 to κ_{max}, depending on the size of the source node. With the addition of an identity element λ, Bryant presents the following natural ordering of path strengths:

$$\omega > \gamma_{max} > ... > \gamma_1 > \kappa_{max} > ... > \kappa_1 > \lambda. \tag{2.1}$$

To compute the resultant steady-state value of a given node in the network, Bryant computes all paths to the node and chooses the one with the maximum strength. The signal value of this path determines the steady-state value of the node. If a second path with this same maximum strength but a different signal value exists, then the resulting steady-state value of the node is X (indeterminate).

To perform these calculations, Bryant employs three operations: $+$ (maximum), $\cdot$ (minimum), and $\sim$ (blocking). The first two operators, $+$ and $\cdot$, are the standard maximum and minimum functions. Given two path strengths, the $+$ operator compares them and returns the larger of the two. Similarly, the $\cdot$ operator returns the smaller path strength. Bryant chose this notation for his minimum and maximum operators, because he uses these operators in his matrix computations (described later in this

section) as if he was performing matrix multiplication and addition. The third operator, $\sim$, compares two path strengths and discards the first if it is less than the second. Since the second path has greater strength, it prevents the first path from influencing the steady-state of the node. This blocking function is defined as follows:

$$a \sim b \equiv \begin{cases} a & \text{if } a \geq b \\ \lambda & \text{otherwise} \end{cases} \tag{2.2}$$

The MOSSIM II algorithm has four distinct stages. In the first stage, Bryant creates the matrices which describe the transistor network. The first two matrices, s and y, contain the size and initial signal value of each vertex in the connectivity graph representing the transistor network; thus, s_i contains the size or capacitance of node i and y_i its initial value. The connectivity between these nodes is given by two matrices, $G1$ and GX; one to hold the *1-edges* and the other to hold the *X-edges*. If there is a closed transistor joining nodes i and j in the transistor network, then there is a 1-edge between these nodes in the corresponding connectivity graph. An entry for this edge is made in matrix $G1$ by assigning the transistor strength of this closed transistor to $G1(i,j)$. Since connectivity is a commutative property, $G1(i,j) = G1(j,i)$. After all of the 1-edges are recorded, the remaining locations in $G1$ are filled with λ. The matrix GX is defined similarly, recording *X-edges* instead of *1-edges*.

When computing path strength, signal paths containing X-edges pose problems. Since a transistor with indeterminate state may be either closed or open, the corresponding edge may or may not actually exist in the graph. If considered, strong signal paths containing X-edges may prevent weaker signal paths which do not contain X-edges from influencing the signal value at a node. To prevent this undesirable condition, X-edges are ignored in the initial path analysis. Signal paths which do not contain any X-edges are called *definite* paths. *Blocking path strength* is defined as the strength of the strongest definite path to a vertex.

The second step of the simulation algorithm is to compute the blocking path strength for each of the nodes in the network. In MOSSIM II, blocking path strength is derived using the following recurrence:

$$\begin{aligned} q(i,0) &= s_i \\ q(i,r) &= s_i + \sum_{j=1}^{n} G1(i,j) \cdot q(j,r-1) \end{aligned} \tag{2.3}$$

The entry $q(i,r)$ gives the strength of the strongest definite path with length less than or equal to r to reach vertex i. The base case reflects the strength of the strongest path to node i with length 0. This path consists of the node itself and has a strength equal to the node size, s_i. The recurrence relation computes the maximum of the node size and the strengths of all paths through the network to node i originating at other nodes and passing through at most r closed transistors. The strength of the path to node i through the closed transistor linking nodes i and j is the minimum of the transistor strength $G1(i,j)$ and the strength of the strongest definite path $q(j,r-1)$ through the network to node j. Note that r is bounded by the number of vertices n in the graph; thus, $q_i = q(i,n)$. This new matrix q is the blocking strength matrix.

The third step is to compute the strongest unblocked pull-up and pull-down paths to each vertex. A pull-up path is a path that originates at a node with a signal value of 1 (or X). Likewise, a pull-down path originates at a node with a signal value of 0 (or X). To calculate unblocked pull-up paths, the algorithm begins at nodes with signal values of 1 (or X) and traces the paths from these nodes to all other vertices in the connectivity graph, computing the strongest path to each vertex. In contrast to the previous step, both definite and indefinite paths are included in this computation. Once the strongest pull-up path to a vertex is computed, it is compared with the blocking path strength for that vertex using the $\sim$ operator given in equation 2.2. If the strength of this pull-up path is less than the blocking path strength, then a stronger definite path exists; therefore, this pull-up path is discarded. Only paths with strength greater than or equal to the blocking strength are saved. By starting at nodes with signal values of 0 (or X), the strongest unblocked pull-down paths are similarly computed. The unblocked pull-up and pull-down paths to a vertex are then used to determine its steady-state.

Bryant introduces matrices u and d representing the strength of the strongest unblocked pull-up and pull-down paths to each vertex. To assist in generating these matrices, two new functions **up** and **down** are also introduced. These functions identify the sources of possible pull-up and pull-down paths and are defined as follows:

$$\mathbf{up}(s_i, y_i) \equiv \begin{cases} s_i & \text{if } y_i = 1 \text{ or } X \\ \lambda & \text{if } y_i = 0 \end{cases} \tag{2.4}$$

$$\mathbf{down}(s_i, y_i) \equiv \begin{cases} s_i & \text{if } y_i = 0 \text{ or } X \\ \lambda & \text{if } y_i = 1 \end{cases} \tag{2.5}$$

Given these two functions, the matrices u and d are generated by the following recurrences:

$$\begin{aligned} u(i,0) &= \mathbf{up}(s_i, y_i) \sim q_i \\ u(i,r) &= \left[\mathbf{up}(s_i, y_i) + \sum_{j=1}^{n} (G1(i,j) + GX(i,j)) \cdot u(j, r-1)\right] \sim q_i \qquad (2.6) \\ d(i,0) &= \mathbf{down}(s_i, y_i) \sim q_i \\ d(i,r) &= \left[\mathbf{down}(s_i, y_i) + \sum_{j=1}^{n} (G1(i,j) + GX(i,j)) \cdot d(j, r-1)\right] \sim q_i \qquad (2.7) \end{aligned}$$

The entry $u(i,r)$ gives the strength of the strongest unblocked pull-up path with length less than or equal to r and destination node i. The base case $u(i,0)$ computes the strength of the pull-up path consisting of the single node i, saving it only if its strength exceeds the blocking strength q_i. The recurrence relation computes the maximum strength of all pull-up paths to node i that pass through at most r transistors. The $\mathbf{up}(s_i, y_i)$ term represents the strength of the pull-up path containing only node i. The rest of the terms represent the strengths of the pull-up paths to node i originating at other nodes in the network. The strength of a pull-up path passing through the transistor linking nodes i and j is computed by taking the minimum of the transistor strength $G1(i,j) + GX(i,j)$[1] and the strength of the strongest unblocked pull-up path $u(j, r-1)$ to node j. Once the strength of the strongest pull-up path to node i is found, it is compared to the blocking strength q_i. If the strength of this pull-up path equals or exceeds the blocking strength, it is saved. As with q, the recursion terminates in at most n steps; thus, u_i is assigned the value $u(i,n)$. Matrix d is computed analogously.

In the final step, the results of the previous computation are used to determine the steady-state response of the network. If there is an unblocked pull-up path to a vertex and no unblocked pull-down paths, then the new signal value at the vertex is 1. Similarly, if there is an unblocked pull-down path to the vertex and no unblocked pull-up paths, then the new signal value is 0. If both unblocked pull-up and pull-down paths exist, then the signal value is taken to be X. This function is defined as follows:

$$Y_i \equiv \begin{cases} 1 & \text{if } d_i = \lambda \\ 0 & \text{if } u_i = \lambda \\ X & \text{otherwise} \end{cases} \qquad (2.8)$$

[1] In this computation, 1-edges and X-edges are considered.

2.2.2 Extensions to Handle Unknown Transistor Strength

To create an algorithm for simulating transistor networks containing uncertain transistor strengths, we modify the path analysis algorithm given above to accommodate intervals. In particular, we define three interval operators which are the logical extensions of the discrete operators $+$, $\cdot$, and $\sim$. Each interval operator is obtained from its non-interval counterpart using the standard derivation: $F(\hat{x}, \hat{y}) \equiv \{f(x, y) \mid x \in \hat{x}\ and\ y \in \hat{y}\}$. In other words, we derive the interval definition of each operator by applying the associated discrete function to all combinations of elements from each interval and taking the union of the results. The resulting maximum operator for intervals $\boxplus$ and the minimum operator $\boxdot$ are defined as follows:

$$[a, b] \boxplus [c, d] \equiv [a + c, b + d] \tag{2.9}$$

$$[a, b] \boxdot [c, d] \equiv [a \cdot c, b \cdot d]. \tag{2.10}$$

In addition to these two operators, we introduce a blocking operator for intervals $\simeq$ defined as follows:

$$[a, b] \simeq [c, d] \equiv \begin{cases} [a + c, b] & \text{if } b \geq c \\ [\lambda, \lambda] & \text{otherwise} \end{cases} \tag{2.11}$$

This operator compares two interval strengths and returns only those portions of the first interval that equal or exceed the lower bound of the second. In essence, the lower bound of the second interval acts as the blocking strength. Any strength less than this blocking strength is discarded, because we already have a definite path with greater strength.

Having defined these interval operations, we substitute them into the path analysis equations given in the previous section to generate a switch-level simulation algorithm which handles unknown transistor strength.

2.2.3 Extensions to Handle Uncertain Node Size

It should be obvious that our interval version of the switch-level simulation algorithm can also be applied to uncertain node sizes, since node size and transistor strength have the same underlying algebraic structure. To handle uncertain node sizes, we represent them as intervals over the ordered set $\{\kappa_1, \kappa_2, ..., \kappa_{max}, \omega\}$. We then employ the interval operations given above to manipulate these size ranges. No further changes to the algorithm are required.

	s_i	y_i
0	$[\omega, \omega]$	1
1	$[\kappa_1, \kappa_1]$	X
2	$[\kappa_1, \kappa_1]$	X
3	$[\kappa_1, \kappa_1]$	X
4	$[\omega, \omega]$	0

Table 2.3: Describing the Network – the Vertices

G1					
	0	1	2	3	4
0	$[\lambda, \lambda]$	$[\gamma_1, \gamma_1]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
1	$[\gamma_1, \gamma_1]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_1, \gamma_3]$	$[\lambda, \lambda]$
2	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_1, \gamma_2]$
3	$[\lambda, \lambda]$	$[\gamma_1, \gamma_3]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_2, \gamma_2]$
4	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_1, \gamma_2]$	$[\gamma_2, \gamma_2]$	$[\lambda, \lambda]$

GX					
	0	1	2	3	4
0	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
1	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_2, \gamma_2]$	$[\lambda, \lambda]$	$[\gamma_2, \gamma_3]$
2	$[\lambda, \lambda]$	$[\gamma_2, \gamma_2]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
3	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
4	$[\lambda, \lambda]$	$[\gamma_2, \gamma_3]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$

Table 2.4: Describing the Network – the Edges

2.2.4 Examples

Tables 2.3-2.5 present the analysis of the transistor network given in Figure 2.5. The first two tables describe the initial state of the transistor network. Table 2.5 displays the resultant blocking path strength, unblocked pull-up and pull-down paths, and the final steady-state for each node in the network. Note that the steady state values computed are identical to those derived in Table 2.2.

To further illustrate how the interval operations are employed to determine these results, we examine the computations for one of the nodes. Table 2.6 enumerates all paths through the network which terminate at node 3. The table gives both the signal value and strength of each path. To calculate the blocking path strength of node 3,

	q_i	u_i	d_i	Y_i
0	$[\omega,\omega]$	$[\omega,\omega]$		1
1	$[\gamma_1,\gamma_2]$	$[\gamma_1,\gamma_1]$	$[\gamma_2,\gamma_3]$	X
2	$[\gamma_1,\gamma_2]$	$[\gamma_1,\gamma_1]$	$[\gamma_2,\gamma_2]$	X
3	$[\gamma_2,\gamma_2]$		$[\gamma_2,\gamma_3]$	0
4	$[\omega,\omega]$		$[\omega,\omega]$	0

Table 2.5: Interval Algorithm Results

we find the strength range of the strongest definite path. In this computation, we consider four paths: the path from node 0 through the pull-up transistor to node 3 (with strength $[\gamma_1,\gamma_1]$), the path originating at node 1 (with strength $[\kappa_1,\kappa_1]$), the path consisting of node 3 itself (with strength $[\kappa_1,\kappa_1]$), and the pull-down path originating at node 4 (with strength $[\gamma_2,\gamma_2]$). The blocking path strength is found by computing the maximum of these strengths; thus, $q_3 = [\gamma_1,\gamma_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\gamma_2,\gamma_2] = [\gamma_2,\gamma_2]$. In the next stage of the algorithm, we calculate unblocked pull-up and pull-down path strengths. We accomplish this by finding the strength ranges of the strongest pull-up and pull-down paths and then comparing them to the blocking path strength to eliminate blocked paths. The strongest pull-down path strength is the maximum strength of all paths originating at nodes with value 0 or X; therefore, the maximum pull-down path strength is $[\kappa_1,\kappa_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\gamma_2,\gamma_2] \boxplus [\gamma_1,\gamma_3] \boxplus [\gamma_1,\gamma_2] = [\gamma_2,\gamma_3]$. Likewise, the strongest pull-up path strength is $[\gamma_1,\gamma_1]$. When compared to the blocking path strength for node 3, we discover that there are no unblocked pull-up paths (since $[\gamma_1,\gamma_1] \simeq [\gamma_2,\gamma_2] = [\lambda,\lambda]$), but that unblocked pull-down paths do exist (since $[\gamma_2,\gamma_3] \simeq [\gamma_2,\gamma_2] = [\gamma_2,\gamma_3]$). Therefore, we conclude that the steady-state value of node 3 is 0.

The CMOS selector shown in Figure 2.6 is another example. Since no explicit strengths are given, we assign strength $[\gamma_1,\gamma_{max}]$ to all transistors and simulate the network. The simulation results given in Table 2.7 show that if all the transistors in the circuit have equal strength, then the network does not exhibit the desired switching behavior when the selected input value changes from 1 to 0. Instead of changing from 0 to 1, the output becomes undefined. Because the pull-up path is too strong, it competes with the input signal, thus producing an undefined steady-state. The conflict is resolved by assigning a weak transistor strength to the uppermost p-transistor and giving all

Path	X-edges?	Value	Strength
0,1,3	No	1	$[\omega,\omega] \boxdot [\gamma_1,\gamma_1] \boxdot [\gamma_1,\gamma_3] = [\gamma_1,\gamma_1]$
1,3	No	X	$[\kappa_1,\kappa_1] \boxdot [\gamma_1,\gamma_3] = [\kappa_1,\kappa_1]$
2,1,3	Yes	X	$[\kappa_1,\kappa_1] \boxdot [\gamma_2,\gamma_2] \boxdot [\gamma_1,\gamma_3] = [\kappa_1,\kappa_1]$
3	No	X	$[\kappa_1,\kappa_1]$
4,3	No	0	$[\omega,\omega] \boxdot [\gamma_2,\gamma_2] = [\gamma_2,\gamma_2]$
4,1,3	Yes	0	$[\omega,\omega] \boxdot [\gamma_2,\gamma_3] \boxdot [\gamma_1,\gamma_3] = [\gamma_1,\gamma_3]$
4,2,1,3	Yes	0	$[\omega,\omega] \boxdot [\gamma_1,\gamma_2] \boxdot [\gamma_2,\gamma_2] \boxdot [\gamma_1,\gamma_3] = [\gamma_1,\gamma_2]$

Table 2.6: Network paths to Node 3

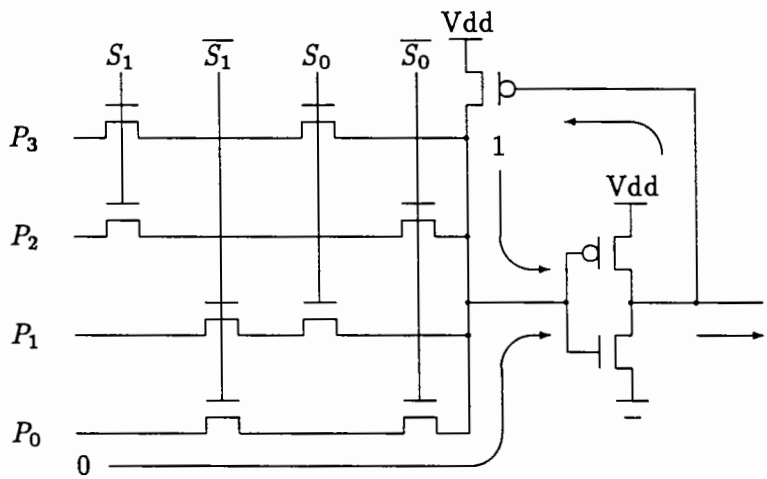


Figure 2.6: CMOS Selector with Feedback

P_i transition	Initial pull-up state	Equal strength Output value	Weak pull-up Output value
$0 \rightarrow 0$	open	1	1
$1 \rightarrow 0$	closed	X	1
$X \rightarrow 0$	indeterminate	X	1
$0 \rightarrow 1$	open	0	0
$1 \rightarrow 1$	closed	0	0
$X \rightarrow 1$	indeterminate	0	0
$0 \rightarrow X$	open	X	X
$1 \rightarrow X$	closed	X	X
$X \rightarrow X$	indeterminate	X	X

Table 2.7: Simulation Results for CMOS Selector Circuit

other transistors in the network greater relative strength. Once this change is made, the circuit exhibits the desired behavior as shown in Table 2.7.

2.3 Proof of the Method

In this section, we formally prove that our interval method correctly computes the resultant steady-state value of a transistor network with interval transistor strengths and node sizes. In his presentation of MOSSIM II, Bryant lists the fundamental properties of his path analysis algebra. He further proves that his algorithm works whenever these properties hold. In the next section, we summarize these properties. We then present our interval algebra and show that it exhibits the same properties. Finally, we conclude that if the original algorithm using Bryant's algebra correctly analyzes a circuit with discrete transistor strengths and node sizes, then the extended algorithm employing our interval algebra will correctly analyze the same circuit with interval transistor strengths and node sizes.

2.3.1 Bryant's Algebra

Let S be an ordered set of discrete transistor strengths and node sizes; that is $S = \{\lambda, \kappa_1, ..., \kappa_{max}, \gamma_1, ..., \gamma_{max}, \omega\}$ such that $\lambda < \kappa_1 < ... < \kappa_{max} < \gamma_1 < ... < \gamma_{max} < \omega$. Bryant defines three operators: maximum ($+$), minimum ($\cdot$), and blocking ($\sim$) over the set S. He then proves that these operators have the properties necessary to correctly determine the steady-state. The fundamental properties of the maximum operation

are: (1) commutativity, (2) associativity, (3) idempotence (4) identity λ, (5) closure, and (6) monotonicity (i.e., $a \geq b \Rightarrow (a+c) \geq (b+c)$ and $(c+a) \geq (c+b)$). Similarly, the minimum operator must have the following properties: (1) associativity, (2) identity ω, (3) annihilator λ, (4) closure, and (5) monotonicity. Furthermore, minimum distributes over maximum. The blocking operator is (1) idempotent, (2) distributes over both minimum and maximum, and (3) is monotonic in its first argument (i.e., $a \geq b \Rightarrow (a \sim c) \geq (b \sim c)$).

2.3.2 The Interval Algebra

Let S' be the set of all intervals over S (i.e., $S' = \{[a,b] \mid a,b \in S \text{ and } a \leq b\}$). We introduce three interval operations over the set S': interval maximum ($\boxplus$), interval minimum ($\boxdot$), and interval blocking ($\simeq$). We now show that these interval operators have the same properties as their discrete counterparts.

Proof. We first examine the properties of the interval maximum operator. Since $\boxplus$ is defined in terms of $+$ which is commutative, associative, and idempotent, it is trivial to show that interval maximum is also (1) commutative, (2) associative, and (3) idempotent. (4) Given that λ is the identity element of $+$, $[\lambda, \lambda]$ is the identity element of $\boxplus$. (5) To prove that S' is closed over $\boxplus$, we must show that $[a,b] \boxplus [c,d] \in S'$ for all $[a,b], [c,d] \in S'$. By definition, $[a,b] \boxplus [c,d] = [a+c, b+d]$. Since $+$ is closed over S, $a+c$ and $b+d$ are in S. All that remains is to show that $a+c \leq b+d$. This follows from the definition of an interval ($[a,b]$ implies $a \leq b$) and the monotonicity of $+$. Hence, $[a,b] \boxplus [c,d] \in S'$. (6) To even be able to say whether $\boxplus$ is monotonic, we require an interval definition for the relational operator $\geq$. We define $\geq$ for intervals as follows:

$$[a,b] \geq [c,d] \Longleftrightarrow a \geq c \text{ and } b \geq d.$$

Assuming this definition, $\boxplus$ is monotonic if

$$[a,b] \geq [c,d] \Longrightarrow ([a,b] \boxplus [e,f]) \geq ([c,d] \boxplus [e,f]) \text{ and } ([e,f] \boxplus [a,b]) \geq ([e,f] \boxplus [c,d]).$$

Applying the interval definition of $\geq$, this simplifies to

$$\begin{aligned} a \geq c \text{ and } b \geq d \quad &\Longrightarrow \quad a+e \geq c+e \text{ and } b+f \geq d+f \\ &\text{and} \quad e+a \geq e+c \text{ and } f+b \geq f+d. \end{aligned}$$

This follows from the monotonicity of $+$.

The properties of the interval minimum operator may be proved using similar arguments. Clearly, $\boxdot$ is (1) associative, (2) has identity $[\omega, \omega]$ and (3) annihilator $[\lambda, \lambda]$, (4) is closed, and (5) monotonic. Applying the definitions of $\boxdot$ and $\boxplus$ and noting that $\cdot$ distributes over $+$, we see that $\boxdot$ distributes over $\boxplus$.

Next, we examine the properties of the interval blocking operator $\simeq$. (1) To prove that $\simeq$ is idempotent, we must show that $[a,b] \simeq [a,b] = [a,b]$ for all $[a,b] \in S'$. This follows directly from the definition of $\simeq$ and the idempotence of $+$, since $[a,b] \simeq [a,b] = [a+a,b]$. (2) The next property is that $\simeq$ distributes over $\boxplus$. We must show that $([a,b] \boxplus [c,d]) \simeq [e,f] = ([a,b] \simeq [e,f]) \boxplus ([c,d] \simeq [e,f])$ for all $[a,b],[c,d],[e,f] \in S'$. We begin by applying the definitions of $\simeq$ and $\boxplus$ to expand each expression. Expanding the first expression yields the following:

$$([a,b] \boxplus [c,d]) \simeq [e,f] = \begin{cases} [a+c+e, b+d] & \text{if } b+d \geq e \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

Similarly, expanding the second expression gives:

$$([a,b] \simeq [e,f]) \boxplus ([c,d] \simeq [e,f]) = \begin{cases} [(a+e)+(c+e), b+d] & \text{if } b \geq e \text{ and } d \geq e \\ [a+e, b] & \text{if } b \geq e > d \\ [c+e, d] & \text{if } d \geq e > b \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

To determine whether these definitions are equivalent, we examine each of the four cases. In the first case, $b \geq e$ and $d \geq e$. Clearly, this implies $b+d \geq e$. Since $+$ is commutative, associative, and idempotent, $(a+e)+(c+e) = a+c+e$; thus, the expressions are equivalent. Second, consider the case $b \geq e > d$. Here also $b+d \geq e$; therefore, we must show that $[a+c+e, b+d] = [a+e, b]$. We are given $b > d$, thus $b+d = b$. Since $e > d \geq c$, $a+c+e = a+e$. The third case ($d \geq e > b$) is similar to the second and is proved by repeating the previous arguments with $[a,b]$ and $[c,d]$ exchanged. The final case is trivial, since both expressions yield $[\lambda, \lambda]$ when $b < e$ and $d < e$. Hence, both expressions are equivalent.

Next we prove that $\simeq$ distributes over $\boxdot$. We must show that $([a,b] \boxdot [c,d]) \simeq [e,f] = ([a,b] \simeq [e,f]) \boxdot ([c,d] \simeq [e,f])$ for all $[a,b],[c,d],[e,f] \in S'$. The argument here is similar. We begin by expanding each expression using the definitions of $\boxdot$ and $\simeq$ and then perform case analysis to determine equivalence. The first expression here

gives:

$$([a,b] \boxdot [c,d]) \simeq [e,f] = \begin{cases} [(a \cdot c) + e, b \cdot d] & \text{if } b \cdot d \geq e \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

Given that λ is the annihilator for minimum, the second expression yields the following when expanded:

$$([a,b] \simeq [e,f]) \boxdot ([c,d] \simeq [e,f]) = \begin{cases} [(a+e) \cdot (c+e), b \cdot d] & \text{if } b \geq e \text{ and } d \geq e \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

At this point the proof becomes trivial, since these expressions are clearly equivalent.

Finally, we prove that (3) $\simeq$ is monotonic in its first argument (i.e., $[a,b] \geq [c,d] \Longrightarrow ([a,b] \simeq [e,f]) \geq ([c,d] \simeq [e,f])$). From the interval definition of $\geq$ and the assumption that $[a,b] \geq [c,d]$, $a \geq c$ and $b \geq d$; thus, we only have to examine three cases. In the first case, we have $b \geq d \geq e$; therefore, we must show that $a + e \geq c + e$. Clearly this is true, since $+$ is monotonic. The second case is trivial, since if $b \geq e > d$, then $[a + e, b] \geq [\lambda, \lambda]$. The last case occurs when both $b < e$ and $d < e$. In this case, both expressions yield $[\lambda, \lambda]$. We have thus shown that $\simeq$ is monotonic in its first argument.

This proves that our interval extensions to the operators in Bryant's algebra exhibit the same properties. Since Bryant proves that the algorithm works whenever these properties hold, we conclude that our interval algorithm employing these interval operators will correctly compute the steady-state of a transistor network with interval transistor strengths and node sizes. □

2.4 Performance

Our interval path analysis algorithm, Intssim, has been implemented in the C programming language and runs on several machines (VAX-11/780, Encore Multimax, and SUN SPARCstation). We ran both the Intssim and MOSSIM II programs on identical circuits and recorded their response times. Experiments show that the interval version takes about twice as long as MOSSIM II to compute the steady-state value of a transistor network. This is the expected result, since the interval algorithm performs two calculations for each minimum, maximum, or blocking operator encountered. The worst-case time complexity of both path analysis algorithms is $O(s+t)$, where t represents the number of transistors and s represents the number of distinct node sizes and transistor strengths.

2.5 Summary

In this chapter, we considered the problem of simulating transistor networks containing uncertain transistor strengths and node sizes. We represented uncertain transistor strengths by strength ranges and developed an interval algebra to manipulate these ranges. We further generalized an existing switch-level simulator, MOSSIM II, to create a simulation algorithm for networks with interval transistor strengths and node sizes. Our algorithm has a running time of about twice that of the original and a comparable worst-case time complexity. We also presented a formal proof of correctness for our method.

Chapter 3

Modeling Uncertainty in RC Timing Analysis I

Timing verification is an important aspect of VLSI design. Its purpose is to predict the operating speed of a VLSI chip before it is fabricated. This speed is a function of three components: structure, technology, and the manufacturing process. The first two of these are well known to chip designers. Consider, for example, carry-propagate and carry-lookahead adders. Although both perform the same function, they have different structures and hence operating speeds. A comparison of CMOS and nMOS technologies provides an example of the effects of technology on performance. By replacing slow depletion mode pull-ups by complementary logic, CMOS designs achieve faster operating speeds than their nMOS counterparts. The third component, the manufacturing process, is also an important factor in chip performance. During fabrication, even slight variations in any of a multitude of processing parameters (e.g., exposure time, temperature, gas pressure) produce variations in the process and consequently in device parameters. For example, over and under-etching, oxide growth rates, and ion implantation levels all affect the electrical characteristics of the resulting circuit. In particular, they affect the transistor sizes, threshold voltages, resistance factors, and capacitance factors used to perform timing analysis. As a result, the circuit specifications given for each technology are only approximations to the true values. These parameters actually have a range of possible values, typically varying between 1 and 20 percent from those stated. Currently, most timing verifiers use only the average values given to estimate circuit performance, but it is important to know how the circuit will

operate with other possible combinations of these values.

While there are many timing verifiers in use today [Hitc82, Joup83, McWi80, Oust83, Wall86], most do not address the problem of parameter uncertainty. A few [Hitc82, McWi80, Wall86], however, provide partial solutions to this problem. These logic-level timing verifiers use bounds on the delay of each macro-component (e.g., AND gates, OR gates, flip-flops) to compute bounds on the performance of the entire circuit, but none of them discusses how these component delays are derived initially. We present a method for RC timing analysis (i.e., computing the delay of a transistor network from the resistances and capacitances of its parts) that can be used to compute these macro-component delays.

The most widely-used approach for modeling parameter uncertainty within existing simulation systems is the Monte Carlo method [Sobo75]. With this technique, analysis is repeated for random combinations of values chosen from within the acceptable range of each parameter. Once computed, these results are tabulated and returned. Unfortunately, accurately determining bounds on the behavior of a circuit requires a large number of trials; thus, while Monte Carlo simulation can be effective, it is also very time consuming. Our goal is to achieve similar accuracy in less time.

In this chapter, we present a theoretical framework based on interval algebra for handling parameter uncertainty in RC analysis. We then illustrate our approach by modifying an existing timing analyzer, Crystal [Oust83, Oust84, Oust86], to create an RC analysis algorithm that incorporates this framework. Given a circuit description and bounds on all circuit parameters (transistor lengths, widths, threshold voltage, capacitances, and resistances), our algorithm computes best and worst-case bounds on the delay of a transistor network. Although we employ Crystal for these experiments, the framework may also be applied to other RC analysis algorithms. We conclude this chapter with an analysis of our method and a comparison with Monte Carlo simulation.

3.1 RC Analysis in Crystal

Crystal is a data-independent timing analyzer for sequential MOS VLSI circuits. Given a design description extracted from a mask layout, Crystal computes and returns a list of the *critical* paths through the circuit. This algorithm has three phases: breaking the circuit into pieces, estimating delay through the pieces (using RC analysis), and using these delays to locate critical paths.

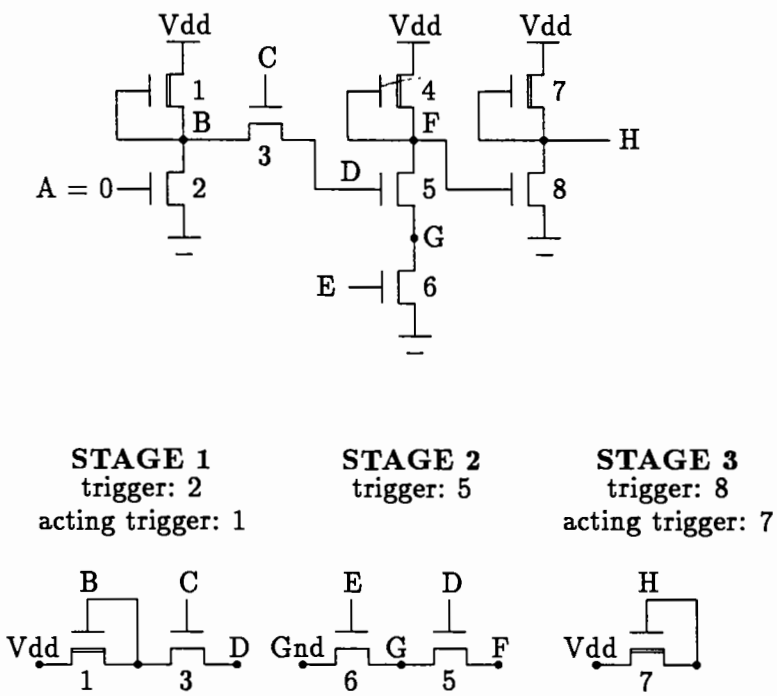


Figure 3.1: Decomposing a Circuit into Stages

In the first phase, Crystal decomposes the circuit into chains of transistors called *stages*. Each stage represents an electrical path through one or more transistors originating at a strong signal source (Vdd or Gnd) and terminating at an output node or transistor gate (called the *target*). A stage is activated by the last transistor in the path to switch on. This transistor is called the *trigger*. In nMOS stages containing depletion load transistors, the trigger transistor is not one of the transistors in the stage; therefore, the depletion load transistor closest to the target acts as the trigger for the stage. For example, stage 1 in Figure 3.1 is activated by a falling signal value at the gate of transistor 2. Since transistor 2 is not on the electrical path from the signal source Vdd to the target node D for the stage, depletion load transistor 1 acts as the trigger. The target transistor for one stage becomes the trigger transistor for the next stage. For an example of circuit decomposition, see Figure 3.1.

In the second phase of critical path analysis, Crystal estimates the delay of each stage. Given the sizes and types of the transistors in the stage, the parasitic resistances

and capacitances of the nodes, the trigger transistor, and the waveform at the gate of the trigger transistor, a delay modeler generates the resulting waveform at the target of the stage. Crystal supports three different models for computing stage delay: the linear RC model, the Slope model, and the PR-Slope model. Of the three models, the PR-Slope model is the most accurate. We will discuss this method in detail in the next section.

In the final phase of analysis, Crystal employs these stage delays to locate critical paths in the design using a longest-path algorithm with a depth-first scanning order. For further details, see [Oust83].

3.1.1 The PR-Slope Model

To estimate delay, the PR-Slope model begins by determining a resistance and capacitance for each node and transistor along the stage. Node capacitances and resistances can either be provided by the circuit extractor or, in some older systems, computed by Crystal directly from the geometry of the node. In the latter case, Crystal is given an area and perimeter for each substrate layer in the node and computes the capacitance and resistance contribution of each. The substrate resistances and capacitances are then summed to compute the resistance and capacitance of the node. In addition to the above contributions, the gate-to-source capacitances of adjacent branch transistors are also included in the node calculations.

A transistor is modeled as a perfect switch in series with a resistor. This resistance is fixed for non-trigger transistors and is computed from the transistor type and geometry. Specifically, a resistance factor is determined by table lookup on transistor type, usage, and signal value. This factor is then multiplied by the length/width ratio of the transistor to generate an estimate of its effective resistance.

The effective resistance of the trigger transistor is more accurately modeled. This resistance is not a constant, but a function of the input waveform at the gate of the transistor. In particular, resistance is a function of the *rise time* of the gate signal, which is the rate at which the signal is changing when it crosses the threshold voltage. The faster the gate signal changes, the lower the effective resistance across the transistor. To model this effect, the timing analyzer combines the rise time at the transistor gate, the load being driven, and the transistor size to compute a *rise time ratio.* It then consults a table indexed by these rise time ratios to find the resistance factor of the

trigger transistor. This factor is multiplied by the aspect ratio of the transistor to determine its resistance.

To calculate the effective capacitance of each transistor, Crystal determines the capacitance factor by table lookup on transistor type and multiplies this value by the area of the transistor. All transistor characterization tables used in these computations are generated by running SPICE simulations on sample circuits and compiling the results.

Having calculated the resistances and capacitances of each node and transistor in the path, Crystal uses these values to compute the total delay of the stage. The PR-Slope model employs the RC equations presented by Penfield and Rubinstein in [Rubi83]. Since a Crystal stage is just an RC line, these delay equations simplify to the following sum:

$$D = \sum_{t<i\leq T} \sum_{S<j\leq i} C_i R_j \tag{3.1}$$

where t represents the trigger transistor, T the target node, S the source node, R_j the resistance of element j, and C_i the capacitance at element i. For example, the delay of Stage 1 in Figure 3.1 is given by the following equation:

$$D = C_B(R_1 + R_B) + C_3(R_1 + R_B + R_3) + C_D(R_1 + R_B + R_3 + R_D). \tag{3.2}$$

In this example, the node capacitances and resistances are denoted by alphabetic subscripts and the transistor capacitances and resistances are denoted by numeric subscripts.

3.2 Bounds for RC Analysis

The PR-Slope model described above uses the average resistance and capacitance values of circuit devices to compute a single estimate of circuit delay. Our goal, however, is to model the effects of device parameter variation on delay; therefore, we must modify this RC analysis algorithm to compute delay bounds. We begin with a few definitions. Let $F(x_1, x_2, ..., x_n)$ be any real-valued, arithmetic function of n independent variables such that $F : \Re^n \rightarrow \Re$ and $x_i \in \Re$. Now assume that the value of each variable, x_i, is not precisely known. Instead, we are given a range of possible values for each x_i, $\hat{x}_i = [min_i, max_i]$. The tightest bounds on the solutions to F are found by evaluating the function over all possible combinations of input values and taking the union of the

results. This is called the *United Extension of* F [Moor79, p. 18] and is defined by the following:

$$\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \equiv \bigcup_{x_i \in \hat{x}_i} F(x_1, x_2, ..., x_n). \tag{3.3}$$

For timing analysis, $F(x_1, x_2, ..., x_n)$ represents delay Equation 3.1 and the x_i represent the capacitances, resistances, and transistor sizes used to compute this delay. If the value of any of these variables is not precisely known, then the tightest bounds on the solution may be found by computing $\hat{F}$. Because the intervals are defined over $\Re$, each contains an infinite number of possible assignments; therefore, we cannot compute $\hat{F}$ directly. We can approximate $\hat{F}$ by dividing each interval into $k-1$ subintervals and evaluating F at the k endpoints. Since each variable x_i can assume any one of these k possible values, we must evaluate function F k^n times to compute its United Extension. Even for $k = 2$, this computation requires 2^n invocations of F. Since the number of variables n tends to be large even for small stages (e.g., the PR-Slope Model requires six variables per transistor and a minimum of two variables per node), this is not a feasible approach.

3.2.1 A First Approximation

To approximate $\hat{F}$, we create an interval version of the delay equation. Let I be the set of all intervals over $\Re$; thus, $I \equiv \{[a,b] \mid a, b \in \Re\}$. To compute stage delay, we require an interval arithmetic. The needed interval arithmetic operations over the set I are addition ($\oplus$), subtraction ($\ominus$), multiplication ($\odot$), and division ($\oslash$) defined as follows:

$$[a,b] \oplus [c,d] \equiv [a+c, b+d] \tag{3.4}$$

$$[a,b] \ominus [c,d] \equiv [a-d, b-c] \tag{3.5}$$

$$[a,b] \odot [c,d] \equiv [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \tag{3.6}$$

$$[a,b] \oslash [c,d] \equiv [a,b] \odot [1/c, 1/d] \quad \textit{provided that } 0 \notin [c,d] \tag{3.7}$$

The properties of these operators are summarized in the introduction of this thesis. Now let $\mathcal{F} : I^n \rightarrow I$ be the interval equivalent of F derived by replacing the basic arithmetic operators $+$, $-$, $\cdot$, and $/$ in F by their respective interval counterparts $\oplus$, $\ominus$, $\odot$, and $\oslash$. This creates an interval version of the original function that returns bounds on the result when evaluated. We now show that $\mathcal{F}$ approximates $\hat{F}$.

Lemma 3.1 *If* $\hat{x}_1, ..., \hat{x}_n \in I$ *are degenerate, then* $F(x_1, ..., x_n) = \mathcal{F}(\hat{x}_1, ..., \hat{x}_n)$.

Proof. By induction on the number of operators (m) in F. By convention, if $\hat{x} = [x, x]$, then we write $\hat{x} = x$. Over the set of degenerate intervals, each interval operator is equivalent to its respective discrete operator as shown by the following: for all degenerate $\hat{a}, \hat{b} \in I$

$$\hat{a} \oplus \hat{b} = [a, a] \oplus [b, b] = [a + b, a + b] = a + b$$

$$\hat{a} \ominus \hat{b} = [a, a] \ominus [b, b] = [a - b, a - b] = a - b$$

$$\hat{a} \odot \hat{b} = [a, a] \odot [b, b] = [ab, ab] = ab$$

$$\hat{a} \oslash \hat{b} = [a, a] \oslash [b, b] = [a/b, a/b] = a/b.$$

Note also that the set of degenerate intervals is closed over these interval operators; thus, when evaluated over this set, $\mathcal{F}$ always returns a degenerate interval. For the basis ($m = 1$), F must be one of the following four functions: $F = a + b$, $F = a - b$, $F = ab$, or $F = a/b$. We have just demonstrated that $F = \mathcal{F}$ over the set of degenerate intervals for each of these cases. Now we are given that $F = \mathcal{F}$ over the set of degenerate intervals for all functions F consisting of less than m operators. We must show that $F = \mathcal{F}$ for functions with m operators. Any arithmetic function can be written as a binary parse tree, where each parent node contains one of the four arithmetic operators and each child node contains either a value or a subtree representing a subexpression of the function. If we remove the root of the parse tree, we are left with two subtrees representing subfunctions. If the original function had m operators, each of these subfunctions must contain less than m operators, since we removed the operator at the root of the parse tree. Let F represent the original function, let F_1 and F_2 represent these subfunctions, and let $*$ be the operator at the root of F. Clearly, $F = F_1 * F_2$. By the definition of $\mathcal{F}$, we know that $\mathcal{F} = \mathcal{F}_1 \circledast \mathcal{F}_2$ where $\circledast$ is the interval counterpart of the $*$ operator. Applying the assumption, we see that $\mathcal{F}_1 = F_1$ and $\mathcal{F}_2 = F_2$ over the set of degenerate intervals. Since $\circledast$ is equivalent to $*$ over the set of degenerate intervals, we conclude that $F = \mathcal{F}$ over this set. □

Theorem 3.1 $\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$

Proof. Let $y_1, y_2, ..., y_n$ be any legal assignment of values to the variables $x_1, x_2, ..., x_n$. For each y_i, $y_i = \hat{y}_i = [y_i, y_i] \in \hat{x}_i = [min_i, max_i]$; therefore, $\hat{y}_i \subseteq \hat{x}_i$. Hence, by the principle of monotonic inclusion, $\mathcal{F}(\hat{y}_1, \hat{y}_2, ..., \hat{y}_n) \subseteq \mathcal{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$. Applying

Lemma 3.1, we note that $F(y_1, y_2, ..., y_n) = \mathcal{F}(\hat{y}_1, \hat{y}_2, ..., \hat{y}_n)$. Since our choice of values was not special, we conclude that $F(y_1, y_2, ..., y_n) \subseteq \mathcal{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$ for all legal assignments $y_1, y_2, ..., y_n$. Since $\hat{F}$ is just the union of the values $F(y_1, y_2, ..., y_n)$, we know that $\hat{F} \subseteq \mathcal{F}$. □

According to the above theorem, we can approximate the United Extension of our delay function by replacing the discrete mathematical operators in the function with the corresponding interval ones. This substitution yields an interval version of the algorithm that computes bounds on the delay of the circuit much more efficiently than exhaustive enumeration.

3.2.2 Improving these Bounds

Up to this point, our discussion has been general and our proofs apply to any real-valued, arithmetic function F. Now, we discuss a particular example: delay equation 3.1. Let D denote this function, let $\hat{D}$ represent its united extension, and let $\mathcal{D}$ be the corresponding interval delay function. While $\mathcal{D}$ does indeed yield correct bounds on the delay of the circuit, they tend to be overly conservative. One reason for this is the lack of multiplicative inverses in the interval algebra. For example, let $\hat{x} = [a, b]$, then $\hat{x} \oslash \hat{x} = [a/b, b/a]$. This yields the expected result $[1, 1]$ only when $a = b$; however, $[1, 1] \subseteq \hat{x} \oslash \hat{x}$ for all $\hat{x}$. Crystal's delay equation contains several such instances. One occurs when we compute the RC contribution of transistor j as in the following:

$$\begin{aligned} C_j \odot R_j &= CperArea_j \odot Area_j \odot RperSquare_j \odot Aspect_j \\ &= CperArea_j \odot L_j \odot W_j \odot RperSquare_j \odot L_j \oslash W_j \\ &= CperArea_j \odot RperSquare_j \odot L_j^2 \odot W_j \oslash W_j \\ &\supseteq CperArea_j \odot RperSquare_j \odot L_j^2 \end{aligned}$$

To alleviate this problem, we must carefully tune the delay equations to eliminate any unnecessary divisions. Specifically, we rewrite function D to create an equivalent version D_2 by expanding the RC terms and eliminating as many of these divisions as possible. For example, the RC terms for transistor j are transformed as follows:

$$\begin{aligned} C_j(R_1 + ... + R_j) &= C_j(R_1 + ... + R_{j-1}) + C_j \cdot R_j \\ &= C_j(R_1 + ... + R_{j-1}) + CperArea_j \cdot RperSquare_j \cdot L_j^2 \end{aligned}$$

Similarly, the RC terms for node k are rewritten as follows:

$$\begin{aligned}
C_k(R_1 + ... + R_k) &= C_k(R_1 + ... + R_{k-2}) + C_k \cdot R_{k-1} + C_k \cdot R_k \\
&= C_k(R_1 + ... + R_{k-2}) + C_k \cdot R_k + \\
&\quad (CRest_k + CperWidth_{k-1} \cdot W_{k-1})R_{k-1} \\
&= C_k(R_1 + ... + R_{k-2}) + CRest_k \cdot R_{k-1} + C_k \cdot R_k + \\
&\quad CperWidth_{k-1} \cdot W_{k-1} \cdot R_{k-1} \\
&= C_k(R_1 + ... + R_{k-2}) + CRest_k \cdot R_{k-1} + C_k \cdot R_k + \\
&\quad CperWidth_{k-1} \cdot W_{k-1} \cdot RperSquare_{k-1} \cdot Aspect_{k-1} \\
&= C_k(R_1 + ... + R_{k-2}) + CRest_k \cdot R_{k-1} + C_k \cdot R_k + \\
&\quad CperWidth_{k-1} \cdot RperSquare_{k-1} \cdot L_{k-1}
\end{aligned}$$

where $C_k = CRest_k + CperWidth_{k-1} \cdot W_{k-1}$. In other words, $CperWidth_{k-1} \cdot W_{k-1}$ is the portion of node k's capacitance contributed by transistor $k-1$ and $CRest_k$ is the remainder.

Once we have D_2, we can replace the arithmetic operators in it with the corresponding interval operators to create an interval version, $\mathcal{D}_2$. Although the delay functions D and D_2 are equivalent, their interval counterparts $\mathcal{D}$ and $\mathcal{D}_2$ are not. Since $\mathcal{D}_2$ contains fewer interval divisions, the bounds that it produces are always as good as those produced by $\mathcal{D}$. Often they are tighter. In our next theorem, we show that $\mathcal{D}_2$ approximates $\hat{D}$, producing bounds that are at least as good as $\mathcal{D}$.

Theorem 3.2 $\hat{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}_2(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$

Proof. We begin by proving that $\hat{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}_2(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$. Applying Theorem 3.1, we note that $\hat{D}_2 \subseteq \mathcal{D}_2$. But $D = D_2$; therefore, $\hat{D} \subseteq \mathcal{D}_2$. Now we must show that $\mathcal{D}_2(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$. Since all our variables (capacitance factors, resistance factors, transistor lengths and widths) are positive real numbers, the special case of the distributive property holds. Specifically,

$$\begin{aligned}
C_k \odot (R_1 \oplus ... \oplus R_k) &= C_k \odot (R_1 \oplus ... \oplus R_{k-1}) \oplus C_k \odot R_k \\
&= C_k \odot (R_1 \oplus ... \oplus R_{k-2}) \oplus C_k \odot R_{k-1} \oplus C_k \odot R_k.
\end{aligned}$$

Hence $\mathcal{D}$ only differs from $\mathcal{D}_2$ in the simplified RC terms. We noted above that $[1,1] \subseteq \hat{x} \oslash \hat{x}$; therefore, by monotonic inclusion,

$$CperArea_k \odot RperSquare_k \odot L_k^2 \subseteq CperArea_k \odot Area_k \odot RperSquare_k \odot Aspect_k$$

$$CperWidth_k \odot RperSquare_k \odot L_k \subseteq CperWidth_k \odot W_k \odot RperSquare_k \odot Aspect_k$$

Applying the principle of monotonic inclusion again, we conclude that $\mathcal{D}_2 \subseteq \mathcal{D}$. □

3.3 Performance

Using the revised interval delay function $\mathcal{D}_2$, we have implemented an interval version (Intcryst) of the RC analysis algorithm. We also implemented a Monte Carlo version of D for comparison. To select values from the range of each variable, the Monte Carlo program employs a random number generator with a uniform distribution. It then uses the original delay function D to compute the delay for each trial and returns the minimum, maximum, and average delays calculated. Both implementations were written in the C programming language and run on a SUN SPARCstation. We now present an analysis of the performance of our interval algorithm as compared to Monte Carlo simulation.

3.3.1 Quality of the Bounds

To determine the quality of the bounds produced by both the Monte Carlo (50,000 trials) and interval methods, we conducted a number of experiments comparing their bounds to the theoretical bounds ($\hat{D}$) for the given circuit. For these experiments, the theoretical bounds were approximated by running a Monte Carlo simulation with an unlimited number of trials. We also tested nominal delay (i.e., the delay computed using the average values of the input parameters). For all methods, delay was calculated in two stages. First, we determined the resistance and capacitance factors for all transistors by table lookup. Then we used the values obtained to compute delay.

The first experiment illustrates the relationship between delay and stage length (i.e., the number of transistors in a stage) for a family of nMOS circuits. As shown in Figure 3.2, the Monte Carlo and nominal delay methods always underestimate the theoretical bounds, while the interval approach always overestimates them. The graph in Figure 3.3 depicts the average percent error between the theoretical delay bounds and the bounds generated using the other methods (i.e., error = theoretical upper (lower) bound - estimated upper (lower) bound/ theoretical upper (lower) bound). As shown in Figure 3.3, both the Monte Carlo and interval methods produce bounds with similar accuracy, coming within 10% of the theoretical bounds for all stage lengths tested. The

Stage Length	Intcryst		Monte Carlo		Nominal	
	lower	upper	lower	upper	lower	upper
1	0	0	-4.42	3.91	-18.98	16.08
3	0.26	-0.24	-4.87	4.67	-16.23	13.49

Table 3.1: Average % Error for Upper and Lower Delay Bounds

graph further shows that the error of both methods grows with the length of the stage. Fortunately, stages encountered in most circuits are typically small, containing fewer than five transistors. As the graph demonstrates, nominal delay, the method currently employed in most timing analyzers, provides the poorest estimate of worst-case timing behaviors.

We also analyzed a portion of the Brown Systolic Array (B-SYS)[Hugh89], a 2-μ CMOS design. In particular, we surveyed 33 stages extracted from the ALU of the chip. The stages ranged in length from one to three transistors, with an average length of 2.45 transistors. The average percent error for each method is displayed in Table 3.1. As shown, Intcryst produced the tightest bounds, coming within 0.3% of the theoretical bounds on average. The Monte Carlo method (10,000 trials) produced an average error of about 4.6%, while the nominal delay had an average error of 15.6%.

The third example is an actual critical path computation taken from the B-SYS chip. In this case, we analyzed a portion of the circuit containing approximately 2,000 transistors. The critical path returned was determined to have 9 stages. Table 3.2 displays the delay bounds computed for each stage and the total for the critical path. Table 3.3 gives the percent error in stage delay for each method. As shown, the results are similar to those obtained in the previous experiment.

3.3.2 Running Time of the Algorithm

We used two measures to compare the speeds of the various implementations. First, we counted the occurrences of each arithmetic operation in each implementation. Table 3.4 contains a breakdown of operations for each of the three functions tested – the original delay function D used by Crystal and the Monte Carlo program, the revised delay function D_2, and the interval version, $\mathcal{D}_2$. In the analysis, N represents the number of transistors in a stage. The table shows that the interval version of the PR-Slope model

Stage	Size	Theoretical		Intcryst		Monte Carlo		Nom
		min	max	min	max	min	max	
1	2	6.8	9.3	6.8	9.3	7.2	8.7	8.0
2	3	54.3	72.9	54.1	73.1	56.6	69.5	63.0
3	2	10.3	14.1	10.3	14.1	10.8	13.4	12.1
4	2	10.6	14.4	10.6	14.4	11.0	13.9	12.4
5	2	1.2	1.6	1.2	1.6	1.2	1.5	1.4
6	1	0.7	1.0	0.7	1.0	0.8	0.9	0.8
7	1	0.8	1.1	0.8	1.1	0.8	1.0	0.9
8	3	20.4	27.4	20.3	27.5	21.2	26.5	23.7
9	1	2.0	2.7	2.0	2.7	2.0	2.6	2.3
Path total		107.1	144.5	106.8	144.8	111.7	138.1	124.5

Table 3.2: B-SYS Critical Path Computation

Stage	Size	Intcryst		Monte		Nominal	
		min	max	min	max	min	max
1	2	0.3	-0.3	-5.9	5.9	-16.7	14.0
2	3	0.3	-0.3	-4.2	4.7	-16.0	13.6
3	2	0.1	-0.1	-4.8	5.3	-17.0	14.3
4	2	0	-0.1	-3.8	3.7	-16.8	14.1
5	2	0.9	-0.6	-5.2	5.7	-18.1	15.4
6	1	0	0	-7.1	5.1	-18.6	15.3
7	1	0	0	-3.9	4.6	-19.5	15.6
8	3	0.4	-0.4	-4.1	3.1	-16.0	13.6
9	1	0	0	-1.5	1.5	-16.2	13.6
Path total		0.2	-0.3	-4.3	4.4	-16.3	13.8

Table 3.3: % Error in Critical Path Delay

Algorithm	Number of Operations			
	+	−	·	/
D (original)	$10N+1$	4	$9N+3$	$N+2$
D_2 (revised)	$18N-2$	4	$23N-10$	$2N+3$
$\mathcal{D}_2$ (intcryst)	$36N-4$	8	$46N-20$	$4N+6$

Table 3.4: Worst-Case Operations Count for a Single Stage with N Transistors

contains approximately four times as many operations in each category as the original delay function; therefore, we expect that this algorithm will require roughly four times longer to compute the stage delay.

To confirm this hypothesis, we ran all three versions on identical circuits and recorded their response times. Our experiments support this conjecture, showing that Intcryst is roughly four times slower than Crystal. Both implementations have $O(N)$ time complexity.

Since the Monte Carlo program uses the original function D, we note that our interval algorithm computes delay bounds for a stage in about the time required to perform four Monte Carlo trials. Our experiments, however, show that 50,000 Monte Carlo trials are often required to produce comparable delay bounds. Our interval algorithm, therefore, offers a vast improvement in speed. To illustrate this, Table 3.5 displays the amount of time each program took to compute delay bounds for the critical path computation presented in Table 3.2. To locate this critical path, Crystal examined almost 1,300 stages. It was able to compute the delay of all these stages in about 0.7 seconds. Intcryst required 2.8 seconds to compute delay bounds for these stages. The Monte Carlo program with 1,000 trials required 12 minutes for its analysis and with 50,000 trials took 10 hours to compute bounds.

3.4 Summary

In this chapter, we presented a method for modeling the effects of uncertainty in RC analysis. Representing uncertain parameters as intervals, we used interval algebra to create a mathematical framework for manipulating these uncertain values. We then modified an existing RC analysis algorithm (Crystal's PR-Slope model) to test our approach. Although Crystal was chosen for our experiments, the same techniques may

Method	No. of trials	Time
Crystal	1	0.7 sec
Intcryst	1	2.8 sec
Monte Carlo	1000	12.0 min
Monte Carlo	5000	1 hr
Monte Carlo	10000	2 hrs
Monte Carlo	50000	10 hrs

Table 3.5: Relative Speed of Each Algorithm for a Critical Path Computation

be applied to other timing analysis algorithms.

Given a transistor network with uncertain input parameters, our interval algorithm computes bounds on the delay. We claimed that the delay estimates produced by our algorithm approximate the theoretical delay bounds for the circuit and provided proofs to support our hypothesis. We then implemented our algorithm and tested its performance. Analysis shows that its operating speed is not significantly slower than that of the non-interval RC analysis algorithm.

When compared to Monte Carlo simulation, our algorithm is much more efficient, operating several orders of magnitude faster for the same quality results. For stages with few transistors, the accuracy of both methods is similar: both come within 10% of the theoretical bounds in the worst-case. In the next chapter, we present other interval-based methods that further improve the accuracy of these bounds.

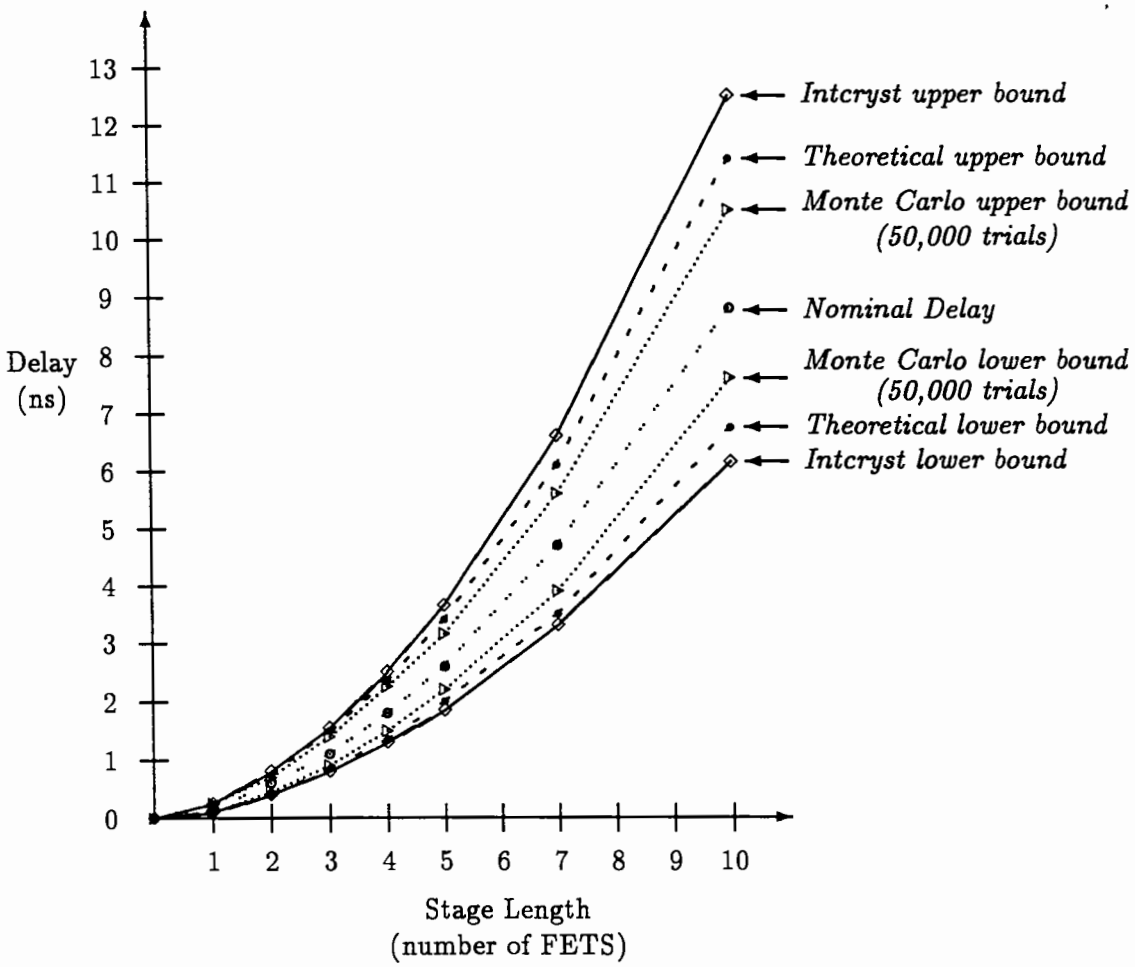


Figure 3.2: Delay vs. Stage Length

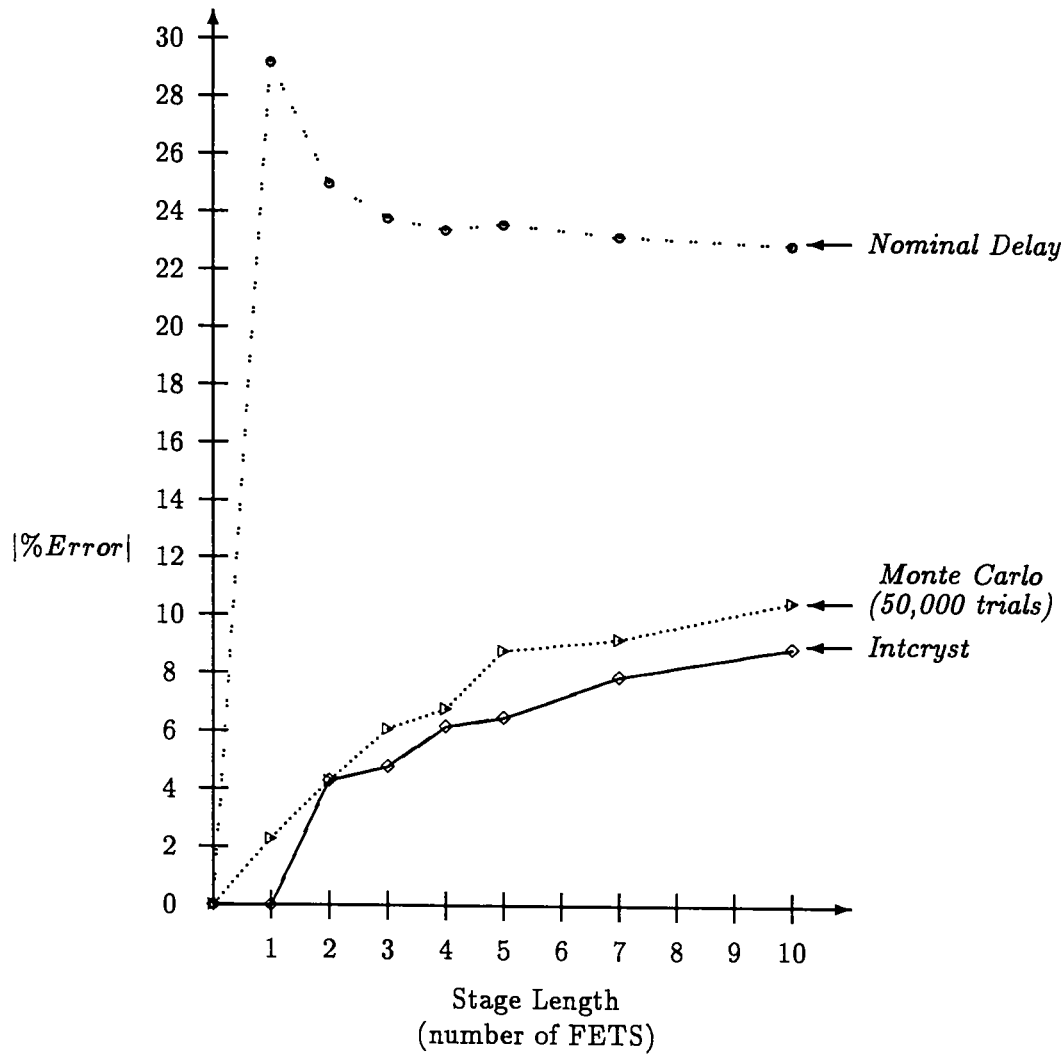


Figure 3.3: % Error vs. Stage Length

Chapter 4

Modeling Uncertainty in RC Timing Analysis II

In the previous chapter, we presented a method based on interval arithmetic for modeling the effects of uncertainty in RC analysis. Using this technique, we implemented Intcryst, an interval version of Crystal's PR-Slope model [Oust83, Oust84, Oust86], and compared the bounds that it generated to the theoretical bounds ($\hat{D}$) for several circuits. While the interval results always contain the theoretical bounds, they are often overly conservative. The reason for this is that the interval algebra ignores variable interdependencies. Each occurrence of a variable is considered independent of all other occurrences. For example, consider the function $F(x) = x + 1/x$ where $\hat{x} = [0.5, 2]$. The United Extension $\hat{F}$ of this function is $[2, 2.5]$, but the interval solution is $[1, 4]$. The interval upper bound of four results from using the maximum value of x for the first occurrence of the variable and its minimum value for the second (i.e., $2 + 1/0.5 = 4$). If we wish to produce tighter delay bounds, we must develop alternative methods that account for variable interdependence.

In this chapter, we explore two other *general* interval-based techniques for approximating the United Extension of a function. The first of these methods employs centred forms and the second uses interval differential calculus, an interval extension of real differential calculus. To illustrate these methods, we incorporate them into Crystal's PR-slope model to create two new algorithms for computing bounds on circuit delay. Although Crystal was chosen for these experiments, the same techniques can be applied to other timing verifiers. Finally, we analyze the performance of these methods,

comparing them to our original interval algorithm and with Monte Carlo simulation.

4.1 Krawczyk's Centred Form

Centred forms provide an alternative method for approximating the United Extension, $\hat{F}$. While there are many types of centred forms (see [Rats84]), all share a common framework and structure. Given a real-valued function $F(x_1, x_2, ..., x_n)$ and the interval range $\hat{x}_i$ of each variable x_i, all methods begin by choosing a particular set of values, $c_1, c_2, ..., c_n$, such that each $c_i \in \hat{x}_i$ and evaluating $F(c_1, c_2, ..., c_n)$. While any choice $c_i \in \hat{x}_i$ can be used, the midpoint of each interval is typically chosen. They then rewrite the function F to create an equivalent expression (denoted F_C) which has the following structure:

$$F_C(x_1, x_2, ..., x_n) \equiv F(c_1, c_2, ..., c_n) + G(x_1 - c_1, x_2 - c_2, ..., x_n - c_n) \qquad (4.1)$$

This new representation is known as the *General Centred-Form of F.*

Centred forms differ from one another in their choice of the function G. For instance, *standard centred forms* use a Taylor series expansion of order k to create G. The *mean-value centred form* uses derivatives to create G; thus, $G \equiv F'(x) \cdot (x - c)$. *Krawczyk's centred form* uses interval slopes. With this method, there is no general closed form for G; instead G is computed directly from F by construction.

Having transformed F into centred form F_C, they then create an interval version of F_C. This is accomplished by replacing each variable x_i in F_C by its interval range $\hat{x}_i$ and each arithmetic operator by its corresponding interval operator. When evaluated, this new interval function produces symmetric bounds centered at $F(c_1, c_2, ..., c_n)$ containing $\hat{F}$. Although function F and its centred form F_C are equivalent, their interval extensions $\mathcal{F}$ and $\mathcal{F}_C$ are not. Since the resulting interval versions contain different numbers and combinations of the arithmetic operations, the accumulation of error is different. In most cases, the centred forms produce tighter bounds.

Each of the centred forms presented above yield bounds which subsume the true bounds $\hat{F}$; thus, $\hat{F} \subseteq \mathcal{F}_C$. Proofs of convergence for all of these centred methods are found in [Rats84] and will not be repeated here.

In choosing a centred form for RC timing analysis, we evaluated each of the methods presented in [Rats84]. With standard centred forms, we would have to precompute a different closed form function G for each function F. In RC analysis, the delay equation

varies with the number of transistors in the stage (called the *stage length*); therefore, to use a standard centred form, we must generate a closed centred expression for each possible stage length encountered. We require a more flexible approach. Mean-value centred forms are one alternative. Ratschek and Rokne, however, claim that this method produces poorer bounds estimates than the higher-order standard centred forms [Rats84, pp. 78-81]. Unlike the standard forms, Krawczyk's centred form does not have to be precomputed and can be easily programmed for any function. The flexibility of this format makes it an ideal candidate for the timing analysis problem.

Krawczyk's centred form [Rats84, pp. 59-62] calculates G directly from function F by construction. We therefore require a straight-line program for computing F. Let $B = \{b_1, b_2, ..., b_r\}$ be the set of real constants that appear in F, let $X = (x_1, x_2, ..., x_n)$ be the vector of independent variables in F, and let $U = \{u_1, u_2, ..., u_s\}$ be a finite set dependent variables used to compute F. A straight-line program is a sequence of s computational steps for calculating the value of a function and is defined as follows:

$u_i = x_i$	for $i = 1, ..., n$	/* load variables */
$u_i = b_{i-n}$	for $i = n+1, ..., n+r$	/* load constants */
$u_i = u_j *_i u_k$	for $i = n+r+1, ..., s$	/* calculate F */
	where $j, k < i$ and $*_i \in \{+, -, \cdot, /\}$	

For example, if $F(x) = x + 1/x$, then the following sequence of instructions constitutes one possible program for computing F:

$$\begin{aligned} u_1 &= x \\ u_2 &= 1 \\ u_3 &= u_2/u_1 \\ u_4 &= u_1 + u_3 \end{aligned}$$

Given a straight-line program for computing F, we employ it to compute the interval slope G. Let F_i denote the function corresponding to the ith step of the straight line program (e.g., $F_1 = x$, $F_2 = 1$, $F_3 = 1/x$, and $F_4 = x + 1/x$). Likewise, let $\mathcal{F}_i$ denote the interval extension of F_i (e.g., $\mathcal{F}_1 = \hat{x}$, $\mathcal{F}_2 = [1,1]$, $\mathcal{F}_3 = [1,1] \oslash \hat{x}$, and $\mathcal{F}_4 = \hat{x} \oplus [1,1] \oslash \hat{x}$). Define $E_1, E_2, ..., E_n \in I^n$ to be n-dimensional interval unit vectors such that $E_1 = ([1,1],[0,0],...,[0,0])$, $E_2 = ([0,0],[1,1],...,[0,0])$, ..., $E_n = ([0,0],...,[0,0],[1,1])$. Likewise, let $O = ([0,0],[0,0],...,[0,0]) \in I^n$ be an n-dimensional

i	u_i	$F_i(c)$	$\mathcal{F}_i(\hat{x})$	G_i
1	x	1.25	[0.5,2]	[1,1]
2	1	1	[1,1]	[0,0]
3	u_2/u_1	0.8	[0.5,2]	[-1.6,-0.4]
4	$u_1 + u_3$	**2.05**	**[1,4]**	**[-0.6,0.6]**

Table 4.1: Computing the Interval Slope G

interval zero vector. The interval slope $G = (\hat{g}_1, \hat{g}_2, ..., \hat{g}_n)$ is defined recursively from the s-instruction straight-line program as follows:

$G_i = E_i$	if $u_i = x_i$	/* load variables */
$G_i = O$	if $u_i = b_{i-n}$	/* load constants */
$G_i = G_j \oplus G_k$	if $u_i = u_j + u_k$	/* addition */
$G_i = G_j \ominus G_k$	if $u_i = u_j - u_k$	/* subtraction */
$G_i = G_j \odot \mathcal{F}_k(\hat{X}) \oplus G_k \odot F_j(C)$	if $u_i = u_j u_k$	/* multiplication */
$G_i = G_j \oslash \mathcal{F}_k(\hat{X}) \ominus G_k \odot F_i(C) \oslash \mathcal{F}_k(\hat{X})$	if $u_i = u_j/u_k$	/* division */

The recursion terminates after s steps, the number of instructions in the program, and the interval slope $G = G_s$. For example, let $F(x) = x + 1/x$, $\hat{x} = [0.5, 2]$, and $c = 1.25$. Using the function procedure given above, we can derive the interval slope G for this function. The results of each computational step are shown in Table 4.1. The last row of the table displays the resulting interval slope.

Having found G, we can construct Krawczyk's centred form $\mathcal{F}_C$ for function F using the following formula:

$$\mathcal{F}_C(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \equiv F(c_1, c_2, ..., c_n) \oplus \sum_{i=1}^{n} \hat{g}_i \odot (\hat{x}_i \ominus c_i). \tag{4.2}$$

Using the results in Table 4.1, this equation yields $\mathcal{F}_C = [1.6, 2.5]$. This interval is centered at $F(c) = 2.05$ and represents the smallest such interval containing the theoretical bounds $[2, 2.5]$. Recall that the interval bounds $\mathcal{F}$ for the same function are $[1, 4]$. In this case, the centred form produces tighter bounds.

This algorithm has a time complexity of $O(sn)$ where s represents the number of instructions in the program for computing F and n represents the number of independent variables.

4.2 Interval Differential Calculus

4.2.1 Rall's Algorithm

The second method for approximating $\hat{F}$ employs interval differential calculus, which was introduced by Rall [Rall86]. Given a function $F(x)$ over $\Re$ and an interval $\hat{x} = [a, b]$, Rall uses differential calculus to compute $\hat{F}$. The derivative F' provides information about the monotonicity of the function which in turn can be used to obtain a better approximation to $\hat{F}$. For instance, if $F'(x) \geq 0$ for all $x \in \hat{x}$, then function F is monotone increasing over the interval $\hat{x}$ and

$$\hat{F}(\hat{x}) = [F(a), F(b)]. \tag{4.3}$$

Likewise, if $F'(x) \leq 0$ for all $x \in \hat{x}$, then the function is monotone decreasing over the interval $\hat{x}$ and

$$\hat{F}(\hat{x}) = [F(b), F(a)]. \tag{4.4}$$

Rall further demonstrates that $F'(x)$ for all $x \in \hat{x}$ can be approximated using interval differential arithmetic, a natural extension of real differential arithmetic.

The operands for real differential arithmetic are ordered pairs in $\Re^2$ such that the first element of the pair contains a variable or function and the second element its derivative. If $X = (x, x')$ and $Y = (y, y')$ are two such operands, then the rules of real differential arithmetic are:

$$X + Y = (x, x') + (y, y') = (x + y, x' + y') \tag{4.5}$$

$$X - Y = (x, x') - (y, y') = (x - y, x' - y') \tag{4.6}$$

$$X \cdot Y = (x, x') \cdot (y, y') = (x \cdot y, x \cdot y' + y \cdot x') \tag{4.7}$$

$$X/Y = (x, x')/(y, y') = (x/y, (x' - (x/y) \cdot y')/y), \quad y \neq 0 \tag{4.8}$$

The interval differential algebra is similar except that all operands are interval pairs and the basic arithmetic operators $+$, $-$, $\cdot$, and $/$ have been replaced by their interval equivalents $\oplus$, $\ominus$, $\odot$, and $\oslash$. If $\hat{X} = (\hat{x}, \hat{x}')$ and $\hat{Y} = (\hat{y}, \hat{y}')$, then the interval differential operators are:

$$\hat{X} \oplus \hat{Y} = (\hat{x}, \hat{x}') \oplus (\hat{y}, \hat{y}') = (\hat{x} \oplus \hat{y}, \hat{x}' \oplus \hat{y}') \tag{4.9}$$

$$\hat{X} \ominus \hat{Y} = (\hat{x}, \hat{x}') \ominus (\hat{y}, \hat{y}') = (\hat{x} \ominus \hat{y}, \hat{x}' \ominus \hat{y}') \tag{4.10}$$

$$\hat{X} \odot \hat{Y} = (\hat{x}, \hat{x}') \odot (\hat{y}, \hat{y}') = (\hat{x} \odot \hat{y}, \hat{x} \odot \hat{y}' \oplus \hat{y} \odot \hat{x}') \tag{4.11}$$

```
Current bounds = F((a + b)/2)
Add interval x̂ = [a, b] to evaluation queue
While queue not empty
    Get next (sub)interval ŷ from queue
    Compute F'(ŷ)
    Check range of F'(ŷ)
    If monotone
        Compute exact (sub)interval bounds using Equation 4.3 or 4.4
        Update current bounds
    Else if possible, bisect ŷ into halves ŷ1 and ŷ2
        Add intervals ŷ1 and ŷ2 to evaluation queue
    Else (interval ŷ is too small to bisect)
        Approximate (sub)interval bounds using F(ŷ)
        Update current bounds
Return current bounds
```

Figure 4.1: Rall's Interval Differential Algorithm

$$\hat{X} \oslash \hat{Y} = (\hat{x}, \hat{x}') \oslash (\hat{y}, \hat{y}') = (\hat{x} \oslash \hat{y}, (\hat{x}' \ominus (\hat{x} \oslash \hat{y}) \odot \hat{y}') \oslash \hat{y}) \qquad (4.12)$$
$$\textit{where } [0,0] \notin \hat{y}$$

We use these interval differential operators to compute the interval derivative of F, $\mathcal{F}'(\hat{x})$.

Having introduced interval differential calculus, Rall incorporates it into his algorithm to estimate $\hat{F}(\hat{x})$ as summarized in Figure 4.1. We begin by computing the interval derivative of function F. If this derivative indicates that the function is monotone over the interval $\hat{x}$, then we compute $\hat{F}(\hat{x})$ directly using Equation 4.3 or 4.4; otherwise, we partition the interval $\hat{x}$ into two halves and repeat the analysis on each half. The number of interval bisections performed by this algorithm is specified by the user. When this number is exceeded, the bounds must be estimated. In this case, Rall uses $\mathcal{F}$ to estimate subinterval bounds. The final bounds on the function are found by taking the union of all subinterval bounds. The number of estimated subinterval components indicates the quality of the final result. In particular, the fewer the number of non-monotone subintervals, the better the resulting bounds are likely to be.

To illustrate this approach, consider our previous example $F(x) = x + 1/x$ with $\hat{x} = [0.5, 2.0]$ and a bisection limit of 3. The results of each stage of this computation are displayed in Table 4.2. The last column in the table shows the subinterval bounds (if any) computed for each stage. Taking the union of these bounds yields a final result

$\hat{x}$	$\mathcal{F}(\hat{x})$	$\mathcal{F}'(\hat{x})$	Action	$\mathcal{F}_I(\hat{x})$
0.50, 2.00	1.00,4.00	-3.00,0.75	bisect	
0.50, 1.25	1.30,3.25	-3.00,0.36	bisect	
1.25, 2.00	1.75,2.80	0.36,0.75	compute bounds	2.05,2.50
0.50, 0.88	1.64,2.88	-3.00,-0.31	compute bounds	2.02,2.50
0.88, 1.25	1.67,2.39	-0.31,0.36	bisect	
0.88, 1.06	1.82,2.21	-0.31,0.11	estimate bounds	1.82,2.21
1.06, 1.25	1.86,2.19	0.11,0.36	compute bounds	2.00,2.05
TOTAL				**1.82,2.50**

Table 4.2: An Interval Differential Bounds Computation

of $[1.82, 2.50]$. The theoretical bounds for this function are $[2, 2.5]$ and the interval bounds are $[1, 4]$. For this function, the interval differential method $\mathcal{F}_I(\hat{x})$ produced much better results than the original interval approach.

4.2.2 Extensions to Rall's Algorithm

Rall's algorithm only works for single-variable functions, so we have generalized it to create one for multi-variable functions. Given a function $F(x_1, x_2, ..., x_n)$ over $\Re$ and the range of values for each variable $x_i \in \hat{x}_i = [min_i, max_i]$, we wish to approximate $\hat{F}$ using interval differential calculus. For multi-variable functions, we use *partial* derivatives. Let $L = (l_1, l_2, ..., l_n)$ and $U = (u_1, u_2, ..., u_n)$ be lower and upper bound vectors over $\Re^n$. If $\partial F/\partial x_i \geq 0$ for all legal value assignments, then F is monotone increasing with respect to x_i and

$$\begin{aligned} l_i &= min_i \\ u_i &= max_i. \end{aligned} \tag{4.13}$$

Likewise, if $\partial F/\partial x_i \leq 0$ for all legal value assignments, then F is monotone decreasing with respect to x_i and

$$\begin{aligned} l_i &= max_i \\ u_i &= min_i. \end{aligned} \tag{4.14}$$

If the function F is monotone over the ranges of all variables $x_1, x_2, ..., x_n$, then we can precisely compute $\hat{F}$ as follows:

$$\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) = [F(l_1, l_2, ..., l_n), F(u_1, u_2, ..., u_n)] \tag{4.15}$$

We can approximate each partial derivative $\partial F/\partial x_i$ evaluated at all possible assignments $x_j \in \hat{x}_j$ with an interval partial derivative, $\partial \mathcal{F}/\partial x_i$. The interval partial differential algebra needed for this computation is similar to the algebra presented in the previous section. As before, the operands are variable-derivative pairs, but in this case, the derivative is represented as a vector of n partial derivatives (e.g., $\hat{X} = (\hat{x}, (\hat{x}'_1, \hat{x}'_2, ..., \hat{x}'_n))$ where $\hat{x}'_i$ denotes the partial derivative of the i^{th} variable with respect to x). The arithmetic operators for this calculus are defined as follows:

$$\hat{X} \oplus \hat{Y} = (\hat{x} \oplus \hat{y}, (\hat{x}'_1 \oplus \hat{y}'_1, ..., \hat{x}'_n \oplus \hat{y}'_n)) \tag{4.16}$$

$$\hat{X} \ominus \hat{Y} = (\hat{x} \ominus \hat{y}, (\hat{x}'_1 \ominus \hat{y}'_1, ..., \hat{x}'_n \ominus \hat{y}'_n)) \tag{4.17}$$

$$\hat{X} \odot \hat{Y} = (\hat{x} \odot \hat{y}, (\hat{x} \odot \hat{y}'_1 \ \oplus \ \hat{y} \odot \hat{x}'_1, ..., \hat{x} \odot \hat{y}'_n \ \oplus \ \hat{y} \odot \hat{x}'_n)) \tag{4.18}$$

$$\hat{X} \oslash \hat{Y} = (\hat{x} \oslash \hat{y}, ((\hat{x}'_1 \ \ominus \ (\hat{x} \oslash \hat{y}) \odot \hat{y}'_1) \oslash \hat{y}, ..., (\hat{x}'_n \ \ominus \ (\hat{x} \oslash \hat{y}) \odot \hat{y}'_n) \oslash \hat{y})) \quad \textit{where} \ [0,0] \notin \hat{y} \tag{4.19}$$

Now when F is evaluated using these interval differential operations, the result is the operand $(\mathcal{F}, (\partial \mathcal{F}/\partial x_1, \partial \mathcal{F}/\partial x_2, ..., \partial \mathcal{F}/\partial x_n))$. The C language code for each of these operators in presented in Appendix C.

We contend that these interval partial derivatives may be used to determine the monotonicity of F. Recall that F is monotone increasing (decreasing) with respect to the variable x_i, if $\partial F/\partial x_i \geq 0$ (≤ 0) for all possible assignments $x_j \in \hat{x}_j$. By definition, $\partial F/\partial x_i$ evaluated at all possible variable values is just the United Extension of $\partial F/\partial x_i$. Furthermore, Theorem 3.1 states that the interval extension of a function produces bounds which contain its United Extension; therefore,

$$\hat{\partial F/\partial x_i}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \ \subseteq \ \partial \mathcal{F}/\partial x_i(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n). \tag{4.20}$$

Hence, if $\partial \mathcal{F}/\partial x_i \geq 0$, then $\hat{\partial F/\partial x_i} \geq 0$ and F is monotone increasing with respect to x_i. We thus conclude that the interval partial derivatives computed by our algebra may be employed to determine the monotonicity of the function F.

Having defined the interval partial differential algebra, we are now ready to present our version of the bounds algorithm (See Figure 4.2). The algorithm begins by computing interval partial derivatives for each variable. It then checks the ranges of all partial derivatives for monotonicity, loading the lower L and upper U bounds arrays with the appropriate values. If the function is monotone with respect to all variables, then the true bounds on the function can be precisely found. If, however, the function

```
Current bounds = F(c_1, ..., c_n) where c_i = (min_i + max_i)/2
Add interval vector X̂ = (x̂_1, x̂_2, ..., x̂_n) to evaluation queue
While queue not empty
    Get next entry Ŷ = (ŷ_1, ..., ŷ_n) from queue
    Compute ∂F/∂y_i for all i = 1, ..., n
    Check ranges of all partial derivatives, loading bound arrays L and U
    If F is monotone for all y_i
        Compute exact (sub)interval bounds using Equation 4.15
        Update current bounds
    Else if possible, bisect Ŷ into Ŷ_1 and Ŷ_2
        Add vectors Ŷ_1 and Ŷ_2 to evaluation queue
    Else (intervals are too small to bisect)
        Approximate (sub)interval bounds using F(Ŷ)
        Update current bounds
Return current bounds
```

Figure 4.2: Multi-Variable Interval Differential Algorithm

is non-monotone over the range of some variable, we bisect the intervals and repeat the analysis on the pieces. As before, $\mathcal{F}$ is used to estimate the bounds when the subintervals become too small.

The partitioning method employed in this algorithm is crucial to both its efficiency and correctness. If we subdivide and carelessly group the intervals, we risk underestimating the true bounds. On the other hand, if we are too conservative (e.g., calculate all combinations of subintervals), the problem becomes computationally intractable. We therefore seek a balance between these two extremes. Our approach is to bisect only the non-monotone intervals and group the resulting subintervals such that all the values that are likely to produce upper bounds on the solution are collected together and all the values that are likely to produce lower bounds are similarly collected. More formally, if non-monotone interval $\hat{x}_i = [min_i, max_i]$, then let $\hat{x}_{i,1} = [min_i, mid_i]$ and $\hat{x}_{i,2} = [mid_i, max_i]$ where $mid_i = (min_i + max_i)/2$. Also let $\hat{L}$ represent the set of subintervals $\hat{x}_{i,j}$ that are likely to produce lower bounds and $\hat{U}$ represent the set of subintervals that are likely to produce upper bounds. The goal is to produce a partitioning $(\hat{L}, \hat{U})$ of the problem space. The difficulty here is that F is non-monotone with respect to variable x_i and we do not know which values of x_i minimize or maximize F. Here again, we use the partial derivatives to make an educated guess. Specifically, the magnitude of the derivative indicates the slope of the curve, while its sign denotes

$\hat{x}_1$	$\hat{x}_2$	$\hat{x}_3$	$\partial\mathcal{F}/\partial x_1$	$\partial\mathcal{F}/\partial x_2$	$\partial\mathcal{F}/\partial x_3$	Action	$\mathcal{F}_I$
0.5, 1.5	1.0,2.0	4.0,5.0	-7.0,0.6	-4.3,1.0	0.5,1.0	bisect	
0.5, 1.0	1.0,1.5	4.0,5.0	-5.0,0.0	-4.0,0.2	0.7,1.0	bisect	
1.0, 1.5	1.5,2.0	4.0,5.0	-1.0,0.3	-1.6,0.0	0.5,0.7	bisect	
0.5, 1.0	1.0,1.25	4.0,5.0	-4.0,0.0	-4.0,-0.6	0.8,1.0	compute	5.45,7.50
0.5, 1.0	1.25,1.5	4.0,5.0	-5.0,-0.3	-2.2,0.2	0.7,0.8	bisect	
1.0, 1.25	1.5,2.0	4.0,5.0	-1.0,0.1	-1.4,0.0	0.5,0.7	bisect	
1.25, 1.5	1.5,2.0	4.0,5.0	-0.3,0.3	-1.6,-0.2	0.5,0.7	bisect	
0.5, 1.0	1.25,1.38	4.0,5.0	-4.5,-0.3	-2.2,-0.1	0.7,0.8	compute	5.28,7.00
0.5, 1.0	1.38,1.5	4.0,5.0	-5.0,-0.4	-1.6,0.2	0.7,0.7	estimate	4.54,7.64
1.0, 1.12	1.5,2.0	4.0,5.0	-1.0,-0.2	-1.3,0.0	0.5,0.7	compute	4.90,5.83
1.12, 1.25	1.5,2.0	4.0,5.0	-0.6,0.1	-1.4,-0.1	0.5,0.7	estimate	4.32,6.36
1.25, 1.38	1.5,2.0	4.0,5.0	-0.3,0.2	-1.5,-0.2	0.5,0.7	estimate	4.34,6.31
1.38, 1.5	1.5,2.0	4.0,5.0	-0.1,0.3	-1.6,-0.3	0.5,0.7	estimate	4.38,6.29
TOTAL							**4.32,7.64**

Table 4.3: A Multi-Variable Interval Differential Bounds Computation

the direction (whether the function is increasing or decreasing). For non-monotone x_i, its upper and lower derivative bounds specify the maximum increasing and decreasing slopes respectively. A large upper bound implies that the function rapidly increases over some values of $\hat{x}_i$. Likewise, a large lower bound implies that the function rapidly decreases over some values of $\hat{x}_i$. Therefore, if the magnitude of the upper bound is greater than that of the lower, we know that the function increases faster than it decreases and assign $\hat{x}_{i,1}$ to $\hat{L}$ and $\hat{x}_{i,2}$ to $\hat{U}$. If the magnitude of the lower bound is the greater, then we assume that the function decreases faster than it increases and assign $\hat{x}_{i,2}$ to $\hat{L}$ and $\hat{x}_{i,1}$ to $\hat{L}$. In practice, this partitioning method works very well, efficiently yielding tight bounds that approximate the theoretical solution.

To illustrate this algorithm, consider $F(x_1, x_2, x_3) = x_1 + x_2/x_1 + x_3/x_2$ where $\hat{x}_1 = [0.5, 1.5]$, $\hat{x}_2 = [1, 2]$, $\hat{x}_3 = [4, 5]$, and the bisection limit is 3. The results of each stage of the computation are displayed in Table 4.3. As the table shows, the interval differential bounds $\mathcal{F}_I$ for this function are $[4.32, 7.64]$. The theoretical bounds $\hat{F}$ are $[4.83, 7.50]$ and the interval bounds $\mathcal{F}$ are $[3.17, 10.50]$. Clearly, the interval differential approach produced a significantly better approximation to $\hat{F}$ than $\mathcal{F}$ did.

Because we partition and group non-monotonic variables, the outer loop of our interval differential algorithm (see Figure 4.2) is executed a maximum of $2^{b+1} - 1$ times, where b represents the maximum number of bisections allowed. Therefore, the worst-case computational complexity of this algorithm is $O(2^{b+1}sn)$, where b is the

bisection limit, s is the number of arithmetic operations in F, and n is the number of independent variables. Fortunately, the algorithm achieves fairly good results with small values of b. In this paper, all interval differential bounds were generated with $b = 3$.

4.3 Improved Delay Estimates for Timing Analysis

Having described two general interval techniques for approximating the United Extension, we now apply these methods to the timing analysis problem. As in the previous chapter, we have chosen Crystal's PR-slope model [Oust83, Oust84, Oust86].

4.3.1 The Centred Version

First, we produce a Krawczyk's centred-form version of the PR-slope delay modeler. For this problem, the function $D(x_1, x_2, ..., x_n)$ is the delay function given in Equation 3.1 and the variables $x_1, x_2, ..., x_n$ are the resistance factors, capacitance factors, and transistor sizes needed to compute the delay. The transistor resistance and capacitance factors are predetermined by table lookup.[1]

Computing the centred form of this delay function is accomplished in two phases. First, we calculate the interval slope $\hat{g}_i$ for each variable x_i, then we use these interval slopes to find bounds on the delay. To calculate the interval slope, we require a straight-line program for computing D. We therefore encode the delay equation within a computer program and employ it to compute the interval slope vector G. Once we have determined G, we use Equation 4.2 to calculate the final delay bounds.

4.3.2 The Interval Differential Version

To compute interval differential delay bounds, we must first create an interval differential version of the delay function D. This is accomplished by replacing each arithmetic operator $+$, $-$, $\cdot$, and $/$ in D by its interval differential equivalent specified in Equations 4.16 through 4.19 and each variable x_i in D by an interval operand $\hat{X}_i$ indicating its value and derivative. When evaluated, this new function returns the interval partial

[1]Because Crystal uses table lookup to determine the trigger transistor resistance factor, any dependency between this value and the other variables is hidden from the algorithm. Both the centred and interval differential methods model variable interdependence if all dependent values are explicitly expressed as functions of the independent variables. Because this is not the case in Crystal, we chose to precompute this factor and treat it as an independent variable for the delay computation.

derivative $\partial \mathcal{F}/\partial x_i$ of each x_i. Then we follow the algorithm given in Figure 4.2 to compute delay bounds.

4.4 Performance

We implemented both centred-form (Cencryst) and interval differential (Idcryst) versions of Crystal's PR-Slope model. To test these methods, we compared them against the original interval algorithm (Intcryst), a Monte Carlo version (Crystal with 10,000 trials), and the nominal delay (Crystal). To select variable values for each Monte Carlo trial, we employed a random number generator with a uniform distribution. In Idcryst, the maximum number of bisections was set to three. All implementations were written in the C programming language and run on a SUN SPARCstation. In this section, we present an analysis of the performance of each algorithm.

4.4.1 Quality of the Bounds

To determine the quality of the bounds, we conducted a number of experiments comparing the generated bounds to the theoretical bounds ($\hat{D}$) for a given circuit. As before, the theoretical bounds were approximated by Monte Carlo simulation with an unlimited number of trials. For all methods, the trigger transistor resistance was precomputed and treated as an independent variable in the delay function. Figures 4.3 and 4.4 illustrate the relationship between delay and stage length (i.e., the number of transistors in the stage) for a family of nMOS stages. These graphs display the percentage error for both the upper and lower bounds generated by each method. As these graphs demonstrate, the Monte Carlo and nominal value techniques always underestimate the bounds, while the interval approaches always overestimate them. Of all the methods, Idcryst produces the best bounds, coming within about 4% of the desired result. Note that the centred method generates a skewed interval, producing a tight upper bound and a poor lower bound. Since centred-forms yield an interval centered around the nominal value, we conclude that the nominal delay does not fall in the middle of the true bounds, thus causing the skewed centred values that we observe. Intcryst produces bounds within 8% of the theoretical values and the Monte Carlo technique comes within 12%. The nominal delay produces the poorest estimate of best- and worst-case delay. For most of these methods, the error grows with the

Stage	Size	Theoretical		Cencryst		Idcryst		Intcryst	
		min	max	min	max	min	max	min	max
1	2	6.8	9.3	6.7	9.3	6.8	9.3	6.8	9.3
2	3	54.3	72.9	53.1	72.9	54.3	72.9	54.1	73.1
3	2	10.3	14.1	10.1	14.1	10.3	14.1	10.3	14.1
4	2	10.6	14.4	10.4	14.4	10.6	14.4	10.6	14.4
5	2	1.2	1.6	1.1	1.6	1.2	1.6	1.2	1.6
6	1	0.7	1.0	0.7	1.0	0.7	1.0	0.7	1.0
7	1	0.8	1.1	0.7	1.1	0.8	1.1	0.8	1.1
8	3	20.4	27.4	19.9	27.4	20.4	27.4	20.3	27.5
9	1	2.0	2.7	1.9	2.7	2.0	2.7	2.0	2.7
Path total		107.1	144.5	104.6	144.5	107.1	144.5	106.8	144.8

Table 4.4: A B-SYS Critical Path Delay Computation (ns)

length of the stage. Fortunately, the stages encountered in most circuits are typically small, containing fewer than five transistors.

In addition to the above experiments, we also analyzed a portion of the Brown Systolic Array (B-SYS)[Hugh89]. We extracted a 2,000 transistor piece of the circuit and computed its critical path. This path consists of nine stages, each ranging in length from one to three transistors. Table 4.4 displays the delay bounds generated by each interval method for the entire critical path and for each individual stage in the path. Table 4.5 gives the percentage error in the bounds for each method, including the Monte Carlo and Nominal results displayed in the previous chapter. As the tables shows, the interval differential version produces the tightest delay bounds, achieving the theoretical bounds for all stages. Because the function is monotone over the ranges of all variables, this interval technique can precisely calculate the bounds. As in the previous experiments, Cencryst produces very tight upper bounds (matching the theoretical ones), but poorer lower bounds (only coming within 2.3% of the desired result). On average, Intcryst comes within 0.3% of both upper and lower bounds. The Monte Carlo method produces an average error of about 4.3%, while the nominal delay has an average error of about 15%.

4.4.2 Running Time of the Algorithm

The graph in Figure 4.5 depicts the relative speed of each of the implementations. As expected, the Monte Carlo method is the least efficient of the bounding techniques. Of

Stage	Size	Cencryst		Idcryst		Intcryst		Monte Carlo		Nominal	
		min	max	min	max	min	max	min	max	min	max
1	2	2.3	-0.1	0	0	0.3	-0.3	-5.9	5.9	-16.7	14.0
2	3	2.3	0	0	0	0.3	-0.3	-4.2	4.7	-16.0	13.6
3	2	2.4	-0.1	0	0	0.1	-0.1	-4.8	5.3	-17.0	14.3
4	2	2.4	0	0	0	0	-0.1	-3.8	3.7	-16.8	14.1
5	2	3.5	0	0	0	0.9	-0.6	-5.2	5.7	-18.1	15.4
6	1	2.9	0	0	0	0	0	-7.1	5.1	-18.6	15.3
7	1	3.9	0	0	0	0	0	-3.9	4.6	-19.5	15.6
8	3	2.3	-0.1	0	0	0.4	-0.4	-4.1	3.1	-16.0	13.6
9	1	2.0	0	0	0	0	0	-1.5	1.5	-16.2	13.6
Path total		2.3	0	0	0	0.2	-0.3	-4.3	4.4	-16.3	13.8

Table 4.5: % Error in Delay Bounds for the Critical Path

the interval-based techniques, the original interval algorithm is the most efficient. As stated in the previous chapter, this method runs roughly four times slower than the non-interval PR-slope model in Crystal and has a time complexity of $O(n)$, where n is the number of independent variables.

The centred-form method is the next fastest interval method. As previously stated, its running time is $0(sn)$, where s is the number of instructions in the straight-line program for computing the given function. For Crystal's delay equation, s is proportional to the number of variables in Equation 3.1; therefore, the running time of Cencryst is $O(n^2)$.

The interval differential method is the slowest of the interval algorithms. Since it computes derivatives for each of the n variables, this method requires $O(n^2)$ time for each function evaluation. If the function is monotone over the range of all variables, then the algorithm evaluates this function only once. If, however, the function is non-monotone, then we must bisect the intervals and evaluate the function over the subintervals. In the worst-case, the algorithm may evaluate the function $2^{b+1}-1$ times, where b represents the maximum number of interval bisections allowed. Therefore, this algorithm has a worst-case running time of $O(2^{b+1}n^2)$.

4.5 Summary

In this chapter, we presented two interval-based methods that improve on our original technique for modeling uncertainty in RC timing analysis. The first of these uses

Krawczyk's centred form, while the second employs an extension of Rall's interval differential calculus.

A comparison of the three interval-based approaches reveals a classic trade-off between efficiency and precision. Our experiments demonstrated that both the centred (Cencryst) and interval differential (Idcryst) algorithms produce tighter bounds than the original interval method (Intcryst). The primary reason for this is that the new methods take into account variable interdependence, while the original does not. The original, however, requires less time to compute its bounds.

The centred method is the second fastest interval approach, but generates bounds of inconsistent quality. In all of our experiments, this algorithm yielded a tight upper bound and a poor lower bound. These skewed bounds may be attributed to the fact that the nominal delay did not fall in the center of the true interval. Since the resulting centred interval is centered at this nominal value, Cencryst must overestimate the lower bound in order to encompass the theoretical upper bound.

In general, the interval differential approach is the slowest of the algorithms, but generates the tightest bounds. In our experiments, this algorithm produced bounds within about 4% of the true results using at most three bisections. It is also the only one of the three methods that provides a measure of the goodness of its estimate; the number of estimated subinterval bounds indicates the precision of the result.

All three interval-based approaches are much more efficient than Monte Carlo simulation. In experiments with 10,000 Monte Carlo trials, all three also generated significantly tighter bounds.

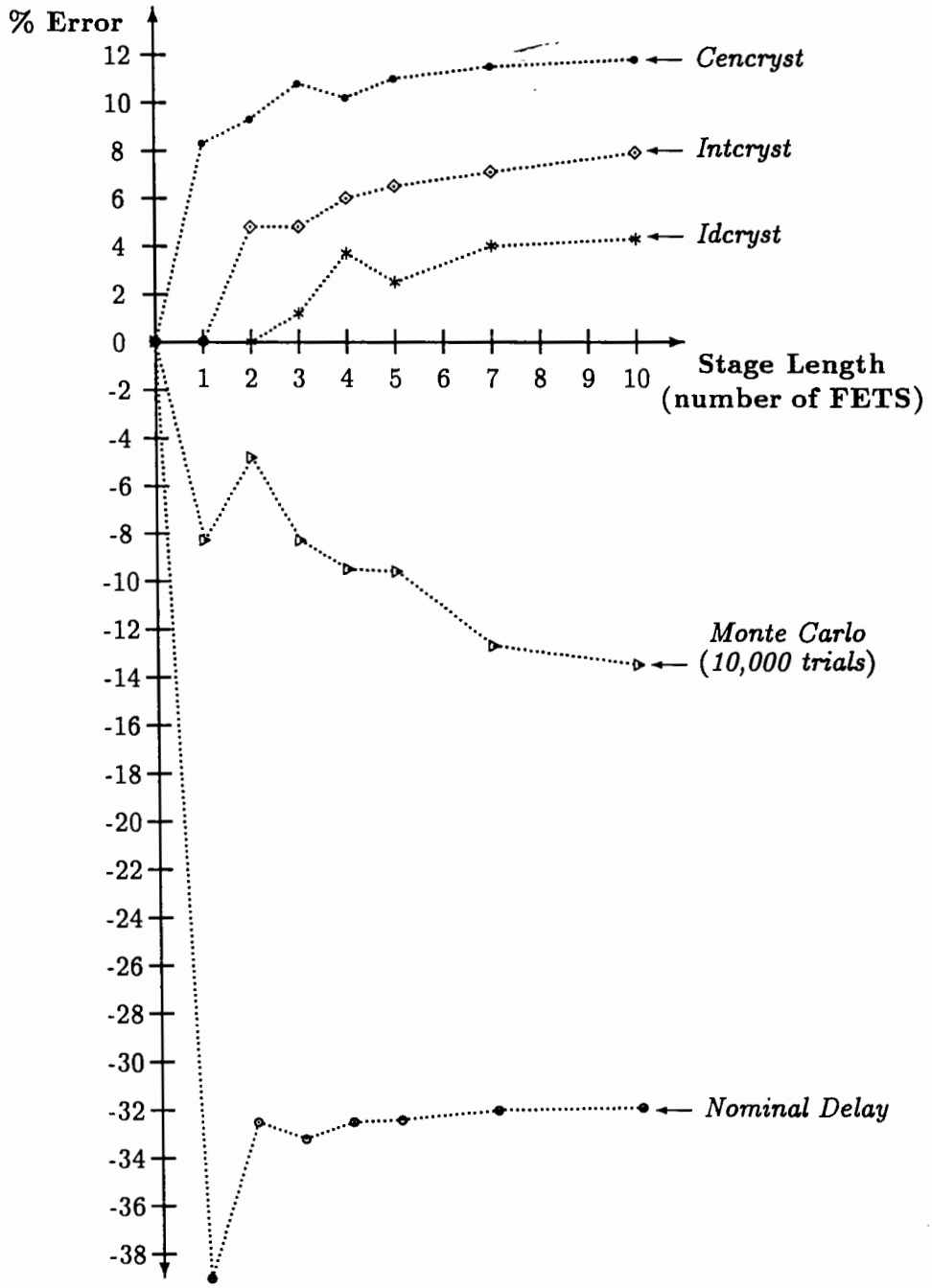


Figure 4.3: %Error vs. Stage Length (Lower Bounds)

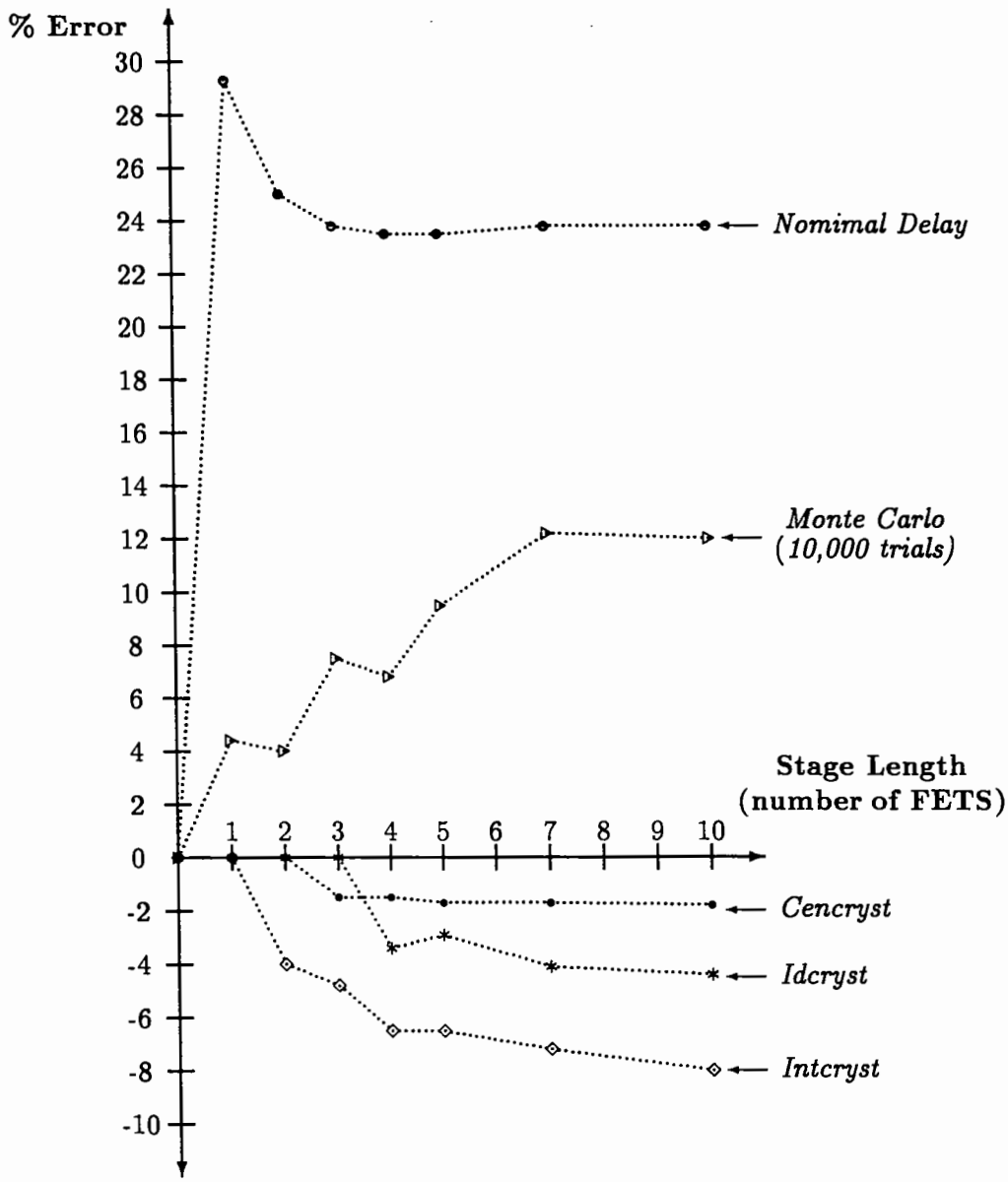


Figure 4.4: %Error vs. Stage Length (Upper Bounds)

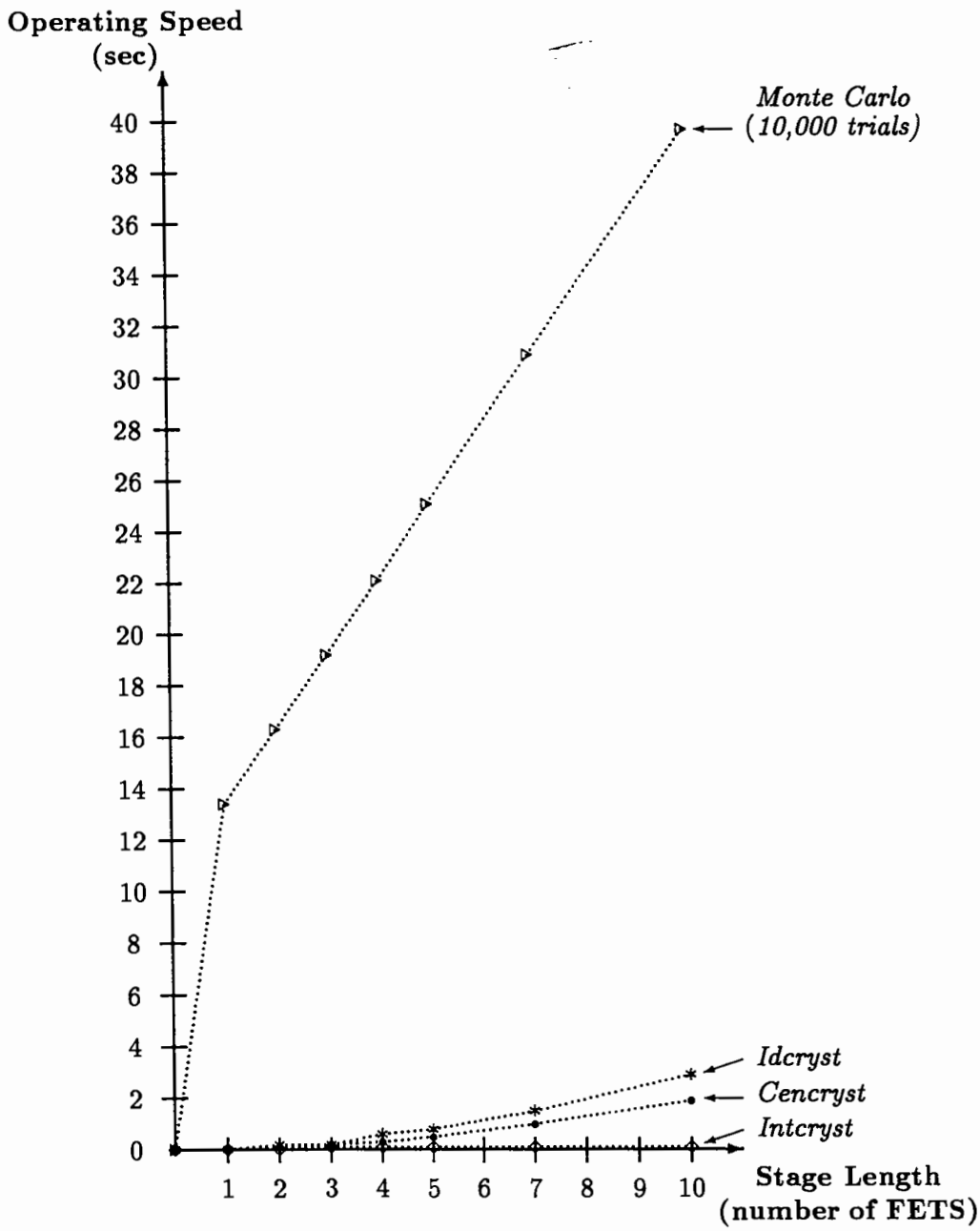


Figure 4.5: Operating Speed vs. Stage Length

Chapter 5

Uncertainty and the Longest Path Problem

In previous chapters, we discussed the problem of uncertainty in timing analysis and presented interval methods for computing bounds on stage delay. In this chapter, we address the second half of the problem – how to combine these uncertain stage delays to compute bounds on the delay of the entire circuit. To solve this problem, we employ a longest path algorithm.

The *classical* longest/shortest path problem[Tarj83, pp. 85-96] can be summarized as follows: Let $G = (V, E)$ be a directed graph with weighted edges, such that $length(v, w) \in \Re$ for all edges $(v, w) \in E$. Also designate $s \in V$ to be the source vertex. A path p is defined as a sequence of consecutive graph edges originating at the source vertex s. The length of a path p, denoted $length(p)$, is defined as the sum of the lengths of its edges. The longest (shortest) path from vertex s to any other vertex $v \in V$ reachable from s is the path from s to v whose length is maximum (minimum). If the graph contains a positive (negative) cycle, then no longest (shortest) path exists to vertices reachable from the cycle. For a given vertex $v \in V$, the standard algorithm returns the length $a \in \Re$ of the longest (shortest) path to vertex v and one representative path p from s to v such that $length(p) = a$.

For timing analysis, graph G depicts the timing dependencies between nodes in the circuit. In Crystal[Oust83, Oust84, Oust86], the vertices represent the input, output, and target nodes in the circuit. The edges of the graph represent stages activated by signal changes at these nodes. Each graph edge is labeled with the delay for that

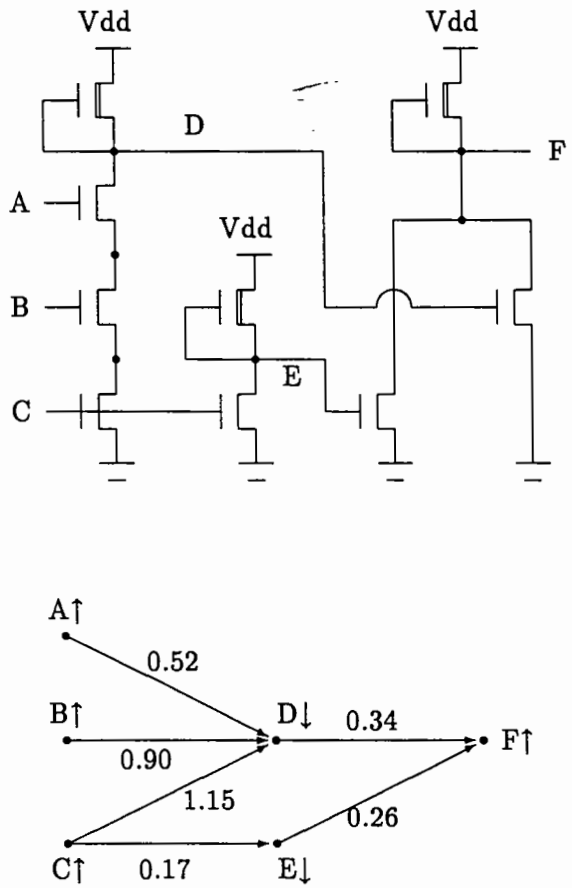


Figure 5.1: Timing Dependency Graph for a Transistor Network

stage computed by RC analysis. Figure 5.1 presents a nMOS transistor network and one possible timing dependency graph for this circuit. The vertices in the graph correspond to the labeled nodes in the circuit and the edges represent stages. For instance, a rising signal at node A could allow current to flow from GND to node D, thus producing a falling signal value at node D. When analyzed using Crystal's PR-Slope model, this stage has a delay of 0.52 ns. To represent the stage, the graph contains a directed edge from A to D with a weight of 0.52 ns. All the delays shown in this graph are computed using Crystal's PR-Slope model.

For timing verification at the logic level, similar dependency graphs are created [Wall86]. Here, the graph vertices represent the inputs and outputs of logic blocks and

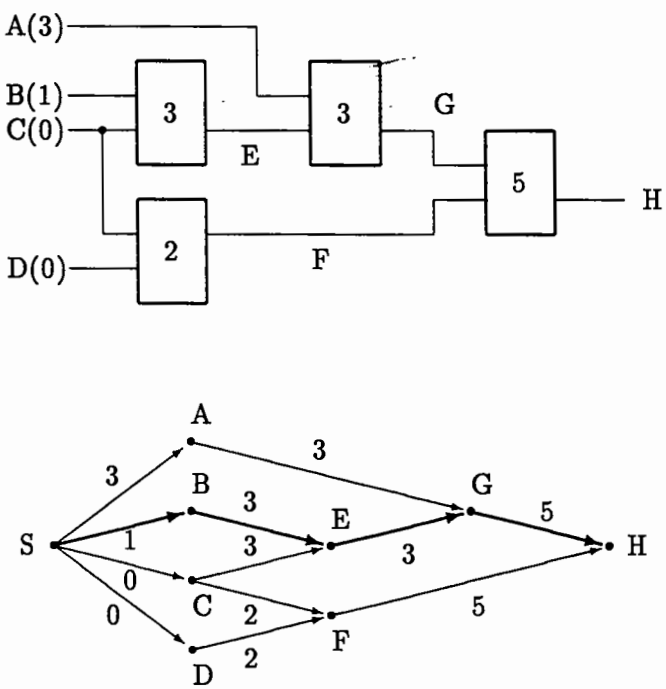


Figure 5.2: Timing Dependency Graph for a Logic Circuit

the edges represent the timing dependencies between them. Associated with each edge is the delay through the logic block for that input/output pair. For example, Figure 5.2 presents a simple logic circuit. Each circuit input is labeled with its *input arrival time* in parentheses and each logic block is labeled with its associated delay (ns). Below the circuit is its associated timing dependency graph. As before, the vertices in the graph correspond to the labeled nodes in the circuit and the edges are labeled with the delay through the circuit between the nodes. Note that we added a source vertex S to the graph and edges from this source vertex to each input node. We labeled each of these edges with the appropriate arrival time for each input.

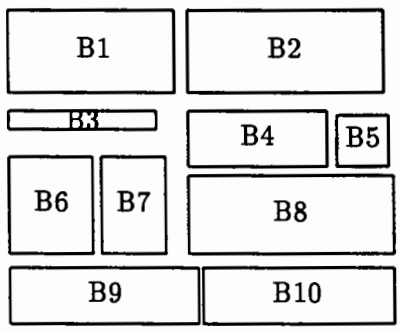
Once created, these timing dependency graphs are used to estimate the operating speed of the circuit and to detect potential timing errors. The longest path through the graph indicates the path through the circuit with maximum delay. For combinational circuits, this is used to determine minimum operating speed. For instance, the circuit in Figure 5.2 has a maximum delay of 12ns as determined by the path through vertices

SBEGH. For sequential circuits, the maximum delay between clocked components is used to determine the clock rate. Similarly, the shortest path through the graph indicates minimal delay and is used to locate race conditions. Race conditions occur when the delays along different paths through the circuit cause conflicting signals to arrive at a node.

Placement provides another instance of the longest path problem. Here, the graph (called a *channel position graph*) indicates the relative position of the cells in the layout. [Prea79]. Each vertex in the graph corresponds to a boundary between a cell block and a routing channel. Each edge represents one of these channels or logic cells and is labeled with its physical dimensions. Figure 5.3 depicts a sample layout and a vertical channel position graph for this layout[1]. Although routing is not shown in the placement diagram, the channels are sized to allow the necessary connections between blocks. The graph presents the relative positions of the blocks. Each edge in the graph is labeled with a block or channel identifier followed by its width (measured in mils). Block identifiers are denoted by the letter B and the corresponding number of the block. Channels are denoted by the letter C and are numbered from left to right as they are encountered in the layout. The longest path through a channel position graph defines the minimum length or width of the final layout. For instance, the minimum length of the layout in Figure 5.3 is 92 mils as determined by the path through blocks B1 and B8. Shortening the longest path length is tantamount to achieving a more compact design.

As these examples illustrate, solving the longest/shortest path problem is an important part of many VLSI design tools. For many of these applications, the weights on the graph edges are not exactly known. In timing analysis, the delays of circuit components cannot be precisely predicted prior to fabrication. The channel and cell dimensions needed for automatic placement cannot always be precisely determined. In hierarchical placement systems, the sizes of the circuit components at higher levels in the hierarchy are determined from cell layouts at lower levels. If these lower-level cells are not completely placed and routed, the dimensions of the higher-level components containing them are not known exactly, but it is possible to determine ranges for these values. Also full routing is required to accurately calculate channel width. Because of its computational expense, it is not practical to fully route each trial placement when

[1]This layout was adapted from a circuit in [Prea79].



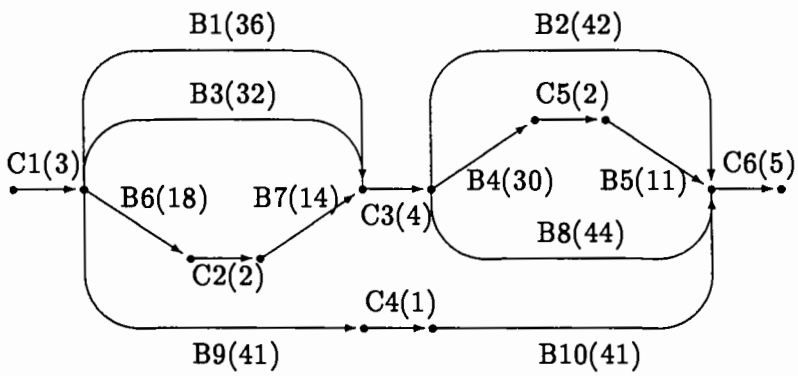


Figure 5.3: Vertical Channel Position Graph for Placement

exploring layout alternatives. Instead, channel width must be estimated until an acceptable placement is found. In these cases, we require a longest path algorithm for graphs with uncertain edge weights.

To model uncertainty, some timing verifiers [McWi80, Wall86] represent delay by the upper and lower bounds on its value. Some others [Hitc82, Wall86] represent uncertain delay with statistical distributions, employing a mean value and standard deviation for each delay. While these timing verifiers provide rules for combining uncertain delays to compute bounds on the delay of the circuit, the algebraic structure behind these rules is only intuitively defined. One of our goals is to formalize this underlying algebraic structure. Second, none of these systems returns the paths that produce the delay bounds, but these paths are needed to improve the speed of the chip. Our second goal is to return path as well as delay information.

The classical longest path algorithm returns one representative longest path to a given vertex. For many VLSI applications, one representative path is not sufficient. In timing analysis, it is not uncommon for several paths to have similar or identical delays. If the timing algorithm returns only one sample path, the designer may waste time reducing the delay on this path, only to discover a second path as bad as the first. It is therefore desirable to return the set of all paths whose delay values are within a certain tolerance of the worst-delay path. Since these paths may contain common edges, the designer can increase the speed of the entire design by reducing the delay through these common pieces. In our longest path algorithm, we return a *set* of longest paths, each of which could potentially be longest.

In this chapter, we present a theoretical framework based on interval algebra for solving the longest/shortest path problem with uncertain edge lengths. More formally, let I be the set of intervals over $\Re$ and let $G = (V, E)$ be a directed graph with interval-weighted edges such that $\mathit{length}(v, w) \in I$ for all edges $(v, w) \in E$. Also let $s \in V$ be designated as the source vertex. For any vertex $v \in V$, we return bounds $\hat{a} \in I$ on the length of the longest (shortest) path to vertex v and the set of all paths $\{p_1, p_2, ..., p_k\}$ from s to v in G such that $\mathit{length}(p_i) \cap \hat{a} \neq \emptyset$ for each path p_i. In other words, we return the set of all paths that have maximum (minimum) length under some possible assignment of values to the edge lengths. To solve this problem, we develop an interval algebra for manipulating uncertain edge lengths and then modify the standard longest/shortest path algorithm [Tarj83] to incorporate this framework. We conclude this chapter by analyzing the performance of the resulting interval path algorithms.

5.1 Classical Solutions to the Longest Path Problem

We begin with a brief review of the standard longest path problem adapted from Tarjan [Tarj83, pp. 85-96]. Given a directed graph with weighted edges, we wish to find paths through the graph with maximum length. There are four variations on this basic problem: single pair, single source, single sink, and all pairs. In the first of these, we seek the longest path between a given pair of vertices. In the single source problem, we are given only the source vertex s and must determine the longest path from s to v for every vertex v. The single sink problem is analogous: given a destination vertex t, find the longest path from v to t for every vertex v. The last version is the all pairs problem. As its name suggests, we seek to locate a longest path from s to t for every pair of vertices s and t. Since the single source problem can be adapted to solve all the other variations [Tarj83, p. 85], we will only discuss solutions to it.

In [Tarj83], Tarjan presents a number of solutions to the single source longest path problem. Although several versions of this algorithm are given, they all share the same basic structure and differ only in the search strategy employed. We therefore present the basic algorithm here and then discuss the different search options. Formally stated, we are given a directed graph $G = (V, E)$ with weighted edges $(v, w) \in E$ such that $length(v, w) \in \Re$ and a source vertex $s \in V$. For each vertex $v \in V$, the algorithm returns the length $a \in \Re$ of the longest path from s to v and a representative path p from s to v such that $length(p) = a$. The longest paths are returned as a *longest path tree* (i.e., a spanning tree rooted at source vertex s each of whose edges is part of a longest path to some vertex v). To compute longest paths, the algorithm partitions the vertices into three groups or states: *unlabeled*, *labeled*, and *scanned*. The unlabeled vertices are those that have not yet been examined, the labeled vertices are those that we wish to examine, and the scanned vertices are those that have already been examined. At the start of processing, the source vertex is marked labeled and all other vertices are marked unlabeled. In the course of the algorithm, labeled vertices are chosen, processed, and marked scanned. When a vertex is analyzed, it will cause other vertices to become labeled. The algorithm finishes when all the vertices have been scanned. These state transitions are summarized in Figure 5.4. In addition to state, each vertex v has two other associated attributes: *dist(v)* and *p(v)*. The first of these attributes is the current maximum path length from source s to vertex v and the second is its current parent in the longest path tree. These attributes are updated as we examine labeled vertices in

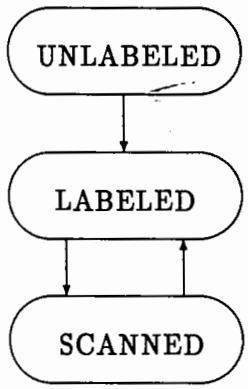


Figure 5.4: State Transitions for Graph Vertices

the graph. When processing concludes, the parent attributes $p(v)$ denote the longest path tree. It is important to note that if the graph contains a positive cycle, no longest path exists. In this case, the algorithm will never halt; therefore, we must check for positive cycles. The algorithm is summarized in Figure 5.5.

As previously mentioned, there are several versions of the above algorithm. Each one differs from the others only in the strategy used to select vertices from the labeled set. The aim of these methods is to increase the efficiency of the algorithm by choosing an efficient scanning order. Some common scanning orders for this problem are topological, breadth-first, and depth-first. The topological approach orders the vertices such that if (v, w) is a directed edge in the graph, v appears before w in the ordering. This approach works best for acyclic graphs and has a running time proportional to m, the number of edges [Tarj83, p. 89]. The breadth-first scanning strategy chooses the least recently labeled vertex. The running time of this algorithm is $O(nm)$ [Tarj83, p. 93]. The third scanning order is depth-first (i.e., we choose the most recently labeled vertex). The worst-case time complexity of this strategy is exponential in the number of vertices [Wall86, p. 684]. For example, suppose that vertex x is on the path from source vertex s to target vertex t. Also suppose that there exists a second path from s to t that also passes through vertex x such that the path length from s to x along this second path is greater than the path length from s to x along the first. Using a depth-first scanning order, we follow the first path marking the vertices scanned as we go. Later in the search, we follow the second path until we reach vertex x. Because this new

```
Initialize the vertices:
  Mark vertex s labeled
  Mark all other vertices v unlabeled
  dist(s) = 0
  For all other vertices v, dist(v) = -∞
  For all vertices v, p(v) = ∅
Until there are no more labeled vertices or we detect a positive cycle
  Select a labeled vertex v
  Mark vertex v scanned
  For each outgoing edge (v, w) for this vertex
    new dist = max(dist(v) + length(v, w), dist(w))
    If new dist ≠ dist(w)
      Update vertex w:
        Mark vertex w labeled
        dist(w) = new dist
        p(w) = v
    End if
  End for
End until
```

Figure 5.5: The Classical Longest Path Algorithm (Tarjan)

path has greater length, the maximum distance from s to x has changed. This change at vertex x will affect all remaining vertices on paths from s to t containing x; thus, vertex x and all its "descendants" must now be re-examined. In the worst-case, we could end up enumerating all paths through the graph; therefore, this algorithm has an exponential worst-case time complexity. Fortunately, the worst-case is rarely encountered in practice. Many timing verifiers, including Crystal[Oust83, Oust84, Oust86], employ this strategy for critical path analysis. Crystal's PR-Slope model uses the rise time from the previous stage to compute the delay of the next stage. Because the depth-first method traces the paths through the graph, it is a natural choice for this type of analysis. Table 5.1 illustrates each of these scanning orders for the graph in Figure 5.2.

5.2 Longest Paths with Uncertain Edge Length

Now we are given a directed graph $G = (V, E)$ with interval weighted edges such that $length(v, w) \in I$ for all edges $(v, w) \in E$ and a source vertex $s \in V$. For each vertex $v \in V$, the interval algorithm must return bounds $\hat{a} \in I$ on the length of the longest

Method	Scanning Order
Topological	SABCDEFGH
Breadth-First	SABCDGEFHGH
Depth-First	SAGHBEGHCFD

Table 5.1: Survey of Scanning Strategies

path from s to v and the set of all paths $\{p_1, p_2, ..., p_k\}$ from s to v such that for each path p_i, $length(p_i) \cap \hat{a} \neq \emptyset$. In effect, the resulting longest "path" to vertex v is actually a set of paths, each of which has maximal length under some legal assignment of values to the graph edges. To illustrate this, consider the directed graph with uncertain edge lengths shown in Figure 5.6. For this example, assume that the intervals are defined over the integers. By enumerating all possible edge length assignments, we can compute the set of longest paths from vertex A to I. Table 5.2 displays the longest path from A to I for each possible combination of edge values. The final totals are calculated by taking the union of the results from each trial. As shown, there are two possible longest paths from A to I (ABEHI and ABECFHI) and the bounds on longest path length are $[21, 23]$. Because we must execute the longest path algorithm for each possible combination of value assignments, this method has time complexity that is exponential in the number of unknown edge lengths even when using a constant time longest path algorithm. A quick method for computing bounds would be to run the standard longest path algorithm twice – once with all minimum edge weights and again with maximum values. While this does yield bounds, it does not return all the paths. As indicated by the first and last entries in the enumeration table, this method would return bounds of $[21, 23]$ for vertex I, but only one longest path (ABECFHI). To locate all longest paths, we present the following interval-based longest path algorithm.

5.2.1 An Interval Longest Path Algorithm

To create an interval longest path algorithm, we must modify the original algorithm in two areas. First, we must create and incorporate an interval algebra for manipulating uncertain edge lengths. Second, we must store multiple longest paths.

To address the first problem, we define two interval operations $\oplus$ (addition) and

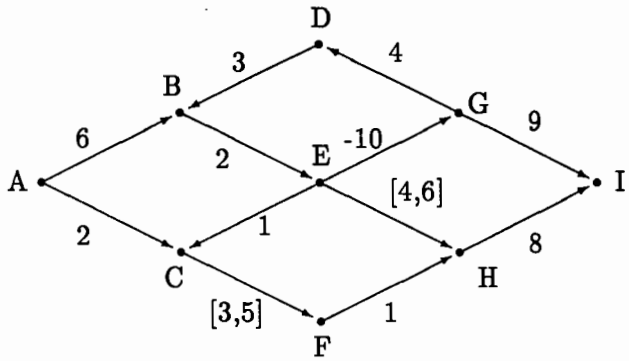


Figure 5.6: Directed Graph with Uncertain Edge Lengths

Edge		Longest Path A → I	
(C,F)	(E,H)	Length	Path
3	4	21	ABECFHI
3	5	21	ABEHI or ABECFHI
3	6	22	ABEHI
4	4	22	ABECFHI
4	5	22	ABECFHI
4	6	22	ABEHI or ABECFHI
5	4	23	ABECFHI
5	5	23	ABECFHI
5	6	23	ABECFHI
Total		[21,23]	{ABEHI, ABECFHI}

Table 5.2: Computing Longest Paths by Exhaustive Enumeration

MAX (maximum) over the set I as follows:

$$[a,b] \oplus [c,d] \equiv [a+c, b+d] \tag{5.1}$$

$$MAX([a,b],[c,d]) \equiv [max(a,c), max(b,d)] \tag{5.2}$$

These interval operators are the logical interval extensions of the discrete operators $+$ and max respectively and are derived from them using the following standard formula: $F(\hat{x},\hat{y}) = \{f(x,y) \mid x \in \hat{x} \text{ and } y \in \hat{y}\}$ [Moor79]. As $+$ is used to compute path length in the original algorithm, $\oplus$ is used to compute bounds on path length in the interval algorithm. Similarly, MAX is used in place of max to compute bounds on longest path length. We also need the interval relational operator $\neq$ to compare interval path lengths. This operator is defined as follows:

$$[a,b] \neq [c,d] \Longleftrightarrow a \neq c \text{ or } b \neq d \tag{5.3}$$

Having derived these equivalent interval operators, we substitute them into the path analysis algorithm to handle unknown edge lengths.

The simple substitution suggested above generates a longest path algorithm which returns bounds on the longest path length, but we also wish to return the set of paths that generate these bounds. To accomplish this task, we must save some additional information. In the original algorithm, the longest paths are saved in the parent attributes of the vertices. If the longest path from s to w contains edge (v,w) then $p(w) = v$. Because we need to store multiple paths, $p(w)$ now contains a set of possible parents for vertex w. We also associate a key (denoted $key(v)$) with each parent in the set, denoting the current maximum path length to vertex w along the path containing this parent. More formally, let path p be a path from s to w containing edge (v,w) such that $length(p) \cap dist(w) \neq \emptyset$; therefore, path p is a potential longest path to vertex w. To store this path, we save parent vertex v with the upper bound of $length(p)$ as its key. Whenever $dist(w)$ is updated, the parent set $p(w)$ must be purged of all parents whose keys are no longer elements of $dist(w)$. $Key(v) \notin dist(w)$ if and only if $length(p) < dist(w)$; therefore, path p through parent v cannot be a longest path to vertex w. When execution completes, these parent sets indicate the resulting set of potential longest paths for the graph.

Parent sets may be implemented in a variety of ways. The simplest implementation is a linked list of parents sorted in increasing order by key. With this method, each parent deletion requires $O(1)$ time, because the parent with the smallest key is always

```
Initialize the vertices:
  Mark source vertex s labeled
  Mark all other vertices v unlabeled
  dist(s) = [0, 0]
  For all other vertices v, dist(v) = [−∞, −∞]
  For all vertices v, p(v) = ∅
Until there are no more labeled vertices or we detect a positive cycle
  Select a labeled vertex v
  Mark vertex v scanned
  For each outgoing edge (v, w) for this vertex
    New path length = dist(v) ⊕ length(v, w)
    key(v) = upper bound of new path length
    New dist = MAX(new path length, dist(w))
    If new dist ≠ dist(w)
      Update vertex w:
        Mark vertex w labeled
        dist(w) = new dist
        Purge p(w) of all parents u such that key(u) ∉ dist(w)
        Add vertex v with key key(v) to p(w)
    Else if key(v) ∈ dist(w)
      Add vertex v with key key(v) to p(w)
    End if
  End for
End until
```

Figure 5.7: The Interval Longest Path Algorithm

found at the front of the list. Insertions, however, have a worst-case time of $O(n)$, since we may have to traverse the entire linked list. A more clever implementation for this set is a heap with the condition that the key of each node is less than that of its children. For this implementation, insertions cost $O(\log n)$, the depth of the heap. Since the parent with the smallest key is always on the top, deletions require constant time. Restoring the heap condition after a deletion, however, requires $O(\log n)$ time, where n represents the number of parents in the set. In the worst-case, n is the number of vertices in the graph. Fortunately, this situation rarely occurs, since in practice most parent sets contain relatively few entries.

As with the original algorithm, any of the scanning strategies given in the previous section may be employed to order the labeled vertices for evaluation. The changes made to incorporate intervals do not affect the scanning strategy at all. The interval algorithm is described in Figure 5.7.

5.2.2 Shortest Paths

The algorithm just presented can be easily altered to compute shortest paths. First, we require an interval minimum operator MIN to compute bounds on shortest path length. MIN is the interval extension of the discrete operator min and is defined as follows:

$$MIN([a,b],[c,d]) \equiv [min(a,c), min(b,d)] \tag{5.4}$$

As MAX and $\oplus$ are used to compute bounds on longest path length, MIN and $\oplus$ are similarly used to compute bounds on shortest path length. Second, we must change the key associated with each vertex in a parent set. Since we are computing shortest paths, we use the lower bound on path length as the key. Hence, if $key(v) \not\in dist(w)$, then $length(p) > dist(w)$ and path p through parent v cannot be a shortest path for vertex w. No further changes are required.

5.2.3 Using the Results

The interval longest path algorithm returns the set of longest paths as the subgraph $G' = (V, E')$ of G such that $(v,w) \in E'$ if (v,w) is an edge on some potential longest path p in G. The parent sets indicate these edges; thus, if $v \in p(w)$ then edge $(v,w) \in E'$. To illustrate this, consider the directed graph in Figure 5.6 with source vertex A. The above algorithm computes the set of longest paths from A to each vertex and bounds on the longest path lengths. The resulting longest path graph G' is displayed in Figure 5.8. Each vertex in the graph is labeled with its maximum distance from source node A. Once created, G' can be used to extract and display useful information about the longest paths in G. Two possible applications are enumerating longest paths and finding common edges.

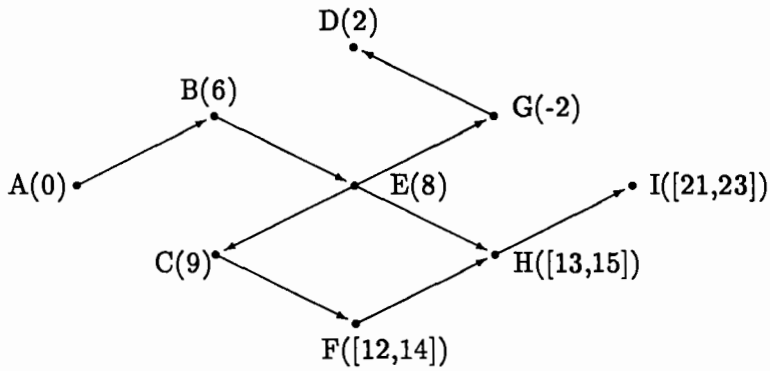
Timing analysis programs typically return the critical path through the circuit in addition to its delay. G' can be used to locate and return a list of potential longest paths from source vertex s to a given destination vertex t. To enumerate these paths, we start at the destination node t and trace the parent pointers back through the graph to source vertex s. Since this algorithm enumerates all possible paths from s to t in G', it has an exponential worst-case time complexity. In the worst-case, the number of paths is $O(k^n)$ where k is the maximum outdegree of a vertex in G' and n is the number of vertices. Because it assumes that each path p from s to t passes through all graph vertices and that each vertex has maximum outdegree, this worst scenario

rarely occurs. For example, the longest path graph in Figure 5.8 has nine vertices with a maximum outdegree of three and yet contains at most two longest paths to any given vertex. Beneath this graph is a detailed listing of the longest paths to vertex I. As shown, there are two possible longest paths from A to I (ABEHI and ABECFHI) and the bounds on longest path length are $[21, 23]$. Note that these results match those given in Table 5.2.

To increase the operating speed of a circuit, the chip designer must reduce the delay along the critical paths through the circuit. If all critical paths pass through the same area of the chip, the designer can reduce the delay on all critical paths by reducing the delay through this common piece. In our longest path algorithm, we therefore wish to locate the edges common to all longest paths between vertices s and t. Assume that we have a connected graph representing the longest paths between these vertices. If an edge is common to all longest paths, its removal will split the graph into two disjoint subgraphs (See Figure 5.9). We, thus, can modify an algorithm for finding articulation points to locate these common edges.[2] To solve this problem, we first construct an undirected graph $G''(V'', E'')$ from G'. This new graph differs from G' in two respects. First, it contains only those vertices and edges in G' that are on some longest path between vertices s and t. Second, we insert a special vertex along each edge of the resulting graph. For instance, if directed edge $(v, w) \in E'$ is on some longest path between s and t, then we create a new vertex u', add vertices v, w and u' to V'', and add undirected edges (v, u') and (u', w) to E''. The graph G'' can be constructed using depth-first search on G' in $O(m)$ time, where m is the number of edges in E'. To locate common edges, we now employ standard algorithms to locate the articulation points in graph G'' [Aho74, pp. 176-187]. A special vertex $u' \in V''$ is an articulation point of G'' if and only if its corresponding edge (v, w) in G' is a common edge. Finding articulation points requires $O(m')$ time, where m' is the number of edges in G''. Since m' is at most $2m$, we conclude that common edges can be located in time $O(m)$, where m is the number of edges in E'.

Figure 5.10 illustrates this algorithm for our previous longest path example. The algorithm begins with the longest path graph G' shown in the upper left-hand corner of the diagram. From this graph, an undirected graph G'' is constructed, containing only the paths from source A to vertex I. Special vertices representing the graph edges have

[2]From a private communication with Roberto Tamassia.




```
The longest path to vertex I has length [ 21.00, 23.00]
The set of all possible longest paths:
  From vertex A to B with edge length [  6.00,  6.00] = [  6.00,  6.00]
  From vertex B to E with edge length [  2.00,  2.00] = [  8.00,  8.00]
  From vertex E to C with edge length [  1.00,  1.00] = [  9.00,  9.00]
  From vertex C to F with edge length [  3.00,  5.00] = [ 12.00, 14.00]
  From vertex F to H with edge length [  1.00,  1.00] = [ 13.00, 15.00]
  From vertex H to I with edge length [  8.00,  8.00] = [ 21.00, 23.00]
    Path length = [ 21.00, 23.00]

  From vertex A to B with edge length [  6.00,  6.00] = [  6.00,  6.00]
  From vertex B to E with edge length [  2.00,  2.00] = [  8.00,  8.00]
  From vertex E to H with edge length [  4.00,  6.00] = [ 12.00, 14.00]
  From vertex H to I with edge length [  8.00,  8.00] = [ 20.00, 22.00]
    Path length = [20.00,22.00]
```

Figure 5.8: Computing Longest Paths with Intervals

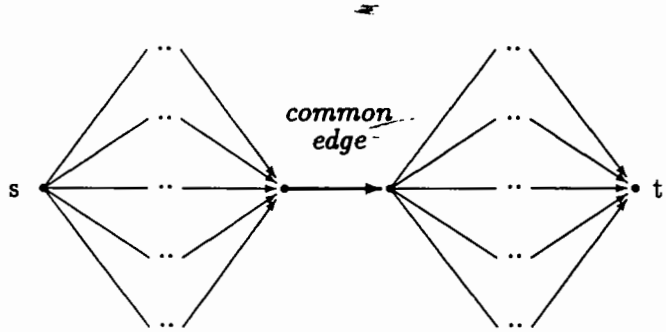


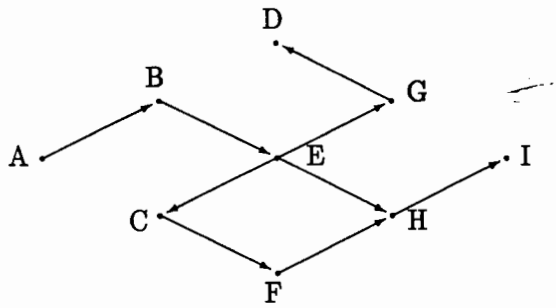
Figure 5.9: Common Edges

also been added to the graph. These vertices are assigned unique identifiers (in lower-case letters) denoting their edge endpoints. The third graph displays the articulation points in graph G''. As shown, vertices B, E, H, ab', be', and hi' are all articulation points. The special vertices among these articulation points represent common edges; therefore, edges (A,B), (B,E), and (H,I) are the edges common to all longest paths between vertices A and I. These edges are highlighted in the final graph.

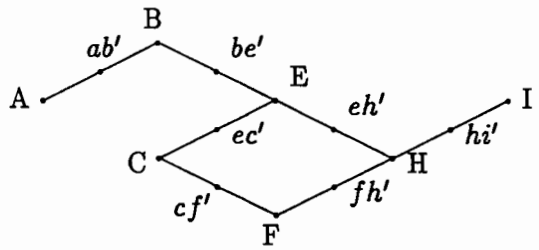
5.2.4 A Timing Example

In Figure 5.11, we present the critical path analysis for the nMOS circuit in Figure 5.1. The first graph in Figure 5.11 depicts the timing dependencies between the labeled nodes in the circuit. The graph vertices represent the input, output, and target nodes in the circuit and the edges represent the timing dependencies between these nodes. For instance, a rising signal value at node C allows current to flow from GND to node E, thus producing a falling signal value at node E. Associated with each edge (v, w) is the delay (ns) along the path through the circuit to vertex w triggered by the signal change at vertex v. These delays were computed using the interval RC analysis algorithm presented in Chapter 3. Because the original timing dependency graph does not have a single source node, we add source vertex S and a dashed edge from S to each of the circuit inputs (A,B, and C). These dashed edges are labeled with the *input arrival times* for each input node.

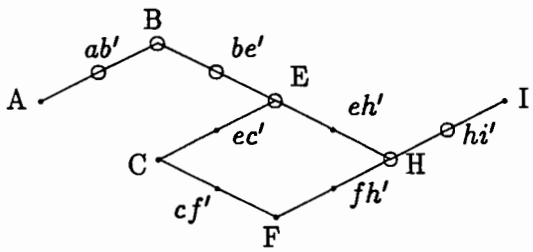
Given the timing dependency graph and the source vertex S, we used our interval longest path algorithm to compute the critical paths through the network. The second



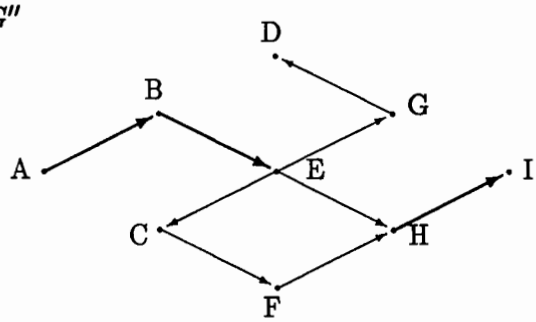
Longest Path Graph G' with source vertex A



Graph G'' for paths to vertex I



Articulation Points in G''



Common Edges in Longest Paths from A to I

Figure 5.10: Finding Common Edges

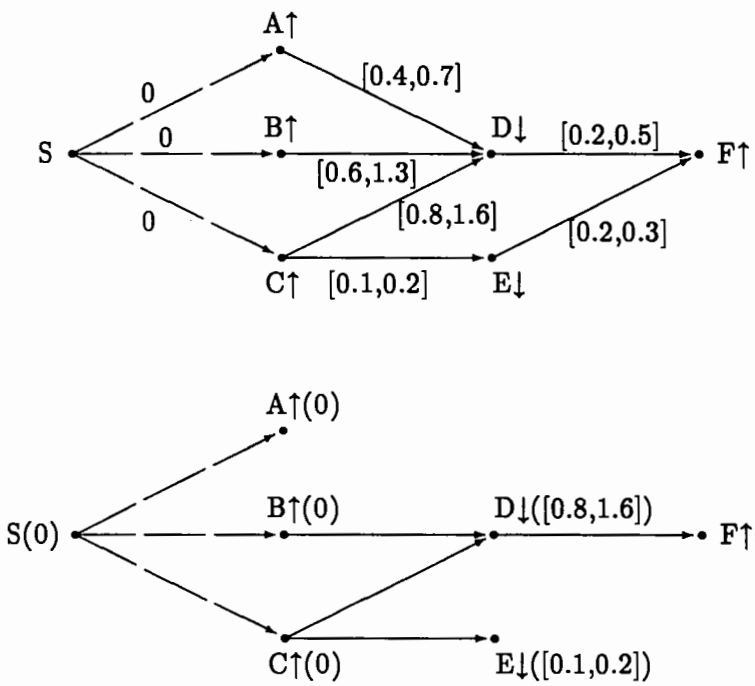
graph in Figure 5.11 presents the results of this computation. This graph depicts the set of longest paths and the maximum delay (in ns) from the source vertex to each node in the graph. The listing beneath this graph gives the details of the longest delay paths from S to F. As shown, there are two longest paths to node F. One is activated by the signal change at node B and the second by the change at node C. The maximum signal delay to node F is $[1.0, 2.1]$ ns. This means that a rising signal at the inputs of the circuit will require from 1.0 to 2.1 ns to propagate to the output through the slowest paths in the circuit.

5.3 Proof of the Method

In this section, we formally prove that our interval algorithms compute the correct bounds on longest or shortest path length on a graph with uncertain edge weights. The classical shortest path problem is an algebraic path problem defined over the closed semi-ring $(\Re, min, +, \infty, 0)$, where $\Re$ includes ∞ and $-\infty$. In [Aho74, pp. 195-201], Aho, Hopcroft and Ullman prove that the classical algorithm for computing shortest path length is correct by the properties of this algebra. Let I denote the set of intervals over $\Re$. We claim that our interval algebra $(I, MIN, \oplus, [\infty, \infty], [0, 0])$ also forms a closed semi-ring. Hence, we conclude that if the original algorithm using the algebra $(\Re, min, +, \infty, 0)$ correctly computes shortest path lengths on graphs with discrete edge weights, then our interval version employing the algebra $(I, MIN, \oplus, [\infty, \infty], [0, 0])$ will correctly calculate bounds on shortest path length on graphs with interval-weighted edges. A similar argument can be made for the algebra $(I, MAX, \oplus, [-\infty, -\infty], [0, 0])$ used when computing longest paths. We now prove that our interval algebra forms a closed semi-ring.

Theorem 5.1 $(I, MIN, \oplus, [\infty, \infty], [0, 0])$ *forms a closed semi-ring.*

Proof. First, we must show that $(I, MIN, [\infty, \infty])$ is a monoid. As shown in Appendix A, MIN is closed, associative, and has identity $[\infty, \infty]$. Similarly, $(I, \oplus, [0, 0])$ is also a monoid. Since ∞ is the annihilator for $+$, $[\infty, \infty]$ is the annihilator for $\oplus$. From the properties of *min*, we see that MIN is commutative and idempotent. Next, we must show that $\oplus$ distributes over MIN; thus, we must show that $MIN([a, b], [c, d]) \oplus [e, f] = MIN([a, b] \oplus [e, f], [c, d] \oplus [e, f])$ and $[e, f] \oplus MIN([a, b], [c, d]) = MIN([e, f] \oplus [a, b], [e, f] \oplus [c, d])$ for all $[a, b], [c, d], [e, f] \in I$. Applying the definitions of MIN and $\oplus$,




```
The longest path to vertex F has length [  1.00,  2.10]
The set of all possible longest paths:
  From vertex S to C with edge length [  0.00,  0.00] = [  0.00,  0.00]
  From vertex C to D with edge length [  0.80,  1.60] = [  0.80,  1.60]
  From vertex D to F with edge length [  0.20,  0.50] = [  1.00,  2.10]
    Path length = [  1.00,  2.10]

  From vertex S to B with edge length [  0.00,  0.00] = [  0.00,  0.00]
  From vertex B to D with edge length [  0.60,  1.30] = [  0.60,  1.30]
  From vertex D to F with edge length [  0.20,  0.50] = [  0.80,  1.80]
    Path length = [  0.80,  1.80]
```

Figure 5.11: Longest Path Analysis for Timing Verification

we must show that $[min(a,c)+e, min(b,d)+f] = [min(a+e, c+e), min(b+f, d+f)]$ and $[e + min(a,c), f + min(b,d)] = [min(e+a, e+c), min(f+b, f+d)]$. Because $+$ distributes over min, this is obviously true. Next, we must show that if $\hat{a}_1, \hat{a}_2, ...$ is countable sequence of elements in I, then $\widehat{MIN}(\hat{a}_1, \hat{a}_2, ...)$ exists and is unique. Furthermore, we must show that MIN is associative, commutative, and idempotent over infinite as well as finite sequences. Since these properties hold for min and MIN is defined in terms of min, these properties hold for MIN. Finally, $\oplus$ must distribute over MIN for countably infinite sequences as well as finite ones. This follows from the definitions of the interval operators and the fact that $+$ distributes over min for countably infinite sequences. □

5.4 Performance

Both the longest and shortest path algorithms described in this chapter were implemented in the C programming language and run on a SUN SPARCstation. Since most VLSI applications of the longest path algorithm employ either breadth or depth-first scanning orders, we incorporated both of them in our implementations. In this section, we discuss the performance of the algorithms. In particular, we examine their running time and the quality of the bounds they produce.

5.4.1 Quality of the Bounds

In timing verification, the longest path algorithm is employed to compute the delay through a circuit. Here, edge lengths correspond to stage delays and are calculated using RC timing analysis. As we stated in Chapter 3, the interval methods for computing stage delays may produce overly conservative bounds for some circuits. An interesting question is how does this "error" in edge length (stage delay) affect path length (circuit delay)? We now show that edge error does not accumulate. Let p be any path through graph G such that path p consists of k edges with actual lengths $\hat{l}_1, \hat{l}_2, ..., \hat{l}_k$. For each edge i, let l_i represent the upper (lower) bound of $\hat{l}_i$. Now let $\hat{c}_1, \hat{c}_2, ..., \hat{c}_k$ be the lengths that we are given for those edges and let c_i denote the upper (lower) bound of $\hat{c}_i$. The percent error e_i in the upper (lower) bound of each edge i is $e_i = \frac{l_i - c_i}{l_i} \cdot 100$. Let L represent the actual upper (lower) bound on the length of path p; thus, $L = \sum_{i=1}^{k} l_i$. Likewise, let C be the computed value for path p; thus, $C = \sum_{i=1}^{k} c_i$. Let E represent

the percent error in the upper (lower) bound on path length for p. We now prove that path length error E is bounded by maximum edge length error; thus,

Theorem 5.2 $E \leq max(e_1, e_2, ..., e_k)$

Proof. By definition, $E = \frac{L-C}{L} \cdot 100$. Expanding L and C, $E = \frac{(l_1+...+l_k)-(c_1+...+c_k)}{L} \cdot 100$. This equation can be rewritten as $E = \frac{(l_1-c_1)+...+(l_k-c_k)}{L} \cdot 100$; therefore, $E = \sum_{i=1}^{k} \frac{(l_i-c_i)}{L} \cdot 100$. Multiplying each term by $\frac{l_i}{l_i}$ and rearranging yields $E = \sum_{i=1}^{k} \frac{l_i}{L}(\frac{l_i-c_i}{l_i} \cdot 100)$. But $\frac{l_i-c_i}{l_i} \cdot 100$ is just e_i; therefore, $E = \sum_{i=1}^{k} \frac{l_i}{L} e_i$. This equation shows that path length error is a weighted sum of the edge length errors. Let $\overline{e} = max(e_1, e_2, ..., e_k)$. Clearly, $E \leq \sum_{i=1}^{k} \frac{l_i}{L} \overline{e}$. But $\sum_{i=1}^{k} \frac{l_i}{L} = 1$; therefore, $E \leq \overline{e} \cdot 1$. We have just proved that $E \leq max(e_1, e_2, ..., e_k)$. □

To illustrate this theorem, Tables 5.3 and 5.4 present an example from timing analysis. Table 5.3 displays the stage delays and total path delay from an actual critical path calculation for a portion of the B-SYS chip [Hugh89]. The *Theoretical* column contains the theoretical stage and path delay bounds and the remaining columns contain bounds generated by various interval and non-interval methods. In Table 5.4, we present the errors in stage and path delay for each method. As expected, the path delay error for each case is less than or equal to the maximum error of its individual stages. Using this theorem, we also conclude that if the bounds on edge length are correct, then our algorithm is guaranteed to find the tightest possible bounds on path length.

5.4.2 Running Time of the Algorithm

Our interval algorithms differ from the non-interval versions in two main areas. First, our interval versions employ interval operations. Since we must perform two discrete operations to evaluate each interval one, we expect that our interval algorithms will run twice as slow as the originals. Second, the interval versions associate a set of parents with each vertex, instead of a single parent. The cost of maintaining these sets decreases the efficiency of the interval versions. If we implement parent sets using heaps, then insertions and deletions each cost $O(\log n)$, where n indicates the size of the heap. In the worst-case, n is the number of vertices in the graph. Hence, with this implementation, the operating speed of the interval algorithms will decrease by a factor of $O(\log n)$ in the worst case.

Stage	Size	Theoretical		Intcryst		Monte Carlo		Nom
		min	max	min	max	min	max	
1	2	6.84	9.28	6.82	9.31	7.24	8.73	7.98
2	3	54.28	72.90	54.14	73.08	56.55	69.49	62.98
3	2	10.34	14.12	10.33	14.14	10.84	13.37	12.10
4	2	10.61	14.43	10.61	14.44	11.01	13.90	12.39
5	2	1.16	1.62	1.15	1.63	1.22	1.53	1.37
6	1	0.70	0.98	0.70	0.98	0.75	0.93	0.83
7	1	0.77	1.09	0.77	1.09	0.80	1.04	0.92
8	3	20.41	27.40	20.33	27.51	21.24	26.54	23.68
9	1	1.97	2.65	1.97	2.65	2.00	2.61	2.29
Path total		107.08	144.47	106.82	144.83	111.65	138.14	124.54

Table 5.3: A Critical Path Delay Calculation (ns)

Stage	Size	Intcryst		Monte Carlo		Nominal	
		min	max	min	max	min	max
1	2	0.29	-0.32	-5.85	5.93	-16.67	14.01
2	3	0.26	-0.25	-4.18	4.68	-16.03	13.61
3	2	0.10	-0.14	-4.84	5.31	-17.02	14.31
4	2	0	-0.07	-3.77	3.67	-16.78	14.14
5	2	0.86	-0.62	-5.17	5.65	-18.10	15.43
6	1	0	0	-7.14	5.10	-18.57	15.31
7	1	0	0	-3.90	4.59	-19.48	15.60
8	3	0.39	-0.40	-4.07	3.14	-16.02	13.58
9	1	0	0	-1.52	1.51	-16.24	13.58
Path total		0.24	-0.25	-4.27	4.38	-16.31	13.80

Table 5.4: % Error in Critical Path Delay Bounds

To confirm this hypothesis, we compared the operating speeds of our algorithms with those of the non-interval versions. As search strategy greatly affects performance, we tested implementations containing both breadth and depth-first scanning orders. We ran both the interval and non-interval implementations of each algorithm on identical graphs and compared their response times. Our tests showed that the interval implementations ran roughly twice as slow as their non-interval counterparts. These results suggest that the increase in operating speed was due largely to the interval operations, rather than the parent lists. A survey of these graphs confirmed this suspicion, demonstrating that most parent lists contain only a few entries.

5.5 Summary

In this chapter, we presented a variation on the classical longest/shortest path problem by introducing uncertain edge lengths. Instances of this problem occur in many areas of VLSI design, including placement and timing verification. To solve this problem, we represented uncertain edge lengths as intervals and created an interval algebra for computing bounds on path length. We then used this algebra to create an algorithm which not only computes bounds on path length, but also returns the set of paths that generated these bounds. We then presented a proof of correctness for our approach.

After implementing our algorithms, we tested their performance. We proved that path length error is bounded by the maximum error of its edge lengths; therefore, if the edge lengths are correct, our algorithms generate the tightest possible bounds on path length. We also tested the speed of these algorithms as compared to their non-interval counterparts. Analysis suggests that the interval versions may be slower than the non-interval ones by $O(\log n)$ in the worst case. Experiments on several graphs, however, demonstrated that the interval versions are not substantially slower than their non-interval counterparts.

Chapter 6

Placement using Uncertain Costs

Rapid increases in circuit size and complexity in the past ten years have made automatic placement an essential element of the VLSI design tool suite. The goal of automatic placement is to find an arrangement of the circuit components that facilitates routing and optimizes certain aspects of the circuit. These goals are encoded within an *objective function*, which provides a measure of the quality of each configuration. [Prea86] presents an overview of the most common automatic placement techniques. These can be divided into two major categories: constructive and iterative. The constructive approaches begin with a partial placement and build a complete placement from it. Typical examples of these algorithms include cluster growth [Dai89], min-cut partitioning [Breu77, Fidu82, Kern70, MS89, Tric86], global placement [Jack86], and branch and bound [Dai89, Prea79, Tric86]. Iterative techniques, such as pairwise interchange and simulated annealing [Kirk83], start with a complete placement and attempt to improve it. A more detailed description of each of these placement methods appears in [Prea86].

In spite of their differences, all placement algorithms share a number of common features. They all evaluate an objective function for each placement generated, compare the values returned by the objective functions to chose the best configuration of elements, and save the best result found. While there are almost as many different objective functions as placement programs, the most common measures of placement quality are circuit area, channel density, maximum wire length, and circuit delay. The

first of these metrics is layout area. To be accurate, this computation must include component and routing area. A problem, however, arises when these areas are not precisely known. In top-down or hierarchical placement, the dimensions of higher-level cells are determined by placements at lower-levels in the hierarchy. If the lower-level cells are not completely placed and routed, the exact dimensions of the higher-level cells containing them may not be known. Similarly, channel width can only be accurately determined by doing a complete routing. This calculation is computationally expensive and thus is practical only for the final placement configuration. While exploring placement alternatives, the routing area is usually estimated.

The second common placement objective is to minimize wire congestion over the circuit. Channels containing many nets have to be wider to accommodate the interconnections and may be difficult to route. Reducing wire congestion increases routability and reduces circuit area. Congestion is calculated by summing the channel density of all channels. Channel density is typically computed by counting the number of nets that cross a routing channel.

Reducing total wire length is another standard goal. Total wire length is defined as the sum over the entire circuit of the lengths of all nets. As with channel width, this can only be accurately calculated after full routing; therefore, wire length for a single wire is usually approximated by computing the half perimeter of the rectangular bounding box containing the net.

A fourth common layout metric is circuit delay. A placement with long wire delays along its critical paths can significantly degrade the performance of a circuit; therefore, many current placement algorithms incorporate delay information in their objective functions. One such method locates the critical delay paths through the circuit and uses these delays to weight the nets. Those nets along the critical path then receive a higher priority in the placement algorithm. By carefully placing the components from this worst-delay path, the designer hopes to reduce wire delay along the critical path and consequently the total delay of the circuit. The delays used to compute these weighting factors are subject to variation. Consequently, logic-level timing verifiers [McWi80, Hitc82] frequently represent component and wire delays as min-max delay bounds or statistical distributions.

This brief survey of the most common measures of placement quality reveals that many objective functions contain parameters whose exact values cannot be predicted in advance. Among them are cell area, routing area, wire length, and circuit delay. While

these values can be estimated, they cannot be precisely known before the layout is finished or the chip is fabricated. In spite of the prevalence of uncertainty in placement metrics, current placement algorithms use only single-valued estimates in their objective functions.

Placement programs that account for uncertainty offer several advantages over current systems. One advantage is a more comprehensive measure of placement quality. Since the parameters used in objective functions have variable values, different combinations of their values will result in different placement costs and possibly a different "best" configuration. In contrast to conventional placement algorithms which return a single optimal configuration, algorithms incorporating uncertainty will return a set of placements, each of which is optimal for some assignment of values. As the placement process continues, the values of these parameters may be refined, allowing the designer to chose one placement from the set of potentially optimal placements. Since it is unlikely that the final values of each parameter will exactly match the precomputed single-valued estimates, the final layout may be quite different from the one generated using a conventional placement algorithm.

Additional information about the best layout is the second advantage to including uncertainty. Systems that return a single configuration give no details about the features of the layout that make it optimal. By comparing the layouts from a set of best configurations, we begin to discover the similarities between them. These similarities reveal the factors that produced the best layouts for a particular circuit. For instance, certain components may appear in the same location in all potentially optimal placements. Also, strongly connected components may always appear in the same relative position. In addition to increasing our understanding of the design, these invariants may also indicate critical areas on the chip, which can be changed to further improve the placement.

A third advantage of incorporating uncertainty is that it facilitates iterative improvement without costly recomputation. As placement proceeds, the values of some uncertain parameters may be refined or may change. For instance, in hierarchical placement, the dimensions of the blocks at higher-levels will be refined as the layouts are completed at the lower-levels of the hierarchy. Similarly, channel width and wire length estimates are updated after full routing. Using current placement algorithms, the designer must re-execute the complete algorithm each time these values change. Since finding a good placement is a computationally expensive process, this is not practical.

Using an algorithm that models uncertainty, we may not have to completely recompute the layout after these changes. As long as the updated values or ranges lie within the ranges initially given for each parameter, the previous placement results are still valid. Instead of running the placement algorithm again, we simply recompute the cost of the placements in the optimal set and discard those that no longer belong.

For these reasons, we propose a framework for placement with uncertain objective functions. Traditional systems employ only single-valued estimates of parameter values in their objective functions, returning one possible cost for each configuration. Using these values, the algorithm returns the configuration whose cost in minimum. Instead of ignoring parameter variation, systems incorporating uncertainty use the full range of possible values for each variable to compute bounds on the cost of a configuration. Using these cost ranges, it returns a set of placements, each of which is optimal for some combination of input values.

In this chapter, we present one solution to this problem. As before, we represent placement costs as intervals and create an algebra for manipulating these uncertain values. We then incorporate this algebra within an existing placement algorithm [Prea79] to create one that handles uncertain objective functions. Although a branch and bound placement algorithm was chosen, the general principles employed apply to other placement strategies. In particular, we address the common issues of evaluating an uncertain objective function, comparing two placements with uncertain costs to find the best configurations, and storing the resulting minimum-cost placements.

6.1 A Branch and Bound Placement Algorithm

In [Prea79], Preas and vanCleemput present a branch and bound placement algorithm for arbitrarily-sized blocks. Because of its exponential computational complexity, the branch and bound technique is only practical for circuits with relatively few components (i.e., fewer than 20). Hence the placement algorithm begins by partitioning the circuit to create several smaller layout problems of a suitable size. The result of top-down partitioning is a hierarchical circuit description, where each level contains a collection of rectangular circuit components and the interconnections between them. Beginning at the bottom levels of the hierarchy, a constructive branch and bound algorithm is applied recursively to compute an initial placement. Because of the size of the search space for some larger circuits, the constructive algorithm may require an unreasonably

long time to find the optimal solution. For these cases, Preas and vanCleemput propose an iterative improvement scheme based on the same branch and bound strategy to improve an existing layout.

6.1.1 The Objective Function

The goal of this placement algorithm is to minimize circuit area; thus, we must estimate the size of each trial placement. Preas and vanCleemput begin by creating the *horizontal* and *vertical channel position graphs* that define the layout. A channel position graph is a directed acyclic graph whose nodes represent the boundaries between circuit components and channels and whose edges represent the channels and circuit blocks themselves. There are two special nodes in each graph depicting the right and left (top and bottom) boundaries of the layout. Associated with each edge is a *length*, representing the physical dimensions of the corresponding block or channel. At the lowest level of the hierarchy, the dimensions of the circuit components are given. At higher levels, circuit block size is computed from layouts generated at lower levels of the hierarchy. Channel width is estimated for all trial placements, but full routing is used to determine the channel width for the final configuration.

Figure 6.1 depicts a sample layout and its corresponding horizontal and vertical channel position graphs[1]. The graph edges corresponding to circuit components are denoted by block identifiers and edges corresponding to channels are labeled with the letter H (horizontal channels) or V (vertical channels) and are given unique identifiers. Each graph edge is also labeled with the estimated length or width of its associated block or channel in parentheses. All dimensions are given in mils.

Once the channel position graphs for a particular placement are created, we traverse the graphs to find the longest paths between the boundary nodes. The length of this longest path gives the minimum length (or width) of the layout. These dimensions are then multiplied to determine circuit area. For the example in Figure 6.1, the longest path through the vertical channel position graph has length 92 mils and determines the minimum length of the circuit. Likewise, the longest path through the horizontal channel position graph has length 71 mils and determines the minimum width of the layout. Hence, the minimum area of this configuration is 6532 square mils.

[1]This layout was adapted from a circuit in [Prea79]

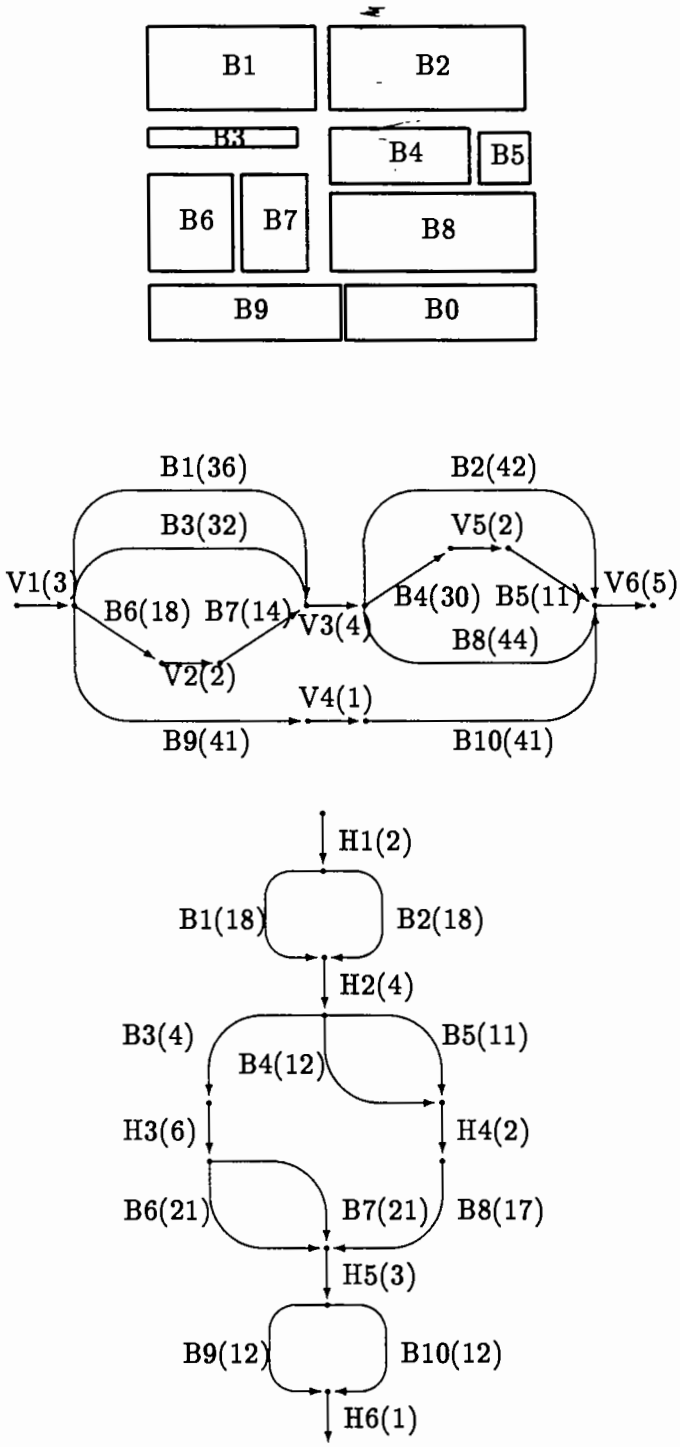


Figure 6.1: Horizontal and Vertical Channel Position Graphs

6.1.2 Branch and Bound Placement

Given a set of circuit blocks and their interconnections, Preas and vanCleemput employ branch and bound techniques to find the configuration with minimum area. Branch and bound is a well-known potentially-exhaustive search strategy [Aho83, pp. 330-336] that traverses a search tree to locate the minimum cost solution. Each node in the search tree represents a set of solutions. At each successive level of the tree, those solutions are refined, until each leaf node contains only a single solution. Associated with each node is a number, indicating a lower bound on its cost. When visiting a node, we compare its cost to that of the current best complete solution. If the cost of the node is greater than or equal to the current minimum cost, we prune that solution from the search tree; otherwise, we consider each of its children and compute their costs. At the leaf nodes, we locate and save the one with the lowest cost. When all the nodes have been either visited or pruned, the search is complete and the best solution is returned.

For the placement problem, the internal nodes of the search tree represent partial placements and the leaf nodes represent complete placements. The root of the tree contains a *seed* placement, consisting of one or more blocks and their positions relative to each other. By adding unplaced blocks to a partial placement, we generate its child configurations. Since our objective function is circuit area, the cost of each (partial or complete) placement is its size as computed from its channel position graphs. The area of the smallest complete placement found in the course of the search is used to prune successive configurations. Initially, we require an estimate of minimum layout area to prune partial placements until a complete placement with lower cost is found. This initial estimate is either derived from designer specifications or computed from a given complete layout. The branch and bound placement algorithm is summarized in Figure 6.2.

To illustrate this algorithm, Figure 6.3 displays a complete branch and bound search tree for a circuit consisting of three components with fixed pin positions and connections. The average dimensions of the three circuit components are shown in Table 6.1. Each node in the tree represents a partial or complete placement. Below each configuration is a placement identifier (e.g., P1, P2, ..., Pn) followed by placement cost. Although routing is not shown in the diagram, its effects are evident in the cost metric. This explains why symmetric placement configurations have slightly different costs.

```
Initialize minimum cost
Generate seed placement
Add placement to search tree
While there are unvisited placements on search tree
    Choose placement from search tree
    If placement cost ≥ minimum cost
        Prune placement from search tree
    Else if placement is complete
        Minimum cost = placement cost
        Save placement as current best
    Else (placement is partial and unpruned, so expand it)
        Choose unplaced block
        For each possible position in partial placement
            Create child placement by adding unplaced block at position
            Compute child placement cost
            If child cost ≥ minimum cost
                Prune child from search tree
        End for
End while
Return the minimum placement
```

Figure 6.2: Branch and Bound Placement Algorithm (Preas and vanCleemput)

The seed placement P1 contains a single block. Its child placements P2-5 are created by adding a second block to the seed placement in all possible positions. The rest of the configurations are leaf nodes representing complete placements consisting of all three blocks. An X below a layout indicates that the placement was pruned in the course of the search. Assume that we are given an initial cost bound of 1200. After examining nodes P1, P2, and P6, we find a complete placement with cost 1100. P6 now becomes the minimum-cost placement. The only other child of P2 with lower cost is P11, which then becomes the best placement. We now examine P3 and its children. Finding none with lower cost, we proceed to P4 and find P25 with cost 960. Since P5's cost is greater that of the current best solution, we prune P5 and do not generate any of its children. At the end of the search, the optimal configuration is P25 with cost 960.

Figure 6.4 presents the results of a layout computation on a circuit with ten cell blocks[2]. The average dimensions (in mils) of each block are displayed in Table 6.3. After generating and examining more than 180,000 partial and complete configurations, the algorithm returned the minimum-cost placement displayed in Figure 6.4. The resulting configuration has an area of 1998 square mils. Below the layout, the horizontal and vertical channel position graphs defining this optimal placement are displayed. As shown the longest paths through the horizontal and vertical channel position graphs are 37 and 54 mils respectively.

6.2 An Interval Placement Algorithm

Having described the branch and bound placement algorithm, we now discuss the changes that must be made to this algorithm to incorporate uncertain objective functions. While its general structure remains unaffected, three aspects of the algorithm do change. They are

- Computing uncertain cost functions
- Comparing values returned by these cost functions
- Saving minimum-cost placements

[2]This example was adapted from a circuit in [Cies84]

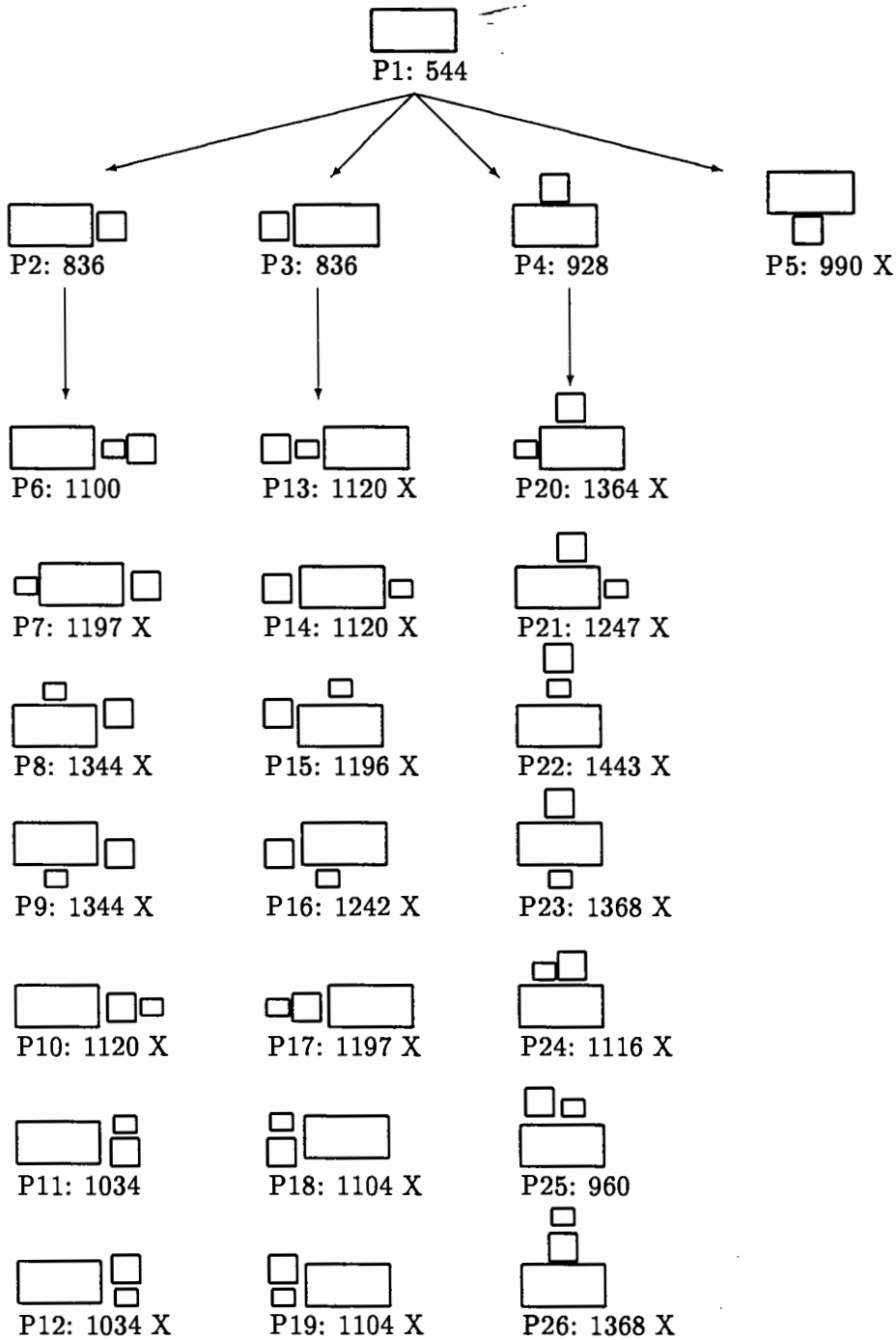
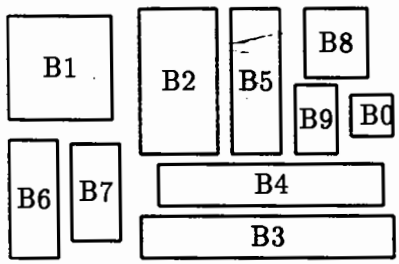
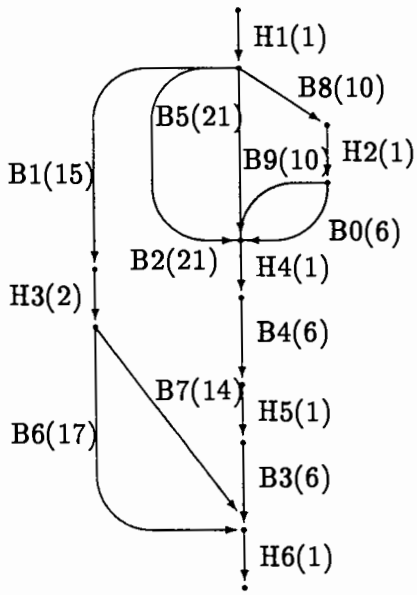


Figure 6.3: Branch and Bound Search Tree for Placement





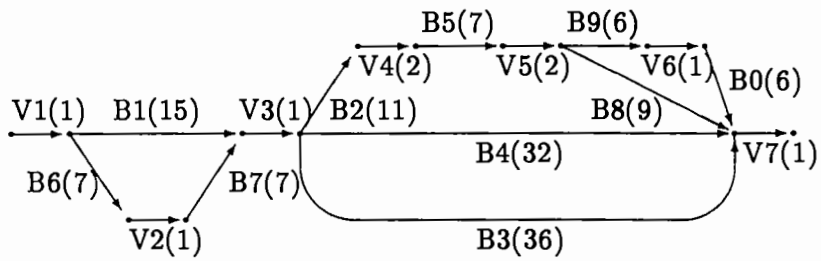


Figure 6.4: Resulting Minimum-Cost Placement with Channel Position Graphs

6.2.1 Computing an Interval Objective Function

The first of these issues is how to calculate an interval objective function. Since there are many different objective functions in current placement programs, it would be impractical to enumerate them all. Instead, we present a general methodology for creating an interval objective function and then illustrate it with a single example (circuit area).

To compute an interval objective function, we represent all uncertain values (e.g., channel widths, component sizes, wire lengths, component delays) in the cost function as intervals. We then create an application-specific interval algebra for manipulating these ranges. The interval operators in this algebra are the logical interval extensions of the discrete operators employed to compute the objective function. These interval operators F are derived from their non-interval counterparts f using the following standard formula [Moor79]:

$$F(\hat{x}, \hat{y}) \equiv \{f(x, y) \mid x \in \hat{x} \text{ and } y \in \hat{y}\}. \tag{6.1}$$

Having derived the equivalent interval operators, we then substitute them into the objective function to create an interval version of it. The resulting interval objective function returns bounds on its value when given interval inputs.

To illustrate this method, suppose that our objective function is layout area. Since the cell dimensions and channel widths needed to compute area may not be known exactly, we represent their values as intervals. Now we must create an interval algebra for manipulating these ranges. The non-interval objective function described in section 6.1.1 employs three operations to compute layout area: $+$ to sum the edge lengths in the graph, max to compare path lengths to determine the longest one, and $\cdot$ to multiply the longest path lengths to calculate the area. For the interval version, we derive the interval extensions $\oplus$ (addition), MAX (maximum), and $\odot$ (multiplication) of these operators using Equation 6.1. The resulting interval operators are defined as follows:

$$[a, b] \oplus [c, d] \equiv [a + c, b + d] \tag{6.2}$$

$$[a, b] \odot [c, d] \equiv [min(ac, ad, bc, bd), max(ac, ad, bc, bd)] \tag{6.3}$$

$$MAX([a, b], [c, d]) \equiv [max(a, c), max(b, d)] \tag{6.4}$$

To compute the dimensions of a given layout, the non-interval version employs a standard longest path algorithm[Tarj83, pp. 85-96] to calculate the length of the

longest path through each channel position graph. By replacing the discrete operators in this algorithm with their interval counterparts, we create an interval longest path algorithm that returns bounds on the length of the longest paths through a directed graph (See Chapter 5 for details). Once bounds on the length and width of the layout are found, the $\odot$ operator can be used to compute bounds on circuit area.

6.2.2 Comparing Placement Cost

Comparing the values returned by the objective functions is the second issue that we must address. Placement costs are used to prune unpromising placements from the search and to choose the best placement. To perform these tasks, we must be able to compare two interval placement costs to determine which is smaller.

In the first instance, we compare a placement cost to the current minimum cost, pruning the placement from the search if its cost exceeds the current minimum value. With interval cost functions, we need an interval comparator. We define the interval comparator $>$ as follows:

$$[a, b] > [c, d] \Longleftrightarrow a > d. \tag{6.5}$$

According to this definition, interval $[a, b]$ is greater than $[c, d]$ if and only if each member of the first interval range is greater than all members of the second. In our interval placement problem, a placement is thus pruned from the search tree if and only if all members of its cost range are greater than all members of the current minimum bound. This guarantees that we already have a better solution for all possible placement cost values; therefore, this placement cannot be a best placement and thus can be discarded.

Second, these metrics are used to compute the minimum placement cost. In our interval placement algorithm, the costs are now intervals; therefore, we need an interval MIN operator to calculate the minimum of two placement costs. This operator is derived from the discrete operator *min* and is defined as follows:

$$MIN([a, b], [c, d]) \equiv [min(a, c), min(b, d)]. \tag{6.6}$$

The interval returned by this operator represents the set of minimum costs obtained by comparing two placements over all possible combinations of their values.

6.2.3 Saving the Minimum-Cost Results

It is not enough to know the minimum placement cost, we must also save the placement configurations themselves. The original placement algorithm computes the minimum placement cost $a \in \Re$ and returns a single representative configuration with cost a. In reality, there may be several configurations with minimum or close to minimum cost, any one of which could be used in the final design. Our interval version of the placement algorithm returns bounds $\hat{a} \in I$ on minimum placement cost and a *set* of configurations such that each placement in this set has a cost that lies within $\hat{a}$. Under some scenario, any of these placements could be optimal. Since each of these placements represents a potential best configuration, we must modify the algorithm to save them. Whenever we locate a complete placement with cost $\hat{c}$ such that $\hat{a} \cap \hat{c} \neq \emptyset$, we first update the minimum placement cost using the MIN operator described above and add the new placement to the set. Since the minimum placement cost may have changed, we must update the set of optimal placements by deleting all placements whose costs now exceed the new minimum cost bound. To perform this function, we use the previously-defined interval $>$ operator. For example, let $\hat{b}$ represent the cost of a particular configuration in the optimal set and let $\hat{a}$ represent the updated minimum-cost bounds. If $\hat{b} > \hat{a}$, then for each element of $\hat{b}$ there exists another configuration in the optimal set with smaller cost. Since this placement cannot be an optimal placement under any scenario, we must delete it from the set.

The set of optimal configurations can be implemented in a variety of ways, including linked lists, binary trees, and heaps. One possibility is a linear linked list sorted in decreasing order by the lower bound on placement cost. Inserting a placement in this list has a worst-case time complexity of $O(n)$, where n is the number of configurations in the set. Deleting an item is much more efficient, since the placements with the largest areas are located at the front of the list and can be retrieved in $O(1)$ time.

Another possible implementation is the binary tree. In this case, placements are ordered such that the minimum cost of the left child is less than that of its parent. Likewise, the minimum cost of the right child is greater than that of its parent. This implementation yields a worst-case insertion time of $O(n)$ and an average time of $O(\log n)$. The deletion times are similar. With a little extra computational effort, we can maintain a balanced binary tree, thus guaranteeing insertion and deletion costs of $O(\log n)$.

```
Initialize minimum cost bound
Generate seed placement
Add placement to search tree
While there are unvisited placements on search tree
    Choose placement from search tree
    If placement cost > minimum cost
        Prune placement from search tree
    Else if placement is complete
        Minimum cost = MIN(minimum cost, placement cost)
        Prune minimum list of all placements whose cost > minimum cost
        Add new placement to list of minimum placements
    Else (placement is partial and unpruned, so expand it)
        Choose unplaced block
        For each possible position in partial placement
            Create child placement by adding unplaced block at position
            Compute child placement cost
            If child cost > minimum cost
                Prune child from search tree
        End for
End while
Return the minimum placements
```

Figure 6.5: Interval Branch and Bound Placement Algorithm

The third and best of these set implementations is a heap with the condition that the minimum cost of each parent configuration is greater than those of its children. The cost of inserting a placement in this structure is $O(\log n)$. Since the placement with the greatest cost is kept at the top of the heap, this item can be retrieved in $O(1)$ time. Restoring the heap property after a deletion requires $O(\log n)$ time. As a result, the total cost of deleting a placement is also $O(\log n)$.

6.2.4 The Interval Algorithm

Incorporating the changes described above yields a new interval branch and bound placement algorithm, as summarized in Figure 6.5.

Figure 6.6 illustrates this interval placement technique. The diagram depicts a branch and bound search tree for a circuit with three components. Table 6.1 gives the interval dimensions of these components. As before, each node represents a partial or complete placement and is labeled with a unique identifier and its associated cost.

Block	Length			Width		
	min	max	avg	min	max	avg
1	29.7	30.3	30.0	14.85	15.15	15.0
2	9.9	10.1	10.0	9.9	10.1	10.0
3	7.92	8.08	8.0	5.94	6.06	6.0

Table 6.1: Component Dimensions for the Search Tree Example

In this example, the costs are interval ranges. In Table 6.2, we show how the set of minimum-cost placements is updated during the search. For each placement visited, the table displays the resulting minimum-cost bound and the contents of the placement set. For each configuration in the optimal set, we also display the lower bound on its cost in parentheses. This value is used to prune placements from the set whenever the minimum cost changes. At the end of the search, the set of minimum-cost placements contains two configurations P25 and P12 and has a minimum cost in the range $[941, 979]$.

For our second example, we re-examine the ten-cell circuit in Figure 6.4. Instead of using only average values, we represent uncertain cell and channel sizes by their interval ranges. The interval dimensions of each block are displayed in Table 6.3. In the course of the search, the algorithm examined over 3 million partial and complete placements and returned a set of five optimal configurations for the circuit. The results of this computation are displayed in Figure 6.7. As shown, the best layouts had an area bound of $[1980.2, 2016.4]$ square mils. A closer examination of the set of minimum-cost placements reveals many similarities. For instance, elements B2, B3, B4 and B5 remain in the same relative position in all best placements. Other elements tend to cluster together, such as B8, B9 and B0. Some components appear in the same position in all five configurations. For example, block B1 is in the upper left-hand corner in all optimal solutions. These invariants indicate areas on the chip which can be improved to reduce the area of all optimal configurations. For example, blocks B2, B3, B4 and B5 all lie on critical paths in the horizontal channel position graphs of each configuration; therefore, these blocks and the channels between them define the width of the layout in all instances. By reducing the area of these blocks or by improving the routing between them, we can reduce the area of all five solutions.

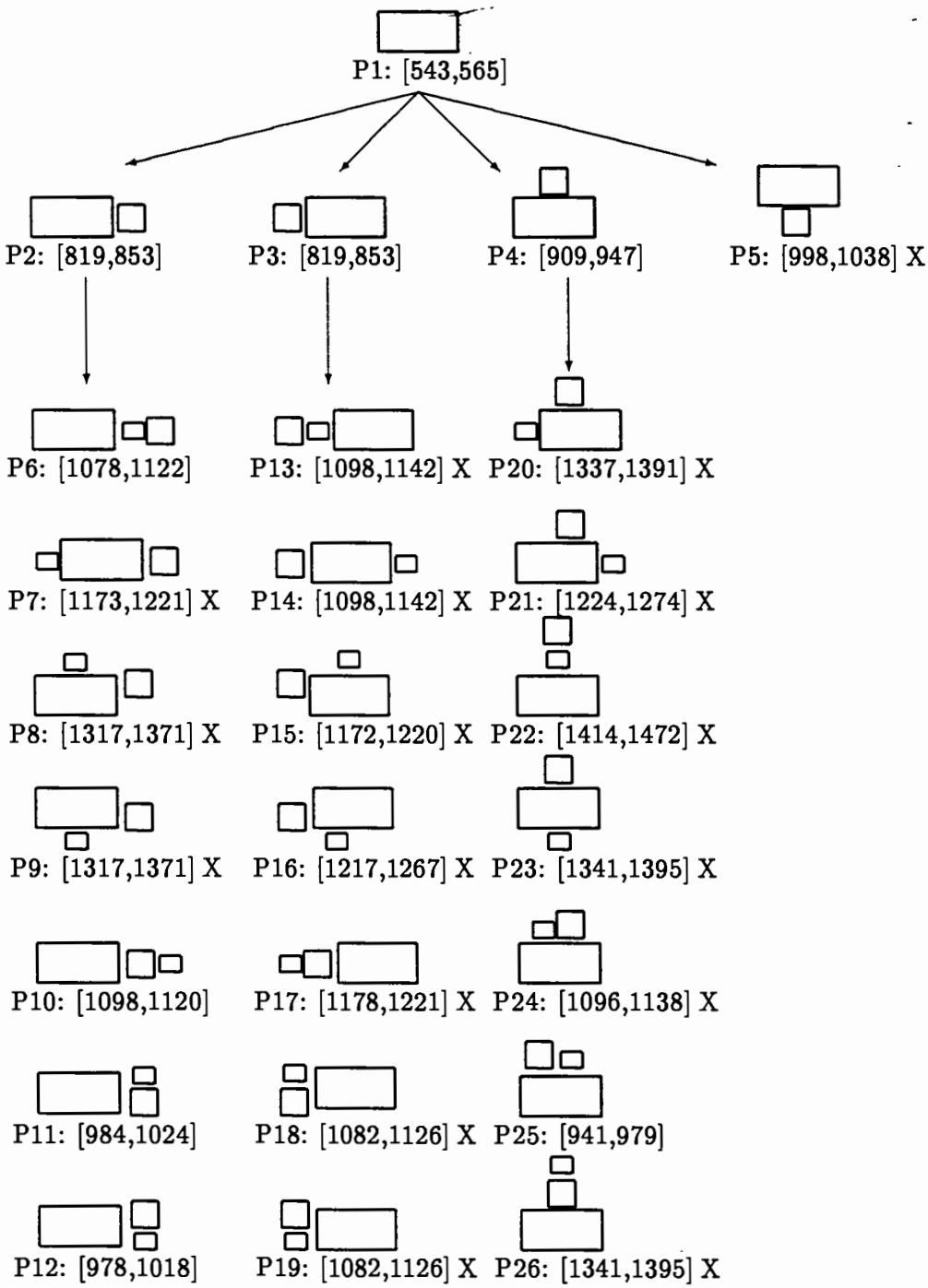
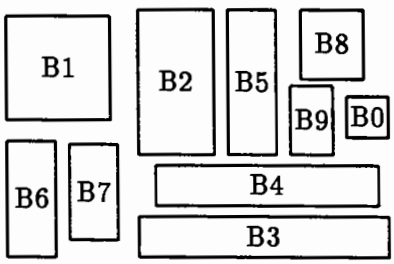


Figure 6.6: Interval Branch and Bound Search Tree for Placement

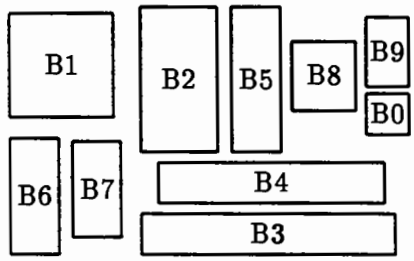
Visit			Minimum-Cost Results	
Placement	Cost	Type	Cost Bounds	Placement Set
P1	[543, 565]	partial	[1200, 1200]	
P2	[819, 853]	partial	[1200, 1200]	
P6	[1078, 1122]	complete	[1078, 1122]	$\{P6(1078)\}$
P7	[1173, 1221]	complete	[1078, 1122]	$\{P6(1078)\}$
P8	[1317, 1371]	complete	[1078, 1122]	$\{P6(1078)\}$
P9	[1317, 1371]	complete	[1078, 1122]	$\{P6(1078)\}$
P10	[1098, 1120]	complete	[1078, 1120]	$\{P10(1098), P6(1078)\}$
P11	[984, 1024]	complete	[984, 1024]	$\{P11(984)\}$
P12	[978, 1018]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P3	[819, 853]	partial	[978, 1018]	$\{P11(984), P12(978)\}$
P13	[1098, 1142]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P14	[1098, 1142]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P15	[1172, 1220]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P16	[1217, 1267]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P17	[1178, 1221]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P18	[1082, 1126]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P19	[1082, 1126]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P4	[909, 947]	partial	[978, 1018]	$\{P11(984), P12(978)\}$
P20	[1337, 1391]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P21	[1224, 1274]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P22	[1414, 1472]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P23	[1341, 1395]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P24	[1096, 1138]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P25	[941, 979]	complete	[941, 979]	$\{P12(978), P25(941)\}$
P26	[1341, 1395]	complete	[941, 979]	$\{P12(978), P25(941)\}$
P5	[998, 1038]	partial	[941, 979]	$\{P12(978), P25(941)\}$
Final			[941, 979]	$\{P12(978), P25(941)\}$

Table 6.2: Finding the Set of Minimum-Cost Placements

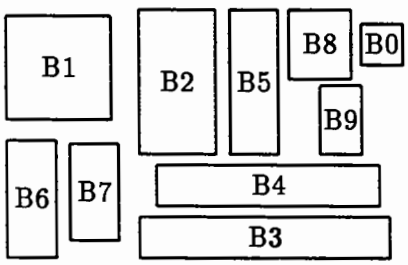
The Minimum-Cost Placements have cost bound = [1980.2, 2016.4]
The set of these placements follows:



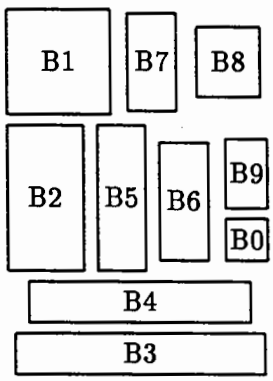
Cost = [1980.2, 2016.4]



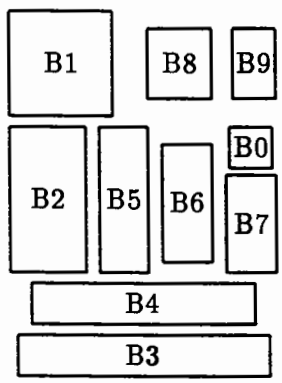
Cost = [1981.9, 2016.4]



Cost = [1982.2, 2016.4]



Cost = [1995.4, 2032.7]



Cost = [1995.6, 2032.7]

Figure 6.7: The Set of Minimum-Cost Placements for a Circuit with 10 Components

Block	Length			Width		
	min	max	avg	min	max	avg
B1	14.92	15.08	15.0	14.92	15.08	15.0
B2	10.94	11.06	11.0	20.89	21.11	21.0
B3	35.82	36.18	36.0	5.97	6.03	6.0
B4	31.84	32.16	32.0	5.97	6.03	6.0
B5	6.96	7.04	7.0	20.89	21.11	21.0
B6	6.96	7.04	7.0	16.91	17.09	17.0
B7	6.96	7.04	7.0	13.93	14.07	14.0
B8	8.95	9.05	9.0	9.95	10.05	10.0
B9	5.97	6.03	6.0	9.95	10.05	10.0
B0	5.97	6.03	6.0	5.97	6.03	6.0

Table 6.3: Block Dimensions for the 10-Component Circuit

Figure 6.8 presents another placement example. This circuit contains eight components with uncertain dimensions (See Table 6.4). Using expected values only, the original branch and bound search algorithm generated over 190,000 partial and complete placements and returned a single solution. The resulting configuration has a area of 4326.0 square mils and appears in the upper left-hand corner of Figure 6.8. Using the interval bounds on the component sizes, the interval branch and bound search algorithm generated almost 310,000 partial and complete placements in the search for the optimal solutions. The results of this search are shown in Figure 6.8. In this case, there are four optimal solutions with a minimum-cost bound of $[4285.4, 4366.8]$ square mils. As shown in the diagram, three of the four best configurations are very similar. In all four solutions, block B6 always appears to the right of block B1 and block B2 always appears in the upper left-hand corner of each layout.

6.3 Proof of the Method

Having presented our interval placement algorithm, we now prove that this method accurately computes bounds on minimum placement area. This proof is divided into two independent parts: computing the interval objective functions and comparing their values. In the first part, we must show that our algorithm for computing the interval objective function is correct. Since the algebra employed to evaluate objective functions is application-specific, this proof applies only to our particular cost function – circuit

The Minimum-Cost Placements have cost bound = [4285.4, 4366.8]
The set of these placements follows:

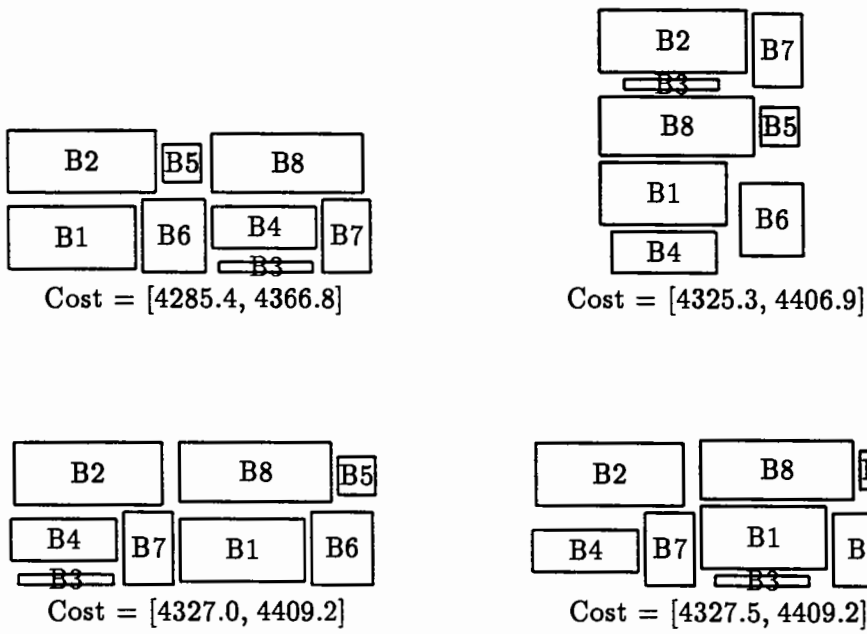


Figure 6.8: The Set of Minimum-Cost Placements for a Circuit with 8 Components

Block	Length			Width		
	min	max	avg	min	max	avg
B1	35.82	36.18	36.0	17.91	18.09	18.0
B2	41.79	42.21	42.0	17.91	18.09	18.0
B3	26.86	27.14	27.0	2.98	3.02	3.0
B4	29.85	30.15	30.0	11.94	12.06	12.0
B5	10.94	11.06	11.0	10.94	11.06	11.0
B6	17.91	18.09	18.0	20.89	21.11	21.0
B7	13.93	14.07	14.0	20.89	21.11	21.0
B8	43.78	44.22	44.0	16.91	17.09	17.0

Table 6.4: Block Dimensions for 8-Component Example

area. To compute the length and width of a given layout, our interval placement program employs an interval longest path algorithm. In Chapter 5, we presented a proof showing that this interval longest path algorithm correctly computes bounds on longest path length. Once the bounds on the dimensions of the layout have been determined, they are multiplied using the interval product operator ($\odot$) to calculate bounds on circuit area. The derivation of this interval product operator and its proof are presented by Moore in [Moor79]. From these proofs, we conclude that our interval objective function correctly computes bounds on circuit area.

Second, we must show that the interval algorithm for computing minimum-cost placements is also correct. Unlike the first part of the analysis, this proof applies to all interval objective functions and also to all interval-based placement strategies. The primary purpose of all placement algorithms is to find the placement that minimizes some cost metric. This reduces to the problem of choosing the smallest of a finite set of elements and may be solved using the algebra $(\Re, min, \infty)$, where $\Re$ also contains the element ∞. Since partial and complete placements may be generated and compared in any order, *min* must be associative and commutative. By definition, all real numbers are less than ∞; therefore, ∞ must be the identity element for *min*. Since the minimum value returned must be a member of the set, we further require that $\Re$ be closed over *min*. Clearly, this algebra has all of these properties.

Now let I be the set of intervals over the augmented set $\Re$. We must show that the extended algebra $(I, MIN, [\infty, \infty])$ exhibits the same properties as the non-interval original. In Appendix A, we demonstrate that the commutative, associative, closure,

and identity $[\infty, \infty]$ properties hold for MIN_*. Since the interval algebra is derived from the non-interval one and exhibits the same properties, we conclude that the algorithm employing this algebra will correctly compute the minimum of a collection of interval elements.

6.4 Performance

Both the interval branch and bound placement algorithm (Intplace) and the non-interval version were implemented in the C programming language and run on a SUN SPARCstation. In this section, we discuss and compare the performance of these two approaches.

6.4.1 Running Time

We begin by analyzing the operating speed of both implementations. Because it is an exhaustive search strategy, the original branch and bound algorithm may generate all possible configurations of the circuit components before finding the best solution; therefore, this algorithm has worst-case running time that is exponential in the number of circuit elements. Since the interval solution is based on the same approach, it too will have a exponential worst-case running time.

Intplace differs from the non-interval version in two main areas: the use of interval operations and the storage of minimum-cost solutions. First, Intplace employs interval operations to perform its calculations and comparisons. For each addition, multiplication, minimum, and maximum operation performed in the original version, we perform two such operations in the interval version. This is particularly evident in the longest path computation used to calculate placement cost; hence, we expect that this function will require twice the time in the interval implementation.

As noted in the previous chapter, the interval longest path algorithm also requires additional time to save the longest paths. Our placement cost calculations, however, need only the longest path lengths, not the paths themselves. By not saving this extraneous path information, we increase the efficiency of the layout cost computations for both interval and non-interval placement algorithms. Without this additional bookkeeping, the longest-path algorithms employed in both placement programs differ only in their choice of interval vs. non-interval operators.

Computing the minimum-cost solution is the second major difference between the interval and non-interval placement algorithms. Instead of returning a single configuration, Intplace returns a set of potential minimum-cost placements. Maintaining this set adds to the operating speed of the interval algorithm. Specifically, whenever the minimum-cost bound is updated in the course of the search, we must insert and delete items from this set. Using a heap to implement the set requires $O(\log n)$ time for each insertion and deletion, where n is the number of placements in the set. The total number of possible complete configurations is exponential in m, the number of circuit elements. In the worst-case, the set of minimum-cost placements could contain all of these possible configurations, yielding insertion and deletion times of $O(m)$. This decreases the operating speed of the interval algorithm by a factor of m in the worst-case. Fortunately, the list of minimum-cost placements is typically small, containing a fraction of the total number of possible complete placements. For example, consider the circuit in Figure 6.7. Of the several billion possible complete placements in the unpruned search tree, the optimal solution contained only five configurations.

We also tested both the interval and non-interval implementations on identical circuit descriptions and recorded their response times. The circuits tested contained up to ten components. For the larger circuits, the programs generated more than 180,000 distinct partial and complete configurations in the search for the best solution. Even so, this figure represents a significantly pruned portion of the almost 98 billion possible configurations in the search space for that problem. In spite of the large numbers of configurations explored, these experiments showed no appreciable difference in the operating speeds of the two algorithms. The apparent similarity in running times is attributed to the observation that most of the computation time is spent generating child placements, rather than evaluating them. The routines for generating child configurations are identical in both implementations.

6.4.2 Interval Width

Since an interval cost is used to prune the search tree in the branch and bound placement algorithm, the width of these intervals will effect the efficiency of the search. Recall that the algorithm prunes a partial placement from the tree when its cost exceeds the current minimum cost. In the interval version, placement cost is greater than minimum cost if and only if the lower bound on placement cost is greater than the

upper bound on minimum cost. Obviously, wider intervals yield larger upper bounds. With a larger upper bound on minimum layout cost, we prune fewer nodes from the search tree and increase the time needed to find the optimal solutions. Even in the non-interval algorithm, pruning efficiency is heavily dependent on circuit structure. In circuits with many blocks of similar shape and size, the standard algorithm may not be able to prune many nodes from the search tree, resulting in a nearly exhaustive enumeration of placement alternatives. For circuit descriptions containing blocks with widely varying sizes, the algorithm can prune a larger portion of the nodes from the search tree. Because pruning efficiency also depends on relative block size, a precise formula indicating the relationship between interval width and pruning efficiency is difficult to determine. We can, however, present several examples illustrating this correlation.

Figure 6.9 depicts the effects of interval width on search efficiency for several circuit descriptions containing five elements. In this graph, interval width is measured by the percent variance from the center of the interval (i.e., interval = center $\pm$ center $\cdot$ variance). Search efficiency is measured by the percentage of the 6565 possible partial and complete configurations generated and examined by the placement algorithm. In this experiment, we started with degenerate intervals and counted the number of partial and complete placements examined in the course of the search. For each circuit description, we varied the width of the intervals by a given percentage and recorded the resulting number of placements examined in locating the optimal solution. The graph displays these results for three different circuit descriptions. The first circuit (denoted by dots with dashed edges) contains blocks with a wide variation of sizes, the second circuit (denoted by triangles and solid lines) contains five blocks with three different sizes, and the third (denoted by diamonds and dotted edges) contains five identical blocks. As the graph shows, the increase in the number of configurations produced was modest for small cost interval widths ($\pm$5%). For large interval widths ($\pm$25%), the algorithm degenerated to an exhaustive enumeration of placement alternatives. Therefore, this algorithm should not be used, if the intervals given for circuit parameters are too wide.

As expected, interval width also affects the size of the optimal placement set. Wider intervals yield wider minimum cost bounds. With a larger upper bound on minimum layout cost, we prune fewer configurations from the optimal solution set. Figure 6.10 presents a graph illustrating the effects of interval width on the size of the optimal placement set for our three circuit examples. In each case, we varied the width of the

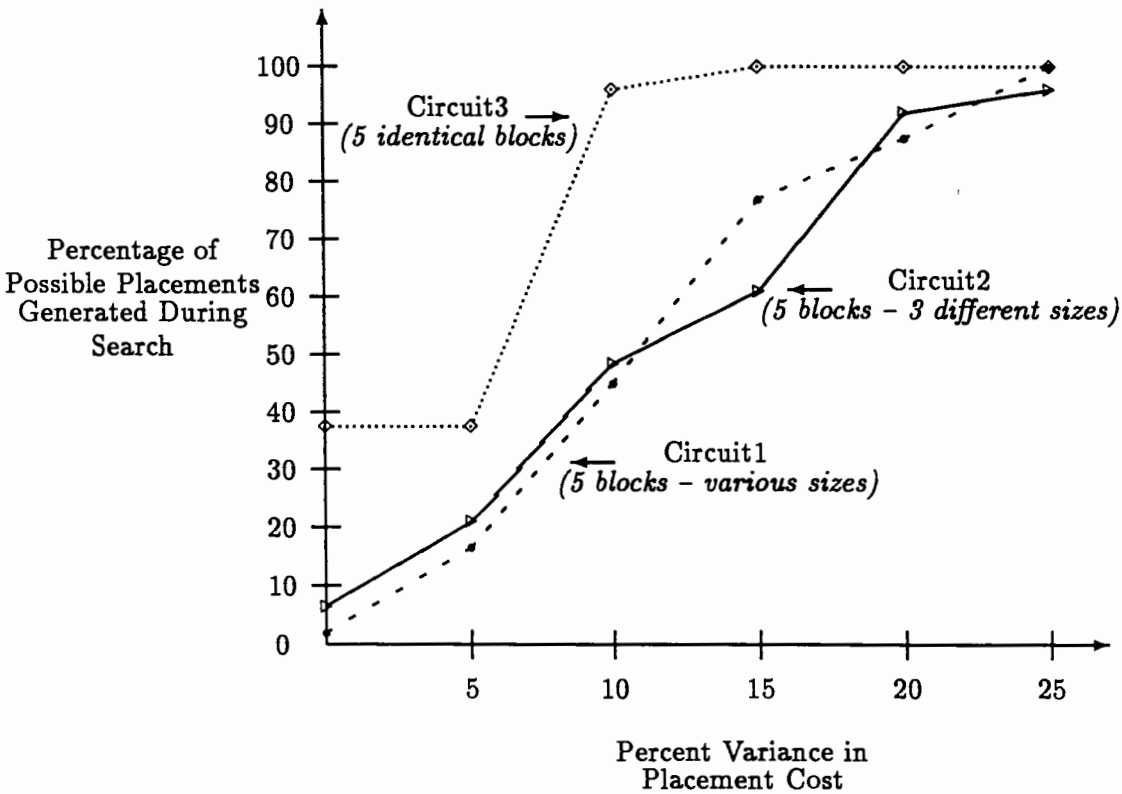


Figure 6.9: Cost Interval Width vs. Pruning Efficiency

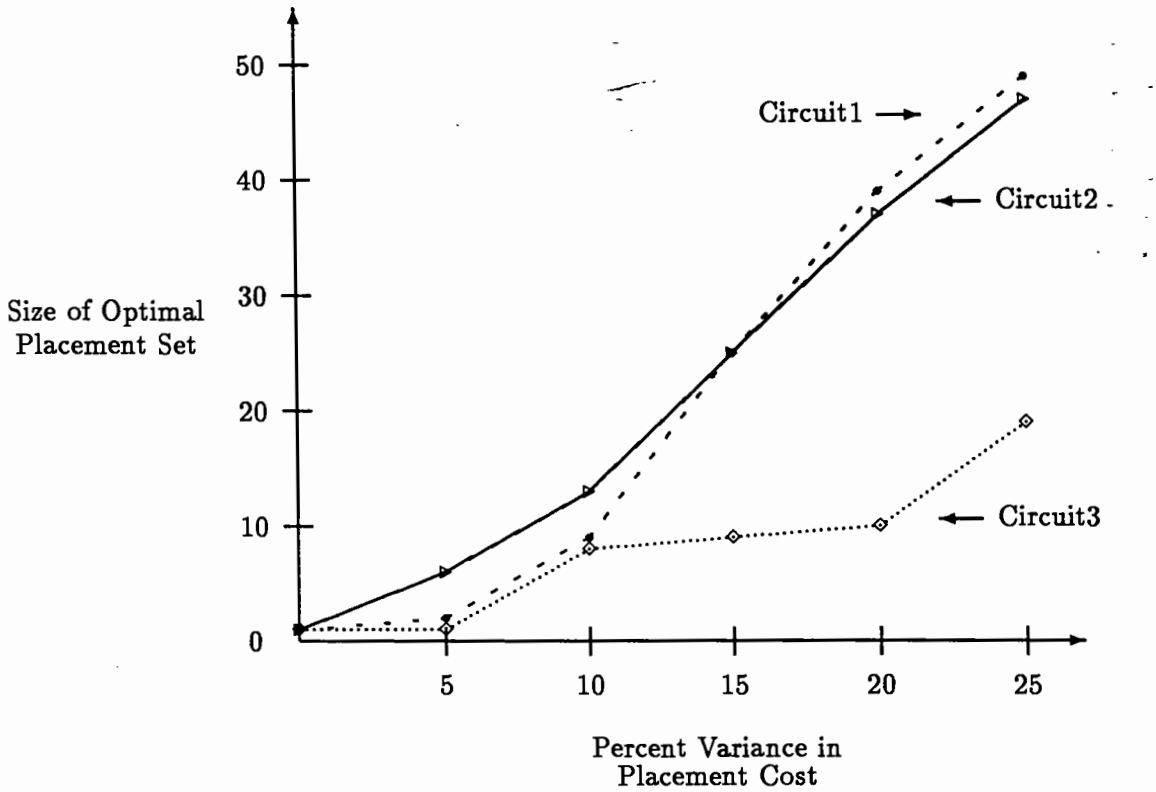


Figure 6.10: Cost Interval Width vs. Optimal Placement Set Size

cost intervals by the given percentage and recorded the number of placements in the final solution set. As shown in the graph, the solution sets for small cost interval widths ($\pm 5\%$) contain few configurations. For larger cost intervals ($\pm 25\%$), the solution sets become too large to be useful. Not surprisingly, the less we know about the circuit, the less we can conclude about the optimal layout.

6.5 Summary

In this chapter, we discussed the problem of uncertain cost functions in placement algorithms. Using several common objective functions, we demonstrated the prevalence of uncertain parameters in placement cost metrics. In spite of this, current placement algorithms use only single-valued estimates in their objective functions. We argued that using the full range of possible values when computing placement costs yields

a more accurate prediction of placement quality, reveals more about the structure of the optimal configurations, and facilitates iterative improvement. For these reasons, we proposed an interval-based approach for modeling uncertainty in placement algorithms. Given the interval range for each uncertain parameter in an objective function, the interval placement algorithm computes and returns bounds on minimum placement cost and a set of placements, each of which is optimal for some combination of input values.

We illustrated this approach by implementing an interval branch and bound placement algorithm, Intplace. We showed how interval operators could be used to compute the interval cost of a placement and to determine the minimum-cost configuration. Although a branch and bound algorithm was chosen, the general principles employed also apply to other placement methods.

To test the performance of our interval approach, we implemented interval and non-interval versions of the placement algorithm. Analysis suggests that the interval program may be slower than the non-interval one by a factor of m, the number of circuit components, in the worst-case. Experiments on several circuits, however, showed no appreciable difference in the operating speeds of the two implementations. We also observed that the width of the intervals affects the efficiency of the search, since wider placement cost bounds reduce the number of placements that are pruned from the search tree. Wider intervals also increase the size of the solution set, as fewer placements can be pruned from the optimal set.

Chapter 7

Conclusions

As the examples in the previous chapters demonstrate, numerical uncertainty is prevalent in VLSI design. Delayed design decisions produce uncertain transistor strengths in switch-level simulation, exploring layout alternatives results in uncertain cell sizes in hierarchical placement, and manufacturing disturbances lead to uncertain electrical characteristics in RC timing analysis. For these and other reasons, the values of many variables used in VLSI analysis cannot be known exactly. In spite of this, most current CAD tools ignore parameter uncertainty and employ only "expected" values in their calculations. Systems that model uncertainty offer several advantages over these conventional approaches. Among their benefits, they provide a more complete characterization of circuit behavior and facilitate iterative improvement. We therefore proposed a new methodology based on interval arithmetic for modeling numerical uncertainty in VLSI design. To illustrate our approach, we applied our technique to several applications, including switch-level simulation, timing analysis, and placement. These applications reveal the strengths and weaknesses of our interval paradigm.

7.1 Advantages of the Method

One strength of the interval method is its simplicity. To create an interval algorithm for each application, we began by deriving the interval extensions of the discrete operators used in the conventional implementation. The basic arithmetic operations (i.e., addition, subtraction, multiplication, and division) were taken from a field of mathematics known as interval arithmetic. Other operations were derived as needed using Equation 1.1. Once derived, the interval operators were incorporated within the algorithm

to create an interval version of it. In two instances, the longest path and placement algorithms, further changes were required. When the longest path algorithm is used for timing verification, it is not sufficient to return bounds on longest path length and a *single* representative longest path. To efficiently reduce circuit delay, we need the *set* of all potential longest paths. Likewise, the interval placement algorithm must return the set of potentially optimal configurations, in addition to computing bounds on optimal placement quality. Storing these extra paths and configurations required. further modifications to the algorithms. Even so, the required additions were simple and intuitive.

The second advantage of the interval approach is its generality. As demonstrated, these techniques can be easily incorporated within many kinds of VLSI CAD tools. They can also be used to model many types of uncertainty, including some that cannot be statistically modeled. Uncertain logic levels and variation due to exploration of design alternatives are two such examples. When an uncertain value is given as a mean value and standard deviation, the 1σ, 2σ, or 3σ endpoints can be used as interval bounds.

Another advantage is speed. For many of our applications, the running time of the interval algorithm is greater than that of its non-interval counterpart by only a small constant. Two exceptions to this rule are the longest path and placement algorithms which return path or layout information, in addition to computing bounds. The longest path algorithm has a worst-case complexity $\log n$ times greater than that of the non-interval version, where n is the number of vertices in the graph. Similarly, the interval placement algorithm is slower by a factor of $2m$ in the worst-case, where m is the number of circuit components. In all applications, experiments comparing the conventional and interval implementations show little difference in operating speed. For RC timing analysis, we also created a Monte Carlo algorithm for comparison. Our tests demonstrate that the interval algorithm runs several orders of magnitude faster than the Monte Carlo version for the same quality results.

7.2 Limitations of the Method

While it has many strengths, the interval paradigm is not without its limitations. As the RC timing example shows, this technique may generate overly conservative bounds for some applications. This is due to an inherent weakness in the algebra: a lack

of additive and multiplicative inverses. For those applications which do not require inverses, the interval algorithms return tight and accurate bounds on the solution. This is true for placement, switch-level simulation, and the longest path computations. For functions that do contain inverses, tight bounds are not guaranteed.

For these cases, we presented two interval-based alternatives for computing better bounds. These methods achieve accuracy at the cost of some extra computational effort. Even so, both methods are significantly faster than Monte Carlo techniques. Like the original interval solution, these approaches are general and can be used on any real-valued arithmetic function $F(x_1, ..., x_n)$.

The first alternative is the centred method presented in Chapter 4. Using this technique, we evaluate the given function F on one particular assignment of values, $c_1, ..., c_n$, (usually the midpoints of all intervals) and create an interval representation of F that produces symmetric bounds centered at $F(c_1, ..., c_n)$ containing the theoretical solution. While there are many types of centred forms, we chose Krawczyk's centred form for its flexibility. This algorithm has a time complexity of $O(sn)$, where s represents the number of arithmetic operations in the function and n represents the number of independent variables.

In Chapter 4, we also presented a bounding method based on interval differential calculus. With this approach, we use derivatives to check the monotonicity of function F with respect to the ranges of each variable. If the function is monotonic in all variables, we can exactly compute bounds on F. If the function is non-monotonic, we subdivide the variable ranges and repeat the analysis on the resulting subintervals. This algorithm has a worst-case time complexity of $O(2^{b+1}sn)$, where b is the number of interval bisections allowed, s is the number of arithmetic operations in F, and n is the number of independent variables. Although it is the slowest of the interval methods presented, it is the only one that provides a measure of the goodness of its bounds.

A second limitation of bounding approaches is that they return performance limits without any indication of the probability of their occurrence. If the probability of these worst-case conditions is small, then these results are again overly conservative. Such results may prompt the design engineer to needlessly overdesign the chip in an effort to prevent events that are unlikely to occur. To avoid this, 1σ or 2σ endpoints may be used in the computations, if 3σ bounds are too pessimistic.

7.3 Future Work

In this thesis, we presented several examples to illustrate the usefulness of our interval approach. These examples are but a sampling of the potential CAD applications for this methodology. Among the many other possible applications are circuit simulation and timing-driven placement.

The necessity of considering the effects of manufacturing disturbances in circuit simulation has long been recognized [Nass84]. As previously mentioned, expensive Monte Carlo techniques are frequently applied in an effort to model the effects of fabrication variations on circuit behavior. Many papers have been written presenting statistical methods for finding correlations between parameters for use in both statistical and worst-case analysis [Dive84]. Because of the computational expense involved in circuit simulation, Monte Carlo techniques are particularly time-consuming. If developed, an accurate interval circuit simulator could produce bounds in much less time.

Another possible application is timing-driven placement. Here, timing considerations are used when computing the layout of a circuit. Configurations containing long wires with large delays can severely degrade the performance of the chip. Timing-driven placement algorithms use information about the critical paths through a circuit to generate a layout that minimizes the wire delays along this path. As we have already shown, the component and wire delays employed in this computation are subject to variation, so interval longest path algorithms can be applied here to locate critical paths through circuits with uncertain component and wire delays. Once computed, these critical paths can be used in several ways to direct the placement. One solution [Jack86] gives higher weights to nets along the critical path and employs a min-cut partitioning scheme to assign circuit components to specific regions on the layout. Since the delays used to determine the weights are uncertain, we would need to create an algorithm for min-cut partitioning on a graph with interval-weighted edges.

In addition to timing-dependent methods, there are many other objective functions for placement that contain uncertain values. In Chapter 6, we cited several examples. While we implemented one possible interval objective function, there are many others to consider.

In addition to exploring other applications for our interval methods, another area for future research is statistical algebras. The goal here is to derive a simple set of

rules for performing arithmetic operations on (presumably Gaussian) statistical distributions. If such algebras were to be developed, they could be incorporated into existing algorithms in the same manner as intervals. The algorithms employing these algebras would then return the probability distribution of the results. This methodology would complement our interval approaches, providing a fast method for generating probability distributions, when worst- and best-case bounds alone are insufficient. With the exception of Hitchcock's rule for adding statistical distributions [Hitc82], such algebraic operators have yet to be developed.

7.4 Closing Remarks

In conclusion, there are some situations for which statistical measures are more appropriate, but bounding methods are simpler and faster. We therefore advocate using our interval methods first to produce preliminary estimates of behavior. Then, if further information is required, the computationally-expensive statistical methods, such as Monte Carlo simulation, should be employed. In situations where statistical methods are not applicable, our interval approach offers a simple, fast, and provably correct solution.

Appendix A

Properties of the Operators

In this appendix, we prove that the associative, commutative, identity, and closure properties hold for the interval minimum (MIN) and maximum (MAX) operators. Let I^+ be the set of intervals over the augmented set of real numbers, $\Re \cup \{\infty\} \cup \{-\infty\}$. Details are given for the MIN operator only. The proofs for the MAX operator are analogous.

Proof.

1. MIN is associative

   For all $[a,b],[c,d],[e,f] \in I^+$, we must show that

   $$MIN(MIN([a,b],[c,d]),[e,f]) = MIN([a,b],MIN([c,d],[e,f])).$$

   By the definition of MIN, we know that

   $$\begin{aligned} MIN(MIN([a,b],[c,d]),[e,f]) &= [min(min(a,c),e), min(min(b,d),f)] \\ MIN([a,b],MIN([c,d],[e,f])) &= [min(a,min(c,e)), min(b,min(d,f))]. \end{aligned}$$

   Because min is associative,

   $$\begin{aligned} min(min(a,c),e) &= min(a,min(c,e)) \; and \\ min(min(b,d),f) &= min(b,min(d,f)). \end{aligned}$$

   Hence, MIN is associative. □

2. MIN is commutative

   For all $[a,b],[c,d] \in I^+$, we must show $MIN([a,b],[c,d]) = MIN([c,d],[a,b])$.

Expanding the expression using the definition of MIN yields

$$\begin{aligned} MIN([a,b],[c,d]) &= [min(a,c), min(b,d)] \text{ and} \\ MIN([c,d],[a,b]) &= [min(c,a), min(d,b)]. \end{aligned}$$

Since min is commutative, these expressions are equivalent. □

3. MIN has identity element $[\infty, \infty]$
   For all $[a,b] \in I^+$, we must show that $MIN([a,b],[\infty,\infty]) = [a,b]$. This follows directly from the definition of MIN and that ∞ is the identity of min. □

4. I^+ is closed with respect to MIN
   For all $[a,b],[c,d] \in I^+$, we must show that $MIN([a,b],[c,d]) \in I^+$. Applying the definition of MIN, $MIN([a,b],[c,d]) = [min(a,c), min(b,d)]$. Since augmented $\Re$ is closed with respect to min, then $min(a,c)$ and $min(b,d) \in \Re \cup \{\infty\} \cup \{-\infty\}$. Now all that remains is to show that $min(a,c) \leq min(b,d)$. Clearly this is true, since by definition $[a,b]$ implies $a \leq b$ and $[c,d]$ implies $c \leq d$; therefore, $[min(a,c), min(b,d)] \in I^+$. □

Appendix B

Common Interval Operations

The following is the C language code for the common interval operations employed in this thesis. They are interval addition (ADD), subtraction (SUB), multiplication (MULT), division (DIV), minimum (MIN), and maximum (MAX). All assume the following definition of an *interval*:

```
typedef struct
  {
  float min;          /* the lower bound of the range */
  float max;          /* the upper bound of the range */
  }  INTERVAL;
```

B.1 Addition

Given two intervals over the real numbers, this routine performs interval addition on these ranges and returns their sum.

```
INTERVAL ADD(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min + range2.min;
  result.max = range1.max + range2.max;
  return(result);
  }
```

B.2 Subtraction

Given two intervals over the real numbers, this routine performs interval subtraction on these ranges and returns their difference.

```
INTERVAL SUB(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min - range2.max;
  result.max = range1.max - range2.min;
  return(result);
  }
```

B.3 Multiplication

Given two intervals over the positive real numbers, this routine performs interval multiplication on these ranges and returns their product.

```
INTERVAL MULT(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min * range2.min;
  result.max = range1.max * range2.max;
  return(result);
  }
```

B.4 Division

Given two intervals over the positive real numbers, this routine performs interval division on these ranges and returns their quotient.

```
INTERVAL DIV(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min / range2.max;
  result.max = range1.max / range2.min;
  return(result);
  }
```

B.5 Minimum

Given two intervals over the real numbers, this routine computes and returns their interval minimum.

```
INTERVAL MIN(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = (range1.min < range2.min) ? range1.min : range2.min;
  result.max = (range1.max < range2.max) ? range1.max : range2.max;
  return(result);
  }
```

B.6 Maximum

Given two intervals over the real numbers, this routine computes and returns their interval maximum.

```
INTERVAL MAX(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = (range1.min > range2.min) ? range1.min : range2.min;
  result.max = (range1.max > range2.max) ? range1.max : range2.max;
  return(result);
  }
```

Appendix C

Interval Differential Operations

In this appendix, we present the C language code for the interval differential operations defined in Chapter 4. These operations use the interval algebraic operators ADD, SUB, MULT, and DIV to perform their calculations. They also employ the following definitions of an *interval* and an *interval operand*:

```
typedef struct
  {
  float min;          /* the lower bound of the range */
  float max;          /* the upper bound of the range */
  }  INTERVAL;

typedef struct
  {
  INTERVAL value;     /* the value of the operand */
  INTERVAL *deriv;    /* the vector of partial derivatives */
  }  IOPERAND;
```

C.1 Addition

Given two interval operands, this function computes and returns the interval sum of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DADD (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL ADD();

                                          /* compute the interval sum */
  result.value = ADD(o1.value, o2.value);
                                           /* create derivative array */
  create_deriv_array(&(result.deriv));
                                        /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = ADD(o1.deriv[i], o2.deriv[i]);
                                                  /* return the result */
  return(result);
  }
```

C.2 Subtraction

Given two interval operands, this function computes and returns the interval difference of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DSUB (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL SUB();

                                    /* compute the interval difference */
  result.value = SUB(o1.value, o2.value);
                                              /* create derivative array */
  create_deriv_array(&(result.deriv));
                                          /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = SUB(o1.deriv[i], o2.deriv[i]);
                                                    /* return the result */
  return(result);
  }
```

C.3 Multiplication

Given two interval operands, this function computes and returns the interval product of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DMULT (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL ADD(), MULT();

                                        /* compute the interval product */
  result.value = MULT(o1.value, o2.value);
                                              /* create derivative array */
  create_deriv_array(&(result.deriv));
                                         /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = ADD(MULT(o1.value, o2.deriv[i]),
                          MULT(o2.value, o1.deriv[i]));
                                                 /* return the result */
  return(result);
  }
```

C.4 Division

Given two interval operands, this function computes and returns the interval quotient of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DDIV (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL SUB(), MULT(), DIV();

                                    /* compute the interval quotient */
  result.value = DIV(o1.value, o2.value);
                                          /* create derivative array */
  create_deriv_array(&(result.deriv));
                                      /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = DIV(SUB(o1.deriv[i],
                              MULT(result.value, o2.deriv[i])),
                          o2.value);
                                                /* return the result */
  return(result);
  }
```

Bibliography

[Aho74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[Aho83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[Breu77] Melvin A. Breuer. A class of min-cut placement algorithms. In *14th IEEE Design Automation Conference*, pages 284–290, 1977.

[Brya80] Randal E. Bryant. An algorithm for MOS logic simulation. *Lambda*, 1(3):46–53, 1980.

[Brya84] Randal E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, pages 160–177, February 1984.

[Byrd85] Richard H. Byrd, Gary D. Hachtel, Micheal R. Lightner, and Michel H. Heydemann. Switch-level simulation: Models, theory, and algorithms. In A. Sangiovanni-Vincentelli, editor, *Computer Aided-Design of VLSI Circuits and Systems*, pages 93–148. JAI Press, Greenwich, CT, 1985.

[Chow88] Paul Chow and Mark Horowitz. The design and testing of MIPS-X. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 95–114. MIT Press, 1988.

[Cies84] Maciej J. Ciesielski and Edwin Kinnen. Digraph relaxation for CAD layouts of cell based integrated circuits. In *1983-84 Computer Science and Computer Engineering Research Review.* University of Rochester, 1983-84.

[Dai89] Wayne Wei-Ming Dai, Bernhard Eschermann, Ernest S. Kuh, and Massoud Pedram. Hierarchical placement and floorplanning in BEAR. *IEEE Transactions on Computer-Aided Design*, 8(12):1335–1349, December 1989.

[Dive84] Dileep A. Divekar. DC statistical circuit analysis for bipolar IC's using parameter correlations – an experimental example. *IEEE Transactions on Computer-Aided Design*, 3(1):101–103, January 1984.

[Fidu82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th IEEE Design Automation Conference*, pages 175–181, 1982.

[Gecs87] Jan Gecsei and Eduard Cerny. Self-adjusting networks for VLSI simulation. *IEEE Transactions on Computers*, 36(9):1114–1120, September 1987.

[Hajj87] Ibraham Hajj and Daniel Saab. Switch-level logic simulation of digital bipolar circuits. *IEEE Transactions on Computer-Aided Design*, 6(2):251–258, March 1987.

[Hard88] Bill Harding. New analysis techniques pave the way for analog simulation. *Computer Design*, pages 37–41, September 1988.

[Haye86] John P. Hayes. Uncertainty, energy, and multiple-valued logics. *IEEE Transactions on Computers*, 35(2):107–114, February 1986.

[Hitc82] Robert B. Hitchcock Sr. Timing verification and the timing analysis program. In *19th IEEE Design Automation Conference*, pages 594–603, 1982.

[Hugh89] Richard Hughey and Daniel P. Lopresti. The B-SYS programmable systolic array. Technical Report CS-89-32, Department of Computer Science, Brown University, 1989.

[Jack86] Micheal Jackson and Ernest Kuh. Performance-driven placement of cell based IC's. In *26th IEEE Design Automation Conference*, pages 370–375, 1986.

[Joup83] Norman P. Jouppi. Timing analysis for nMOS VLSI. In *20th IEEE Design Automation Conference*, pages 411–418, 1983.

[Kern70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[Kirk83] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Kuli81] Ulrich Kulisch and Willard Miranker. *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[McWi80] Thomas M. McWilliams. Verification of timing constraints on large digital systems. In *17th IEEE Design Automation Conference*, pages 139–147, 1980.

[Moor79] Ramon E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.

[MS89] Malgorzata Marek-Sadowska and Shen P. Lin. Timing driven placement. In *IEEE International Conference on Computer-Aided Design*, pages 94–97, November 1989.

[Nass84] Sani R. Nassif, Andrzej J. Strojwas, and Stephen W. Director. Fabrics II: A statistically based IC fabrication process simulator. *IEEE Transactions on Computer-Aided Design*, 3(1):40–46, January 1984.

[Oust83] John K. Ousterhout. Crystal: A timing analyzer for nMOS VLSI circuits. In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 57–70. Computer Science Press, 1983.

[Oust84] John K. Ousterhout. Switch-level delay models for digital MOS VLSI. In *21st IEEE Design Automation Conference*, 1984.

[Oust86] John K. Ousterhout. Using crystal for timing analysis. Technical Report UCB/CSD 86/272, University of California at Berkeley, 1986.

[Prea79] B.T. Preas and W.M. van Cleemput. Placement algorithms for arbitrarily shaped blocks. In *16th IEEE Design Automation Conference*, pages 474–480, 1979.

[Prea86] Bryan T. Preas and Patrick G. Karger. Automatic placement: A review of current techniques. In *23rd IEEE Design Automation Conference*, pages 622–629, 1986.

[Rall86] L.B. Rall. Improved interval bounds for ranges of functions. In K. Nickel, editor, *Interval Mathematics 1985*, pages 143–155. Springer-Verlag, Berlin, 1986.

[Rama83a] Vijaya Ramachandran. An improved switch-level simulator for MOS circuits. In *20th IEEE Design Automation Conference*, pages 293–299, 1983.

[Rama83b] Vijaya Ramachandran. *Studies in VLSI Layout and Simulation.* PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, August 1983.

[Rats84] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions.* Ellis Horwood Limited, Chichester, 1984.

[Rubi83] Jorge Rubinstein, Paul Penfield Jr., and Mark A. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on Computer-Aided Design*, 2(3):202–211, July 1983.

[Sobo75] I.M. Sobol. *The Monte Carlo Method.* Mir Publishers, Moscow, 1975.

[Sund87] Rolf Sundblad and Christer Svensson. Fully dynamic simulation of CMOS circuits. *IEEE Transactions on Computer-Aided Design*, 6(2):282–289, March 1987.

[Tarj83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[Tric86] Micheal T. Trick and Stephen W. Director. LASSIE: Structure to layout for behavioral synthesis tools. In *26th IEEE Design Automation Conference*, pages 104–109, 1986.

[Wall86] David E. Wallace and Carlo H. Sequin. Plug-in timing models for an abstract timing verifier. In *23rd IEEE Design Automation Conference*, pages 683–689, 1986.

[Webs84] *Webster's II New Riverside Dictionary.* Berkeley Publishing Group, New York, 1984.

[Zuko86] Charles A. Zukowski. *The Bounding Approach to VLSI Circuit Simulation.* Kluwer Academic Publishers, Boston, 1986.

An Approach To Uncertainty In VLSI Design

Cheryl Lynn Harkness
Ph.D Dissertation

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-90-15
May 1991

AN APPROACH TO UNCERTAINTY IN VLSI DESIGN

by
Cheryl Lynn Harkness
B. S., Union College, 1984
Sc. M., Brown University, 1986

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1991

Vita

Cheryl Harkness was born in Troy, New York on August 3, 1962. She graduated summa cum laude and with departmental honors from Union College, Schenectady, New York in 1984 receiving a B.S. in Computer Science. Throughout her undergraduate career, Ms. Harkness spent her summers working first as a business applications programmer for an independent software firm and then as a summer intern with the VLSI Design Division at General Electric's Research and Development Center.

Ms. Harkness entered the graduate school at Brown University in September 1984, received her Sc.M. in Computer Science in May 1986, and was admitted to doctoral candidacy in May of the same year. While in graduate school, she has served as a research and teaching assistant for the Computer Science Department. In addition, she spent the summer of 1987 working on timing verification systems at Digital Equipment Corporation in Hudson, Massachusetts. With her advisor, she has published several technical reports on topics related to her thesis. She has co-authored a paper, entitled "Modeling Uncertainty in RC Timing Analysis", which was published in the 1989 *Proceedings of the IEEE International Conference on Computer-Aided Design* and was presented at the conference in November 1989.

Upon completion of her Ph.D. at Brown University, Ms. Harkness will join the Design Technology Center at Hewlett-Packard in Santa Clara, California. She has accepted a position as Software Development Engineer designing hierarchical simulation and verification tools. Her research interests include computer-aided design for VLSI, analysis of algorithms, computational geometry, and computer graphics. She is a member of Phi Beta Kappa, Sigma Xi, and the IEEE.

Abstract

Computer-aided design (CAD) tools take a circuit description and analyze the design to predict circuit behavior. Although these descriptions take many forms ranging from behavioral specifications to mask layouts, all of them contain bits of quantitative information about the circuit (e.g. physical dimensions, logic values, threshold voltages, parasitic capacitances). Often the precise values of these circuit characteristics cannot be determined before the layout is completed and/or the chip is fabricated. Even though the exact values are not known, estimates in the form of bounds, probability distributions, or average values can be obtained for these uncertain circuit parameters. Most current CAD tools employ only "expected" values in their calculations, returning one of several possible outcomes. A fundamental issue is how the circuit will behave with other combinations of possible values.

In this thesis, we present a general framework based on interval algebra for modeling uncertainty in VLSI. Our approach is to represent uncertain parameters as bounding intervals and to develop algebras for manipulating these ranges. We explore applications for this framework in several areas of VLSI design, including switch-level simulation, timing analysis, and placement. For each area, we create an application-specific interval algebra and incorporate this algebra within an existing algorithm for solving the problem. The result is a new interval version of the algorithm that uses the full range of values for each uncertain parameter in its computations and returns all possible solutions.

The different applications reveal the strengths and the weaknesses of the interval paradigm. Among its many advantages are its versatility, simplicity, and speed. Its main liability is that it may produce overly-conservative bounds for some applications. For these cases, we provide alternative interval-based techniques which improve the bounds, but require more computation time.

Acknowledgements

Although this document bears the name of a single author, it could not have been successfully completed without the help and assistance of many others. I would like to express my gratitude to all those who have contributed to this effort and made my stay here at Brown more enjoyable.

My advisor, Daniel Lopresti, deserves many thanks for his guidance and direction over the years. His suggestion that I look into interval arithmetic inspired this dissertation. His thoroughness as a reader and critic is also much appreciated. Through his meticulous efforts, this thesis was greatly improved.

I am indebted to my readers, John Savage and Gerard Baudet, for their many insightful comments. Special thanks to Gerard for sparking my interest in VLSI CAD, for advising me during my first years of graduate school, and for frequent visits back to Brown to check on my progress.

During my summer with Digital, Karem Sakallah introduced me to timing verification. His influence is evident in Chapters 3 through 5 of this thesis. I am also grateful to him for his advice and encouragement. My summer in industry was invaluable, not only for the experience gained, but also for the people that I met. Thanks to Darrylle, Sundar, Chandra, Karem, and Costas for a truly memorable summer.

Over the years at Brown, I have had the privilege of meeting many wonderful people. Chris Chiu, Phillip Hubbard, Richard Hughey, Richard Manning, Swaminathan Manohar, Alex and Susie Nacar, Scott Oaks, Robert Ravenscroft, Cathy Schevon, and John Stasko have all enriched my life with their friendship during these past six years. I am also grateful to Jim and Traudie Campbell for "adopting" me, for providing a haven away from academic life, and for making sure that I was always well-fed. Rob Ravenscroft warrants special thanks for his enduring friendship, for helping me through the tough times, and for sustaining me with some of the most delicious baked goodies

and culinary delights imaginable.

Ευχαριστῶ πάρα πολύ to Lucia Athanassaki for patiently teaching me Modern Greek. Learning Greek under her tutelage was a most pleasant and rewarding diversion. I only regret that I could not continue my lessons this last semester. Lucia, I wish you success with your thesis and career.

I am also thankful for the loving support of my significant other, Costas Spanos. I am indebted to him for his sound advice and inspiration, for his friendship and love, and for a constant supply of chocolate and teddy bears. Thanks also for two particularly unforgettable vacations. I don't know what I would have done without you.

The contribution of my family – brother, sister, parents, and grandparents – is immeasurable. Their support has meant more to me than they could ever imagine. Without my parents, Robert and Carol Harkness, none of this would have been possible. They have always stressed the importance of education and have encouraged their children to make the most of the opportunities before them. Throughout my long college and graduate career, they have sustained me with their unfailing love, encouragement, and financial assistance. This thesis is as much a product of their efforts as my own and I dedicate it to them.

CLH
Providence, RI

Contents

List of Tables

List of Figures

Chapter 1

Introduction

By analyzing a circuit description, VLSI design tools determine the features (i.e., functionality, speed, structure) of the resulting chip. Although circuit descriptions take many forms ranging from high-level behavioral specifications to mask layouts, all contain bits of quantitative information about the circuit (e.g., physical dimensions, logic values, threshold voltages, parasitic capacitances). As a result of postponed design decisions, manufacturing process disturbances, and many other factors, the precise values of these circuit parameters often cannot be determined in advance. For example, in top-down design, we begin with a purely functional specification of desired behavior and gradually refine this description to produce a structural implementation that realizes the given function. Throughout this process, we test portions of the design to analyze performance, to check functionality, or to explore alternatives. Because the design is incomplete at intermediate stages, all the implementation details needed for the analysis may not be known exactly. Even at the mask-level uncertainty exists, since the electrical properties of the circuit are subject to uncontrollable variations in the fabrication process. In these cases, a circuit parameter may assume any one of several possible values.

While the presence of uncertainty in VLSI design is widely recognized, most current computer-aided design (CAD) tools use only "expected" values in their calculations. They analyze one particular instantiation of the design, ignoring the other possibilities (See Figure 1.1). The most notable specific exceptions are found in timing verification, multiple-valued logics, and process simulation. Unfortunately, the only *general*

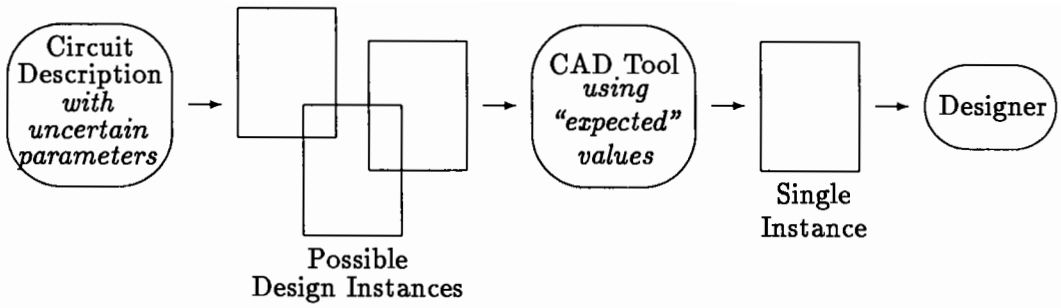


Figure 1.1: Current CAD Tools Ignore Uncertainty

method currently available for modeling parameter uncertainty is the computationally-expensive Monte Carlo technique. The goal of this research is to develop a general and efficient alternative to modeling uncertainty.

The major contribution of this thesis is a new theoretical framework based on interval algebra for modeling uncertainty in VLSI design. Our approach is to model uncertain parameters as intervals and to create interval algebras for manipulating these uncertain values. Once created, these algebras are embedded within existing CAD algorithms, thus creating new interval versions of the algorithms that compute bounds on the results when given interval inputs. We explore applications for this framework in several areas of VLSI design, including simulation, timing analysis, and placement. These applications not only illustrate the approach, but also reveal the strengths and weaknesses of the interval paradigm.

1.1 Uncertainty

Uncertainty arises when some physical aspect of the design (e.g., logic values, capacitances, resistances, physical dimensions, rise and fall times, wire length) needed to perform a phase of the analysis is not precisely known. *Webster's Dictionary* [Webs84, p. 744] gives the following three definitions for the word *uncertain*:

1. "Not yet determined or settled."

2. "Not capable of being known in advance."

3. "Likely to change: variable."

These definitions reveal three sources of uncertainty in VLSI design.

The first definition of an uncertain value is one that is "not yet determined or settled." In VLSI design, such uncertainty may result from delaying design decisions or exploring alternatives. In any circuit description, there are elements that can be implemented in more than one way and often there are many versions of a single part. One of the designer's problems is to choose the best implementation to realize his design. Figure 1.2 illustrates this principle showing two alternative implementations of an adder. Each different implementation will have its own unique set of characteristics (e.g., size, delay). If parts of the design are analyzed before all of these choices are made, we introduce some uncertainty about these characteristics. In top-down design, some implementation details may not be available when testing the circuit at an initial or intermediate stage. One example is estimating wire length or channel width while exploring placement alternatives. To evaluate placement quality, we may need to know wire length or channel width. Unfortunately, these values can only be precisely determined after full routing. Because routing is computationally expensive, it is not practical to perform this calculation for each possible placement. Instead, we use estimates of wire length to determine placement quality. Another example is preliminary behavioral simulation of a switch-level schematic before sizing all the transistors. To simulate this circuit, we need to know the relative strengths of the transistors, a function of transistor size. In this case, we must estimate transistor strengths.

Webster's second definition for uncertainty is "not capable of being known in advance." This type of uncertainty arises as a consequence of uncontrollable disturbances in the manufacturing process used to fabricate a chip. During fabrication, even slight aberrations in any of a multitude of process parameters (e.g., exposure time, temperature, gas pressure) produce variations in the electrical and physical characteristics of the resulting circuit (See Figure 1.3); thus, resistances and capacitances may vary from wafer to wafer and even from chip to chip on a single wafer. The circuit specifications given for each technology are statistical averages based on measurements taken from test runs of a technology. These values are thus only approximations to the real values. The only way to determine the actual values for each chip (and each device on the chip) is to measure them after fabrication, but in order to test designs and predict behavior before fabrication, we must estimate these transistor and interconnect characteristics.

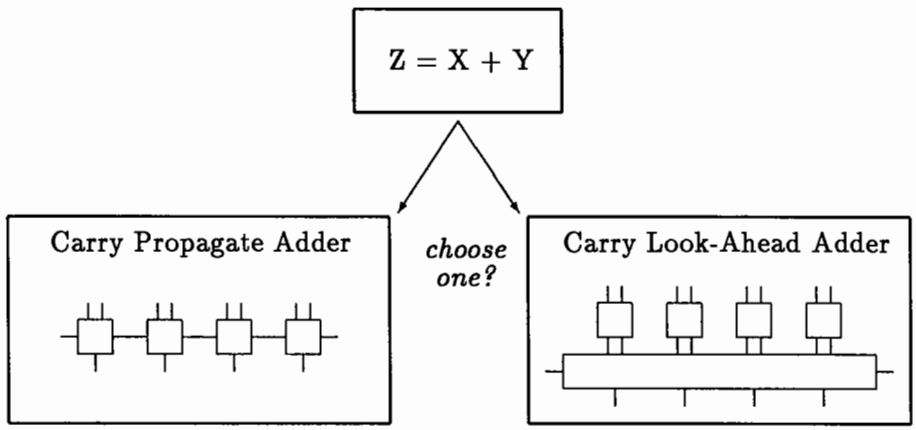


Figure 1.2: Choosing the Best Implementation

Figure 1.3: The Effects of Manufacturing Disturbances

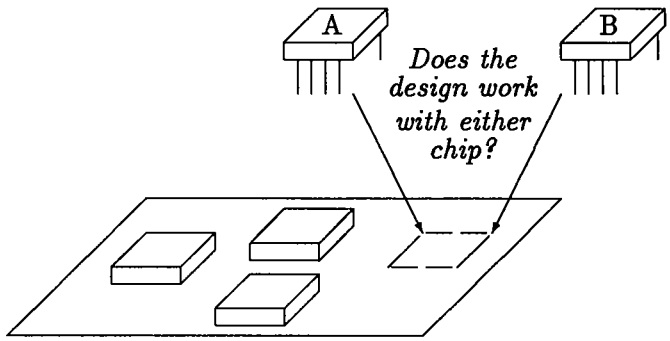


Figure 1.4: Designing for Flexibility

Electrical characteristics are not the only estimates that a designer commonly sees. In semi-custom design, predesigned standard cells are used as building blocks to create larger, more complex circuits. These cells too have estimated performance specifications. Even the specifications given for off-the-shelf parts are estimates. For example, the rise and fall times given for a chip are averages. The actual rise and fall times of each chip will lie within some acceptable tolerance of these given values.

The final definition of uncertain is "likely to change" or "variable." This type of uncertainty occurs when designing for flexibility. Under this scenario, the engineer designs a circuit with the express intention of being able to substitute functionally-equivalent pieces into the final design. As we stated earlier, each piece of a circuit description may be implemented in a variety of ways. But instead of choosing a single implementation for a portion of the circuit, the design engineer could build it so that several different implementations could be inserted into the design and it would still function correctly (See Figure 1.4). For example, consider designing a circuit board using off-the-shelf parts. Two vendors may manufacture chips with identical functionality, but perhaps slightly different operating speeds. For economic reasons, the designer may want to be able to use either chip in his/her final design. Because the implementations have slightly different operating characteristics, the values used in analysis will be uncertain.

These examples illustrate the prevalence of uncertain values in circuit design. As shown, uncertainty exists in all types of circuit descriptions from system-level specifications to technology-dependent layouts. In spite of this, most current design aids ignore

uncertainty and employ only single-valued estimates in their calculations. These programs analyze only one particular set of values and return a single estimate of behavior, but we require methods that test the full range of possible values for each parameter.

1.2 Motivation

What advantages does modeling uncertainty offer? Methods that account for parameter uncertainty are necessary for several reasons. They return more credible results, provide a more complete characterization of circuit behavior, and facilitate iterative improvement.

The first of these benefits is credibility. Using expected values in circuit analysis yields only one of many possible solutions. Unless the true values of all parameters match the average values chosen, the observed behavior of the circuit will differ from the predicted performance. In practice, the probability of exactly predicting the true result is very small. In contrast, approaches that model uncertainty return a range of potential behaviors for the circuit, instead of an exact solution. With these methods, it is more certain that the measured result will fall within the predicted range.

A more complete characterization of the circuit is the second advantage. Using the average values alone may yield an expected solution, but gives no indication of the range of possible solutions that can result. As stated in a recent article:

> Statistical analysis recognizes that components used in a design normally have a range of acceptable values, typically varying between 1 and 20 percent from the stated value. When a breadboard is built and tested, one possible combination of values is observed. Similarly, initial simulations use the exact values shown on the schematic. But the question is whether the design will continue to work satisfactorily with other possible combinations. [Hard88, p.39]

Analysis methods that incorporate uncertainty generate the full range of possible outputs, not just the "expected" result. Depending on the method employed, the range of solutions may be represented as bounds or as a statistical distribution. An added benefit of uncertain approaches is that they generate best- and worst-case bounds automatically.

The third advantage of such methods is that they facilitate efficient iterative design.

Chip design is an iterative process of creation, testing, and modification. Mistakes in the design can require costly re-evaluation. In describing the testing of the MIPS-X chip, Chow and Horowitz state:

> A major problem with the tool set was the amount of time it took to find a bug, fix it and rerun the simulation. During the end of the design, it would take over three CPU hours on a DEC Microvax II to generate and initialize a new simulation run after the layout was changed. [Chow88, pp. 104-5]

Armed with CAD tools that handle uncertain values, the design engineer can test portions of the circuit earlier and more completely. This allows him/her to locate and eliminate some errors much earlier in the design process, when changes are less costly.

Uncertain approaches not only allow the engineer to test the circuit earlier, but can also increase efficiency in other ways. With tools that support only single-valued estimates, the designer must repeat the analysis every time these values change. Bounding results, however, may be reusable. As long as the updated values or ranges still fall within the original ranges given for a variable, the results of previous computations are still valid. This is particularly useful when the source of uncertainty is delayed design decisions. In these cases, the bounds are typically refined as choices are made. Here we can benefit from the results of previous computations and may only have to re-evaluate a small subset of the alternatives, instead of repeating the entire analysis.

1.3 Background

While most current design aids ignore parameter variability, there are several noteworthy exceptions. Foremost among these is Hayes' contribution to multiple-valued logics. In [Haye86], he defines uncertainty as it pertains to discrete logic levels in digital circuit design and introduces a formalism for creating multiple-valued logics. He begins with a set S of n discrete logic levels and defines an uncertain variable (called a *u-value*) as the subset of S containing all possible logic levels that the unknown could assume. For example, if $S = \{0, 1, 2\}$, then $U = \{0, 1\}$ is a u-value over S. Given any two-input operator over the set S, this operator can be extended to handle u-values by applying it to each element of the u-value and taking the union of the results. For example, let S be the binary set $\{0, 1\}$ and consider the boolean operator AND. If X represents the u-value $\{0, 1\}$, then $AND(1, X) = AND(\{1\}, \{0, 1\}) = AND(1, 0) \cup AND(1, 1) =$

$\{0,1\} = X$. Hayes also presents extensions for creating and manipulating strings of u-values, called *p-values*. U-value operators can be further extended to handle p-values by applying the operator pairwise to corresponding vector elements and concatenating the results. For example, $AND(1X,01) = AND(1,0)\ AND(X,1) = 0X$. Using these basic principles, Hayes creates increasingly complex logics from the basic binary set $\{0,1\}$ and gives several examples of logics in common use today.

Hayes' treatment of uncertainty is complete and rigorous as it applies to multiple-valued logics. While his methods are well-suited to logic values, they are not practical as a general method for modeling uncertainty. One reason is that Hayes' operators are computed by the exhaustive enumeration of all possible combinations of operand values. Because digital logics typically consist of small, finite sets of discrete values, exhaustive enumeration is possible. For other applications, the uncertain parameters may be defined over an infinite set (such as $\Re$) and thus may have an infinite number of possible values. In such cases, Hayes' methods cannot be applied. Even so, some of his basic ideas still hold and we incorporate them in our solution.

A second application involving unknown logic values is presented by Bryant in [Brya84]. He describes a switch-level simulator, MOSSIM II, designed to handle unknown transistor state. A transistor may be in any one of three possible states: 0 (open or nonconducting), 1 (closed or conducting), or X (indeterminate). Bryant seeks an efficient method for handling indeterminate transistor state without having to evaluate all 2^k possible assignments of values to the k indeterminate-state transistors. To solve this problem, he presents a *signal flow analysis* algorithm and provides a proof of his method. Other switch-level simulators use similar methods to address the problem of unknown transistor state, including [Byrd85, Gecs87, Hajj87, Rama83a, Rama83b, Sund87]. In this thesis, we introduce another method for modeling uncertainty and apply this technique to create an extension of Bryant's algorithm that models uncertain transistor strength and node size as well.

Systems that model uncertainty also exist for timing verification of digital circuits. These systems may be divided into two classes: bounding and statistical approaches. McWilliams' timing verifier SCALD[McWi80] provides an example of the bounding approach. SCALD is a timing verifier for synchronous, sequential, digital circuits that uses the minimum and maximum propagation delays of circuit components (e.g., logic gates, flip flops) and wires to compute bounds on circuit delay.

Representing the statistical approach, Hitchcock presents a statistically-based timing verifier, TA[Hitc82]. In TA, the delay of each circuit component is represented by a mean value and a standard deviation. The delay of the circuit is represented as a statistical distribution calculated from the delays of the components. Mean arrival time, μ, is computed by summing the mean delays along the path taken and the standard deviation, σ, is found by applying standard convolutions. The longest path is defined as the path with the maximum $\mu + \beta\sigma$, where β is a constant specified by the user representing the confidence level. This model assumes that all distributions are Gaussian.

In [Wall86], Wallace and Sequin propose an abstract timing verifier which provides a standard user interface, but supports several different timing models. Among the delay models supported are a bounding model similar to SCALD and a statistical model patterned after TA. The framework presented by Wallace and Sequin is elegant and the delay models are more precisely defined.

In both bounding methods, the operators for computing bounds are often only intuitively defined. One of our objectives is to identify the underlying algebraic structure needed to compute bounds and to formally define these operators. In addition to computing delay, we also need to return the longest paths themselves. We discuss each of these issues in greater detail in this thesis.

Recent developments in statistical process simulation have focused attention on the need for CAD tools that model uncertainty. In describing their process simulator, Fabrics II [Nass84], Nassif and his colleagues state the following:

> For performance evaluation prior to IC manufacturing, designers usually employ a circuit simulator, such as SPICE. However, the accuracy of such a performance evaluation strongly depends on the accuracy of the device model parameters used in the simulation. In order to guarantee accuracy of circuit simulation, model parameters are extracted from measurements made on fabricated devices ... An alternative method for determining device parameters is to use a sequence of process and device simulators, such as Fabrics II, which contain physical models of fabrication steps and semiconductor devices. [Nass84, p. 41]

Fabrics II is a process simulator that produces statistical values for integrated circuit (IC) parameters. This system consists of two parts: a process simulator and a device

simulator. Given a description of the manufacturing steps including the process parameters (e.g., times and temperatures of the diffusions steps, doses and energies of the ion implantations) and disturbances (i.e., random variables with probability distributions which model disturbances such as mask misalignment and uneven absorption of dopant), the simulator computes a set of physical parameters for the process (e.g., impurity concentrations, sheet resistances, gradient voltages, and oxide thicknesses). From the physical parameters generated by the process simulator and the layout of a device, the device simulator produces statistical estimates of device parameters. These parameters include threshold voltage, intrinsic transconductance, the dimensions of the device, and intranodal and substrate zero-bias capacitances. When coupled with a circuit simulator, this statistical information may be used to sample the performance of an integrated circuit.

Others employ these device parameter estimates to perform circuit simulation. The first method is a novel bounding approach presented by Zukowski. In [Zuko86], he states:

> A circuit designer is often concerned with the performance of a circuit over a fairly wide range of inputs and circuit models. The uncertainties in the simulation, often arising from variations in fabrication processes, produce a range of circuit behaviors that can be captured in a bound. Since complete bounding simulators have not yet been available to circuit designers, they often use a technique called approximate worst case analysis to estimate the range of behaviors. In approximate worst case analysis, an exact simulator is used at least twice, with inputs that are at the extremes thought most likely to produce extreme behavior. One use of a bounding simulator is to replace this approximate procedure with a rigorous one. [Zuko86, p. 53]

To accomplish this goal, his algorithms use simplified circuit models whose behavior bounds that of the more accurate models. His basic strategy consists of three parts: devising efficient methods to compute bounds on the behavior of very simple circuits, transforming the more realistic models into these simple ones by exploiting the monotonic properties of the model, and using relaxation techniques to analyze the subcircuits. The basic elements of his simplified model are a lumped resistor, a purely resistive transistor, and a lumped capacitor. These are combined to create a *bounding*

circuit whose behavior bounds that of the original. The key characteristic of a bounding circuit is that its behavior is a monotonic function of input voltage and component behavior. To compute bounds on the behavior of the circuit, Zukowski's algorithm uses relaxation techniques to analyze the network equations that define behavior. It performs the relaxation twice – once to compute upper bounds and again to compute lower bounds. The monotonicity of the variables determines the values used in each computation.

Instead of creating simplified circuit models, our approach is to define algebras for manipulating bounds. The two methods are complementary and it may be possible to incorporate both within a single system. For example, with a bounding algebra, the relaxation phase would only be performed once to compute both upper and lower bounds.

The second electrical simulation method is more representative of current practice. In [Dive84], Divekar addresses the problem of statistical circuit simulation to determine circuit performance. He advocates using Monte Carlo simulation to model the effects of device parameter uncertainty. The Monte Carlo method [Sobo75] is a commonly used, statistically-based technique for handling uncertainty. Using this procedure, the circuit is simulated many times. Each time, the algorithm randomly selects a value for each variable from its distribution. The results of each trial are recorded and compiled to compute bounds or a statistical distribution of the results. While this method is effective, it is computationally-expensive. For compute-intensive applications, such as simulation, the time complexity of the basic algorithm prohibits large numbers of Monte Carlo trials; however, many trials are necessary to accurately determine circuit behavior. The goal of this thesis is to develop methods for modeling uncertainty which produce similar bounds in less time.

1.4 Bounding vs. Statistical Methods

These current approaches to modeling uncertainty can be divided into two general categories: bounding methods and statistical methods. In the bounding methods, parameter values are chosen at the minimum and maximum endpoints of their ranges to compute worst and best-case circuit behavior. In addition to worst-case bounds, statistical techniques return a probability distribution of the results, represented as a mean value and standard deviation. There are advantages and disadvantages to each

of these methods.

Proponents of statistical methods argue that their approach is more accurate. Bounding models return performance limits without any indication of the probability of their occurrence. If the probability of these worst-case conditions is small, then these results are overly conservative.

While this is certainly true, bounding techniques offer several advantages over statistical ones. First, they are simpler and more intuitive. Operations for manipulating bounds are easier to derive than those for statistical distributions. An arithmetic for combining ranges, called *interval arithmetic* [Kuli81, Moor79], has been developed. In this thesis, we employ the principles of this arithmetic to derive other operations needed in VLSI applications. For statistical distributions, no such algebras exist. Hitchcock does provide a rule for adding two Gaussian distributions; however, more complex operations are needed for other VLSI applications and we cannot assume that the distributions will always be normal. Currently, the only general methods available for statistical analysis are trial methods, such as Monte Carlo simulation.

The second advantage is speed. Bounding algorithms are inherently much faster than statistical methods. As demonstrated in this thesis, bounding methods are several orders of magnitude faster than Monte Carlo simulation.

Third, bounding methods are more general. Value ranges can be used to model types of uncertainty that are not suited to statistical measures. For instance, a mean value and standard deviation have little meaning for multiple-valued logics. Also, uncertainty that arises as a result of exploring design alternatives cannot be modeled as a statistical distribution.

1.5 Our Solution

For the above reasons, we have chosen a bounding approach based on interval analysis to model uncertainty. The principles of our method are derived from a field of mathematics known as *interval arithmetic* [Kuli81, Moor79], introduced in the 1960's to bound truncation error in digital computers. Building on the basic algebraic structures of this arithmetic, we create a formal framework for modeling uncertainty in VLSI design. By deriving application-specific interval algebras for manipulating uncertain values and incorporating them within existing algorithms for analyzing VLSI circuits, we create new interval versions of the algorithms that compute bounds on the results

when given interval inputs.

The first step of this methodology is to represent uncertain values as intervals. We begin with a few definitions. An interval $[a, b]$ over an ordered set of values S is defined as the set $\{x \mid x \in S, a \leq x \leq b\}$. A *degenerate* interval $[a, a]$ is an interval which contains only the one element a. Let I be the set of all intervals over the real numbers $\Re$; thus, $I \equiv \{[a, b] \mid a, b \in \Re\}$. In this thesis, we use two notations to denote intervals: we either represent an interval by its endpoints in square brackets (i.e., $[a, b]$) or as a variable with a hat (i.e., $\hat{x}$).

The second step is to define algebras for manipulating the intervals. Since each application needs different combinations of operators, these algebras are custom-tailored for each application. For example, we need interval arithmetic (i.e., addition, subtraction, etc.) to calculate the delay of a transistor network, but only minimum, maximum, and blocking operations to perform switch-level simulation. While the algebras are application-specific, the method for deriving their interval operators is the same for all applications. Each interval operator F is the logical extension of a real-valued operator f and is derived from it using the following formula [Moor79]:

$$F(\hat{x}, \hat{y}) \equiv \{f(x, y) \mid x \in \hat{x} \text{ and } y \in \hat{y}\}. \tag{1.1}$$

In other words, an interval operator is derived from its real counterpart by applying the non-interval operator to all possible combinations of the elements chosen from each interval and taking the union of the results. Using this technique, a rule is obtained for computing bounds on the result. For example, interval addition $\oplus$ is defined as follows: $[a, b] \oplus [c, d] = \{x + y \mid x \in [a, b] \text{ and } y \in [c, d]\} = [a + c, b + d]$.

Using this formula, the following interval arithmetic operations $\oplus$ (addition), $\ominus$ (subtraction), $\odot$ (multiplication), and $\oslash$ (division) over the set I are defined:

$$[a, b] \oplus [c, d] \equiv [a + c, b + d] \tag{1.2}$$

$$[a, b] \ominus [c, d] \equiv [a - d, b - c] \tag{1.3}$$

$$[a, b] \odot [c, d] \equiv [min(ac, ad, bc, bd), max(ac, ad, bc, bd)] \tag{1.4}$$

$$[a, b] \oslash [c, d] \equiv [a, b] \odot [1/c, 1/d] \quad \textit{provided that } 0 \notin [c, d] \tag{1.5}$$

Similarly, we derive the following minimum (MIN) and maximum (MAX) operators:

$$MIN([a, b], [c, d]) \equiv [min(a, c), min(b, d)] \tag{1.6}$$

$$MAX([a, b], [c, d]) \equiv [max(a, c), max(b, d)] \tag{1.7}$$

In addition, there are several order relations which are useful:

$$[a,b] = [c,d] \iff a = c \text{ and } b = d \tag{1.8}$$

$$[a,b] < [c,d] \iff b < c \tag{1.9}$$

$$[a,b] > [c,d] \iff a > d \tag{1.10}$$

$$[a,b] \leq [c,d] \iff a \leq c \text{ and } b \leq d \tag{1.11}$$

$$[a,b] \geq [c,d] \iff a \geq c \text{ and } b \geq d \tag{1.12}$$

$$[a,b] \parallel [c,d] \iff [a,b] \subset [c,d] \text{ or } [c,d] \subset [a,b] \tag{1.13}$$

The last of these relations states that two intervals are incomparable ($\parallel$), if one is a proper subset of the other.

Over the set of real intervals I, these operations have a number of important properties. Interval addition and multiplication are both commutative and associative. The degenerate intervals $[0,0]$ and $[1,1]$ form the additive and multiplicative identities respectively. The distributive property

$$\hat{x} \odot (\hat{y} \oplus \hat{z}) = (\hat{x} \odot \hat{y}) \oplus (\hat{x} \odot \hat{z}) \text{ for all } \hat{x}, \hat{y}, \hat{z} \in I$$

holds only in the following special cases:

1. if $\hat{x}$ is a degenerate interval or

2. if $\hat{y} \odot \hat{z} > [0,0]$.

However, the subdistributivity property always holds; therefore,

$$\hat{x} \odot (\hat{y} \oplus \hat{z}) \subseteq (\hat{x} \odot \hat{y}) \oplus (\hat{x} \odot \hat{z}).$$

Non-degenerate intervals have no additive or multiplicative inverses; in particular, $\hat{x} \oslash \hat{x} \neq [1,1]$ unless $\hat{x}$ is degenerate. Another important property of interval arithmetic is *monotonic inclusion*: if $\hat{w}, \hat{x}, \hat{y}, \hat{z} \in I$ and $* \in \{\oplus, \ominus, \odot, \oslash\}$

$$\hat{w} \subseteq \hat{y} \text{ and } \hat{x} \subseteq \hat{z} \Longrightarrow \hat{w} * \hat{x} \subseteq \hat{y} * \hat{z}.$$

For the minimum and maximum operators, we must augment the set of real numbers $\Re$ to include ∞ and $-\infty$. Let I^+ denote the set of intervals over this augmented set. The interval minimum and maximum operators are both commutative and associative. $[\infty, \infty]$ and $[-\infty, -\infty]$ are the identity elements of MIN and MAX respectively. I^+

is closed with respect to either operator. Proofs of these properties are provided in Appendix A.

The final step of the method is to embed these interval operators within existing algorithms for analyzing VLSI circuits. This creates an interval version of the algorithm that computes bounds on the result of the computation when given interval input parameters.

1.6 Applications

In the remainder of this thesis, we present several examples chosen from various areas of VLSI analysis to illustrate the usefulness and elegance of the interval approach to uncertainty. These examples run the gamut from simulation to placement and are intended to demonstrate that interval analysis can be successfully incorporated into many different types of VLSI algorithms. Chapter 2 contains the first application, switch-level simulation. Here, we employ interval methods to simulate transistor networks with unknown transistor strengths and node sizes. The second application is timing analysis. This problem has two parts. In Chapters 3 and 4, we perform RC timing analysis to compute bounds on the delay of a transistor network given uncertain device characteristics. In Chapter 5, we combine these delays using critical path analysis to determine bounds on the delay of the circuit. The third application is placement. In Chapter 6, we compute an interval objective function and discuss how uncertain cost functions may be used when seeking minimum-cost placements. These three applications not only illustrate our interval methodology, but also reveal its strengths and weaknesses. In Chapter 7, we conclude by discussing the merits and limitations of the interval paradigm and suggesting other areas for which these same interval techniques may be applied.

Chapter 2

Switch-Level Simulation with Uncertain Signal Strength

The first application of our interval technique is switch-level simulation. Numerous techniques for performing rapid switch-level simulation have been developed in recent years [Brya80, Brya84, Byrd85, Gecs87, Rama83a, Sund87]. Without exception, these methods require detailed knowledge of circuit implementation. In particular, the relative conductances, or strengths, of all transistors must be specified in advance. For example, Figure 2.1 depicts a simple transistor network consisting of two transistors in series. Because the relative strengths of the two transistors are not specified, this circuit cannot be simulated, even though its output is undefined only when both transistors are conducting.

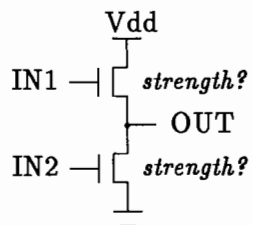


Figure 2.1: A Simple Transistor Network (strengths not specified)

While strength information can be derived from low-level circuit descriptions (e.g. device geometry), it is rarely present in the higher-level specifications used in today's advanced design tools. Uncertainty is inherent in top-down design, where all of the implementation details may not be available when testing the circuit at an intermediate design stage. Simulating a transistor network to verify its behavior before calculating the exact transistor size ratios is one such example.

In this chapter, we describe an extension to an existing switch-level simulator, MOSSIM II [Brya84], capable of handling uncertain transistor strengths. MOSSIM II assumes a certain underlying mathematical structure. We represent transistor strengths by ranges of values and create an interval extension of this algebraic structure to accommodate strength ranges. Our method is simple yet accurate, as we prove later in this chapter. The resulting interval switch-level simulation algorithm, Intssim, has a time complexity comparable to that of the original. In addition, our algorithm also models uncertain node sizes.

2.1 The Transistor Model

2.1.1 Traditional Approach

In switch-level simulation, a transistor is modeled as an ideal switch, in which the gate of the transistor controls the flow of current between the other two terminals (See Figure 2.2). The state of a transistor (i.e., whether the switch is *open* (nonconducting), *closed* (conducting), or *indeterminate*) is uniquely determined by the transistor type and the signal value at its gate. Table 2.1 displays the state information for the three transistor types most commonly used in VLSI technologies.

A network of transistors is modeled as a weighted, undirected graph [Brya84]. The vertices of this graph represent the source or drain terminals of the transistors in the network and the edges indicate the connectivity between these terminals. The vertices of the graph are divided into two mutually exclusive classes: inputs (which are direct connections to the voltage sources Vdd and Gnd) and storage nodes (everything else). Associated with each vertex are two attributes: signal value and node size. The first attribute is the current signal value of the node (0, 1, or X). The size of a node is a measure of its capacitance. Rather than using the actual capacitances themselves, switch-level simulators usually employ a set of discrete nodes sizes. If the vertex is an

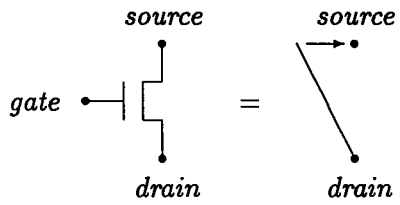


Figure 2.2: The Switch-Level Transistor Model

Gate value	Transistor type		
	n-type	p-type	d-type
0	open	closed	closed
1	closed	open	closed
X	indeterminate	indeterminate	closed

Table 2.1: Transistor State Table

input node, it is assigned size ω, indicating infinite capacitance. All other nodes are ranked by capacitance and assigned values from the set $\{\kappa_1,...,\kappa_{max}\}$.

Each edge in the graph corresponds to a transistor in the network and indicates its state (See Figure 2.3). A graph edge is drawn between two vertices in the graph, if the transistor linking that pair of nodes is *potentially* closed. There are two types of edges: *1-edges* for transistors in a *closed* state and *X-edges* for transistors in an *indeterminate* state. Associated with each edge is a strength attribute, which is a measure of the conductance of the transistor. Conductance is determined by the geometry of a transistor and, in particular, by the ratio of its length to its width. Instead of using the actual ratios in the simulation, switch-level algorithms employ a finite set of discrete strengths, like those used to specify node sizes. To determine these strengths, the transistors are ranked according to ratio size and transistor type and are assigned values, indicating relative conductance, from a set of discrete possibilities, $\{\gamma_1,...,\gamma_{max}\}$.

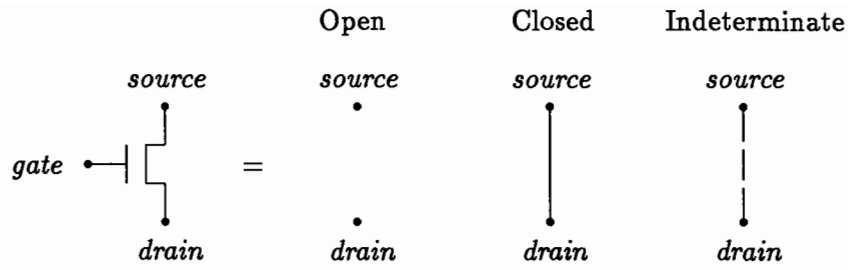


Figure 2.3: Transistor State and Graph Edges

2.1.2 Unknown Transistor Strength

The set of transistor strengths $T = \{\gamma_1, ..., \gamma_{max}\}$ is ordered such that $\gamma_1 < \gamma_2 < ... < \gamma_{max}$. A transistor with unknown or indeterminate strength may assume any number of consecutive strengths chosen from this set; therefore, we represent a transistor's strength by a range of these values or an interval. If the relative strength of a transistor is precisely known, then we represent it as a degenerate interval. Our extensions will manipulate degenerate and non-degenerate intervals with equal ease; thus, our extension subsumes current switch-level simulation algorithms.

2.1.3 An Example

Figure 2.4 depicts a transistor network containing unknown transistor strengths. All transistors in the network are given unique names and assigned interval strengths. From this network, the connectivity graph in Figure 2.5 is generated. Each graph vertex is labeled with a unique identifier followed by its signal value and size attributes in parentheses. For instance, vertex 0 represents the source terminal of transistor t_0. Since it is connected to the voltage source Vdd, this node has a signal value of 1 and an infinite capacitance ω. Likewise, vertex 4 represents the drain terminals of transistors t_2, t_3, and t_5, which are all connected to ground. This vertex thus has a signal value of 0 and an infinite capacitance ω. The rest of the vertices represent transistor terminals that are not directly connected to either Vdd or Gnd; therefore, these vertices are assigned an uncertain signal value X and a relatively weak capacitance κ_1. Each edge in the graph represents a conducting transistor and is labeled with its

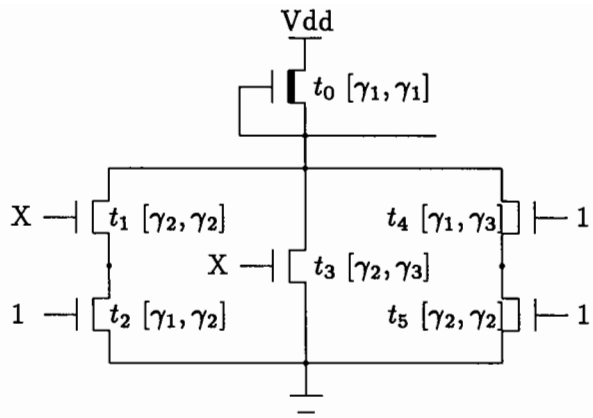


Figure 2.4: Transistor Network with Unknown Transistor Strengths

name and associated strength. 1-edges appear as solid lines and X-edges as dashed lines.

2.2 The Simulation Algorithm

To analyze a network containing transistors with unknown strength, we must effectively determine the steady-state value of the network under all possible strength assignments to each transistor of unknown strength. If the steady-state value of a node is the same regardless of the strengths assigned to these transistors, then that signal value is taken to be the steady-state value of the node. If, however, different strength assignments yield different signal values for the same node, then the steady-state value of the node is taken to be X.

Table 2.2 presents this analysis for the network in Figure 2.5. All combinations of possible strengths are assigned to the unknown-strength transistors in the network and the resulting steady-state is computed using the original MOSSIM II algorithm. Since the network must be analyzed for each possible strength combination, the basic simulation algorithm is executed $\prod_{i=1}^{k} g_i$ times, where k is the number of transistors with unknown strength and g_i is the number of different strengths that can be assigned to transistor i. This method has a worst-case time complexity of $O(g^k)$, where k is

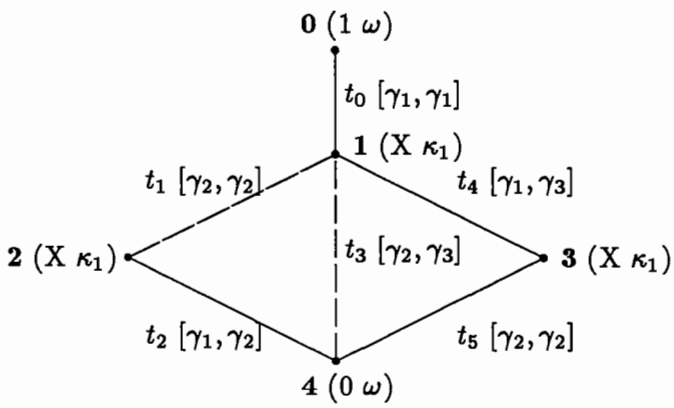


Figure 2.5: Connectivity Graph of Network with Unknown Transistor Strength

number of transistors with unknown strength and g is the maximum number of different transistor strengths. Clearly, a more efficient technique is needed. To remedy this problem, we create an interval version of MOSSIM II which more efficiently computes the steady-state value of the network.

2.2.1 The MOSSIM II Simulation Algorithm

We now review the MOSSIM II switch-level simulation algorithm [Brya84]. We begin with a few definitions. A *current path* through the transistor network is a path originating a signal source that passes through zero or more consecutive conducting transistors. The *signal value* of a path is the signal value of its source node. (e.g., Paths originating at Vdd have a signal value of 1. Paths originating at Gnd have a signal value of 0). The *strength* of a path is defined as the minimum of source node size (capacitance) and the strength (conductance) of the weakest transistor in the path. Bryant identifies three types of paths distinguished by their strengths: input paths, driven paths, and stored-charge paths. An input path consists of a signal node connected to Vdd or Gnd. Since they are directly connected to voltage sources, input paths are the strongest type of paths and have strength ω. Driven paths originate at Vdd or Gnd and pass through one or more transistors. The conductance of the transistors in the path determines the resulting strength of the signal; thus, driven paths have strengths in the range γ_1 to

Strength			Steady-state				
t_2	t_3	t_4	0	1	2	3	4
γ_1	γ_2	γ_1	1	X	X	0	0
γ_1	γ_2	γ_2	1	0	0	0	0
γ_1	γ_2	γ_3	1	0	0	0	0
γ_1	γ_3	γ_1	1	X	X	0	0
γ_1	γ_3	γ_2	1	0	0	0	0
γ_1	γ_3	γ_3	1	0	0	0	0
γ_2	γ_2	γ_1	1	X	0	0	0
γ_2	γ_2	γ_2	1	0	0	0	0
γ_2	γ_2	γ_3	1	0	0	0	0
γ_2	γ_3	γ_1	1	X	0	0	0
γ_2	γ_3	γ_2	1	0	0	0	0
γ_2	γ_3	γ_3	1	0	0	0	0
Final values			1	X	X	0	0

Table 2.2: Combinational Analysis of Network containing Unknown Strengths

γ_{max}. Stored-charge paths originate at storage nodes and pass through zero or more transistors. These paths provide the weakest signals, because the source of the signal is a finite reservoir of charge isolated from a voltage source. Such paths have strengths ranging from κ_1 to κ_{max}, depending on the size of the source node. With the addition of an identity element λ, Bryant presents the following natural ordering of path strengths:

$$\omega > \gamma_{max} > ... > \gamma_1 > \kappa_{max} > ... > \kappa_1 > \lambda. \tag{2.1}$$

To compute the resultant steady-state value of a given node in the network, Bryant computes all paths to the node and chooses the one with the maximum strength. The signal value of this path determines the steady-state value of the node. If a second path with this same maximum strength but a different signal value exists, then the resulting steady-state value of the node is X (indeterminate).

To perform these calculations, Bryant employs three operations: $+$ (maximum), $\cdot$ (minimum), and $\sim$ (blocking). The first two operators, $+$ and $\cdot$, are the standard maximum and minimum functions. Given two path strengths, the $+$ operator compares them and returns the larger of the two. Similarly, the $\cdot$ operator returns the smaller path strength. Bryant chose this notation for his minimum and maximum operators, because he uses these operators in his matrix computations (described later in this

section) as if he was performing matrix multiplication and addition. The third operator, $\sim$, compares two path strengths and discards the first if it is less than the second. Since the second path has greater strength, it prevents the first path from influencing the steady-state of the node. This blocking function is defined as follows:

$$a \sim b \equiv \begin{cases} a & \text{if } a \geq b \\ \lambda & \text{otherwise} \end{cases} \tag{2.2}$$

The MOSSIM II algorithm has four distinct stages. In the first stage, Bryant creates the matrices which describe the transistor network. The first two matrices, s and y, contain the size and initial signal value of each vertex in the connectivity graph representing the transistor network; thus, s_i contains the size or capacitance of node i and y_i its initial value. The connectivity between these nodes is given by two matrices, $G1$ and GX; one to hold the *1-edges* and the other to hold the *X-edges*. If there is a closed transistor joining nodes i and j in the transistor network, then there is a 1-edge between these nodes in the corresponding connectivity graph. An entry for this edge is made in matrix $G1$ by assigning the transistor strength of this closed transistor to $G1(i,j)$. Since connectivity is a commutative property, $G1(i,j) = G1(j,i)$. After all of the 1-edges are recorded, the remaining locations in $G1$ are filled with λ. The matrix GX is defined similarly, recording *X-edges* instead of *1-edges*.

When computing path strength, signal paths containing X-edges pose problems. Since a transistor with indeterminate state may be either closed or open, the corresponding edge may or may not actually exist in the graph. If considered, strong signal paths containing X-edges may prevent weaker signal paths which do not contain X-edges from influencing the signal value at a node. To prevent this undesirable condition, X-edges are ignored in the initial path analysis. Signal paths which do not contain any X-edges are called *definite* paths. *Blocking path strength* is defined as the strength of the strongest definite path to a vertex.

The second step of the simulation algorithm is to compute the blocking path strength for each of the nodes in the network. In MOSSIM II, blocking path strength is derived using the following recurrence:

$$\begin{aligned} q(i,0) &= s_i \\ q(i,r) &= s_i + \sum_{j=1}^{n} G1(i,j) \cdot q(j,r-1) \end{aligned} \tag{2.3}$$

The entry $q(i,r)$ gives the strength of the strongest definite path with length less than or equal to r to reach vertex i. The base case reflects the strength of the strongest path to node i with length 0. This path consists of the node itself and has a strength equal to the node size, s_i. The recurrence relation computes the maximum of the node size and the strengths of all paths through the network to node i originating at other nodes and passing through at most r closed transistors. The strength of the path to node i through the closed transistor linking nodes i and j is the minimum of the transistor strength $G1(i,j)$ and the strength of the strongest definite path $q(j,r-1)$ through the network to node j. Note that r is bounded by the number of vertices n in the graph; thus, $q_i = q(i,n)$. This new matrix q is the blocking strength matrix.

The third step is to compute the strongest unblocked pull-up and pull-down paths to each vertex. A pull-up path is a path that originates at a node with a signal value of 1 (or X). Likewise, a pull-down path originates at a node with a signal value of 0 (or X). To calculate unblocked pull-up paths, the algorithm begins at nodes with signal values of 1 (or X) and traces the paths from these nodes to all other vertices in the connectivity graph, computing the strongest path to each vertex. In contrast to the previous step, both definite and indefinite paths are included in this computation. Once the strongest pull-up path to a vertex is computed, it is compared with the blocking path strength for that vertex using the $\sim$ operator given in equation 2.2. If the strength of this pull-up path is less than the blocking path strength, then a stronger definite path exists; therefore, this pull-up path is discarded. Only paths with strength greater than or equal to the blocking strength are saved. By starting at nodes with signal values of 0 (or X), the strongest unblocked pull-down paths are similarly computed. The unblocked pull-up and pull-down paths to a vertex are then used to determine its steady-state.

Bryant introduces matrices u and d representing the strength of the strongest unblocked pull-up and pull-down paths to each vertex. To assist in generating these matrices, two new functions **up** and **down** are also introduced. These functions identify the sources of possible pull-up and pull-down paths and are defined as follows:

$$\mathbf{up}(s_i, y_i) \equiv \begin{cases} s_i & \text{if } y_i = 1 \text{ or } X \\ \lambda & \text{if } y_i = 0 \end{cases} \tag{2.4}$$

$$\mathbf{down}(s_i, y_i) \equiv \begin{cases} s_i & \text{if } y_i = 0 \text{ or } X \\ \lambda & \text{if } y_i = 1 \end{cases} \tag{2.5}$$

Given these two functions, the matrices u and d are generated by the following recurrences:

$$\begin{aligned} u(i,0) &= \mathbf{up}(s_i, y_i) \sim q_i \\ u(i,r) &= \left[\mathbf{up}(s_i, y_i) + \sum_{j=1}^{n} (G1(i,j) + GX(i,j)) \cdot u(j, r-1)\right] \sim q_i \qquad (2.6) \\ d(i,0) &= \mathbf{down}(s_i, y_i) \sim q_i \\ d(i,r) &= \left[\mathbf{down}(s_i, y_i) + \sum_{j=1}^{n} (G1(i,j) + GX(i,j)) \cdot d(j, r-1)\right] \sim q_i \qquad (2.7) \end{aligned}$$

The entry $u(i,r)$ gives the strength of the strongest unblocked pull-up path with length less than or equal to r and destination node i. The base case $u(i,0)$ computes the strength of the pull-up path consisting of the single node i, saving it only if its strength exceeds the blocking strength q_i. The recurrence relation computes the maximum strength of all pull-up paths to node i that pass through at most r transistors. The $\mathbf{up}(s_i, y_i)$ term represents the strength of the pull-up path containing only node i. The rest of the terms represent the strengths of the pull-up paths to node i originating at other nodes in the network. The strength of a pull-up path passing through the transistor linking nodes i and j is computed by taking the minimum of the transistor strength $G1(i,j) + GX(i,j)$[1] and the strength of the strongest unblocked pull-up path $u(j, r-1)$ to node j. Once the strength of the strongest pull-up path to node i is found, it is compared to the blocking strength q_i. If the strength of this pull-up path equals or exceeds the blocking strength, it is saved. As with q, the recursion terminates in at most n steps; thus, u_i is assigned the value $u(i,n)$. Matrix d is computed analogously.

In the final step, the results of the previous computation are used to determine the steady-state response of the network. If there is an unblocked pull-up path to a vertex and no unblocked pull-down paths, then the new signal value at the vertex is 1. Similarly, if there is an unblocked pull-down path to the vertex and no unblocked pull-up paths, then the new signal value is 0. If both unblocked pull-up and pull-down paths exist, then the signal value is taken to be X. This function is defined as follows:

$$Y_i \equiv \begin{cases} 1 & \text{if } d_i = \lambda \\ 0 & \text{if } u_i = \lambda \\ X & \text{otherwise} \end{cases} \qquad (2.8)$$

[1] In this computation, 1-edges and X-edges are considered.

2.2.2 Extensions to Handle Unknown Transistor Strength

To create an algorithm for simulating transistor networks containing uncertain transistor strengths, we modify the path analysis algorithm given above to accommodate intervals. In particular, we define three interval operators which are the logical extensions of the discrete operators $+$, $\cdot$, and $\sim$. Each interval operator is obtained from its non-interval counterpart using the standard derivation: $F(\hat{x}, \hat{y}) \equiv \{f(x, y) \mid x \in \hat{x}\ and\ y \in \hat{y}\}$. In other words, we derive the interval definition of each operator by applying the associated discrete function to all combinations of elements from each interval and taking the union of the results. The resulting maximum operator for intervals $\boxplus$ and the minimum operator $\boxdot$ are defined as follows:

$$[a, b] \boxplus [c, d] \equiv [a + c, b + d] \tag{2.9}$$

$$[a, b] \boxdot [c, d] \equiv [a \cdot c, b \cdot d]. \tag{2.10}$$

In addition to these two operators, we introduce a blocking operator for intervals $\simeq$ defined as follows:

$$[a, b] \simeq [c, d] \equiv \begin{cases} [a + c, b] & \text{if } b \geq c \\ [\lambda, \lambda] & \text{otherwise} \end{cases} \tag{2.11}$$

This operator compares two interval strengths and returns only those portions of the first interval that equal or exceed the lower bound of the second. In essence, the lower bound of the second interval acts as the blocking strength. Any strength less than this blocking strength is discarded, because we already have a definite path with greater strength.

Having defined these interval operations, we substitute them into the path analysis equations given in the previous section to generate a switch-level simulation algorithm which handles unknown transistor strength.

2.2.3 Extensions to Handle Uncertain Node Size

It should be obvious that our interval version of the switch-level simulation algorithm can also be applied to uncertain node sizes, since node size and transistor strength have the same underlying algebraic structure. To handle uncertain node sizes, we represent them as intervals over the ordered set $\{\kappa_1, \kappa_2, ..., \kappa_{max}, \omega\}$. We then employ the interval operations given above to manipulate these size ranges. No further changes to the algorithm are required.

	s_i	y_i
0	$[\omega, \omega]$	1
1	$[\kappa_1, \kappa_1]$	X
2	$[\kappa_1, \kappa_1]$	X
3	$[\kappa_1, \kappa_1]$	X
4	$[\omega, \omega]$	0

Table 2.3: Describing the Network – the Vertices

G1					
	0	1	2	3	4
0	$[\lambda, \lambda]$	$[\gamma_1, \gamma_1]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
1	$[\gamma_1, \gamma_1]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_1, \gamma_3]$	$[\lambda, \lambda]$
2	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_1, \gamma_2]$
3	$[\lambda, \lambda]$	$[\gamma_1, \gamma_3]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_2, \gamma_2]$
4	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_1, \gamma_2]$	$[\gamma_2, \gamma_2]$	$[\lambda, \lambda]$

GX					
	0	1	2	3	4
0	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
1	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\gamma_2, \gamma_2]$	$[\lambda, \lambda]$	$[\gamma_2, \gamma_3]$
2	$[\lambda, \lambda]$	$[\gamma_2, \gamma_2]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
3	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$
4	$[\lambda, \lambda]$	$[\gamma_2, \gamma_3]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$	$[\lambda, \lambda]$

Table 2.4: Describing the Network – the Edges

2.2.4 Examples

Tables 2.3-2.5 present the analysis of the transistor network given in Figure 2.5. The first two tables describe the initial state of the transistor network. Table 2.5 displays the resultant blocking path strength, unblocked pull-up and pull-down paths, and the final steady-state for each node in the network. Note that the steady state values computed are identical to those derived in Table 2.2.

To further illustrate how the interval operations are employed to determine these results, we examine the computations for one of the nodes. Table 2.6 enumerates all paths through the network which terminate at node 3. The table gives both the signal value and strength of each path. To calculate the blocking path strength of node 3,

	q_i	u_i	d_i	Y_i
0	$[\omega,\omega]$	$[\omega,\omega]$		1
1	$[\gamma_1,\gamma_2]$	$[\gamma_1,\gamma_1]$	$[\gamma_2,\gamma_3]$	X
2	$[\gamma_1,\gamma_2]$	$[\gamma_1,\gamma_1]$	$[\gamma_2,\gamma_2]$	X
3	$[\gamma_2,\gamma_2]$		$[\gamma_2,\gamma_3]$	0
4	$[\omega,\omega]$		$[\omega,\omega]$	0

Table 2.5: Interval Algorithm Results

we find the strength range of the strongest definite path. In this computation, we consider four paths: the path from node 0 through the pull-up transistor to node 3 (with strength $[\gamma_1,\gamma_1]$), the path originating at node 1 (with strength $[\kappa_1,\kappa_1]$), the path consisting of node 3 itself (with strength $[\kappa_1,\kappa_1]$), and the pull-down path originating at node 4 (with strength $[\gamma_2,\gamma_2]$). The blocking path strength is found by computing the maximum of these strengths; thus, $q_3 = [\gamma_1,\gamma_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\gamma_2,\gamma_2] = [\gamma_2,\gamma_2]$. In the next stage of the algorithm, we calculate unblocked pull-up and pull-down path strengths. We accomplish this by finding the strength ranges of the strongest pull-up and pull-down paths and then comparing them to the blocking path strength to eliminate blocked paths. The strongest pull-down path strength is the maximum strength of all paths originating at nodes with value 0 or X; therefore, the maximum pull-down path strength is $[\kappa_1,\kappa_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\kappa_1,\kappa_1] \boxplus [\gamma_2,\gamma_2] \boxplus [\gamma_1,\gamma_3] \boxplus [\gamma_1,\gamma_2] = [\gamma_2,\gamma_3]$. Likewise, the strongest pull-up path strength is $[\gamma_1,\gamma_1]$. When compared to the blocking path strength for node 3, we discover that there are no unblocked pull-up paths (since $[\gamma_1,\gamma_1] \simeq [\gamma_2,\gamma_2] = [\lambda,\lambda]$), but that unblocked pull-down paths do exist (since $[\gamma_2,\gamma_3] \simeq [\gamma_2,\gamma_2] = [\gamma_2,\gamma_3]$). Therefore, we conclude that the steady-state value of node 3 is 0.

The CMOS selector shown in Figure 2.6 is another example. Since no explicit strengths are given, we assign strength $[\gamma_1,\gamma_{max}]$ to all transistors and simulate the network. The simulation results given in Table 2.7 show that if all the transistors in the circuit have equal strength, then the network does not exhibit the desired switching behavior when the selected input value changes from 1 to 0. Instead of changing from 0 to 1, the output becomes undefined. Because the pull-up path is too strong, it competes with the input signal, thus producing an undefined steady-state. The conflict is resolved by assigning a weak transistor strength to the uppermost p-transistor and giving all

Path	X-edges?	Value	Strength
0,1,3	No	1	$[\omega,\omega] \boxdot [\gamma_1,\gamma_1] \boxdot [\gamma_1,\gamma_3] = [\gamma_1,\gamma_1]$
1,3	No	X	$[\kappa_1,\kappa_1] \boxdot [\gamma_1,\gamma_3] = [\kappa_1,\kappa_1]$
2,1,3	Yes	X	$[\kappa_1,\kappa_1] \boxdot [\gamma_2,\gamma_2] \boxdot [\gamma_1,\gamma_3] = [\kappa_1,\kappa_1]$
3	No	X	$[\kappa_1,\kappa_1]$
4,3	No	0	$[\omega,\omega] \boxdot [\gamma_2,\gamma_2] = [\gamma_2,\gamma_2]$
4,1,3	Yes	0	$[\omega,\omega] \boxdot [\gamma_2,\gamma_3] \boxdot [\gamma_1,\gamma_3] = [\gamma_1,\gamma_3]$
4,2,1,3	Yes	0	$[\omega,\omega] \boxdot [\gamma_1,\gamma_2] \boxdot [\gamma_2,\gamma_2] \boxdot [\gamma_1,\gamma_3] = [\gamma_1,\gamma_2]$

Table 2.6: Network paths to Node 3

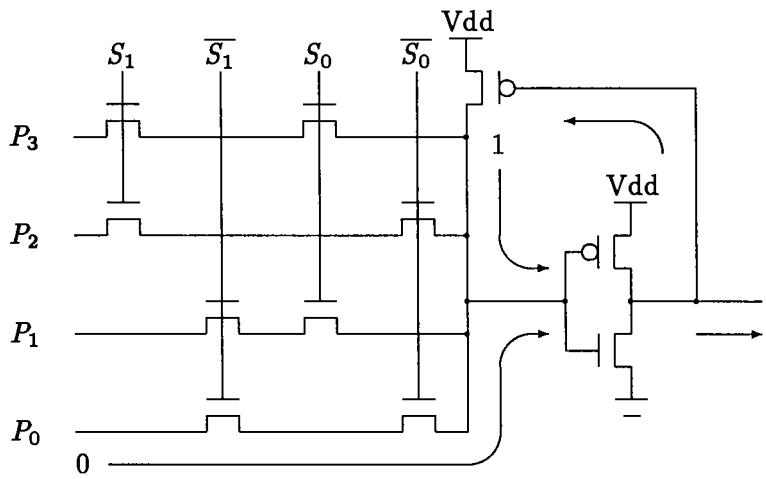


Figure 2.6: CMOS Selector with Feedback

P_i transition	Initial pull-up state	Equal strength Output value	Weak pull-up Output value
$0 \rightarrow 0$	open	1	1
$1 \rightarrow 0$	closed	X	1
$X \rightarrow 0$	indeterminate	X	1
$0 \rightarrow 1$	open	0	0
$1 \rightarrow 1$	closed	0	0
$X \rightarrow 1$	indeterminate	0	0
$0 \rightarrow X$	open	X	X
$1 \rightarrow X$	closed	X	X
$X \rightarrow X$	indeterminate	X	X

Table 2.7: Simulation Results for CMOS Selector Circuit

other transistors in the network greater relative strength. Once this change is made, the circuit exhibits the desired behavior as shown in Table 2.7.

2.3 Proof of the Method

In this section, we formally prove that our interval method correctly computes the resultant steady-state value of a transistor network with interval transistor strengths and node sizes. In his presentation of MOSSIM II, Bryant lists the fundamental properties of his path analysis algebra. He further proves that his algorithm works whenever these properties hold. In the next section, we summarize these properties. We then present our interval algebra and show that it exhibits the same properties. Finally, we conclude that if the original algorithm using Bryant's algebra correctly analyzes a circuit with discrete transistor strengths and node sizes, then the extended algorithm employing our interval algebra will correctly analyze the same circuit with interval transistor strengths and node sizes.

2.3.1 Bryant's Algebra

Let S be an ordered set of discrete transistor strengths and node sizes; that is $S = \{\lambda, \kappa_1, ..., \kappa_{max}, \gamma_1, ..., \gamma_{max}, \omega\}$ such that $\lambda < \kappa_1 < ... < \kappa_{max} < \gamma_1 < ... < \gamma_{max} < \omega$. Bryant defines three operators: maximum $(+)$, minimum $(\cdot)$, and blocking $(\sim)$ over the set S. He then proves that these operators have the properties necessary to correctly determine the steady-state. The fundamental properties of the maximum operation

are: (1) commutativity, (2) associativity, (3) idempotence (4) identity λ, (5) closure, and (6) monotonicity (i.e., $a \geq b \Rightarrow (a+c) \geq (b+c)$ and $(c+a) \geq (c+b)$). Similarly, the minimum operator must have the following properties: (1) associativity, (2) identity ω, (3) annihilator λ, (4) closure, and (5) monotonicity. Furthermore, minimum distributes over maximum. The blocking operator is (1) idempotent, (2) distributes over both minimum and maximum, and (3) is monotonic in its first argument (i.e., $a \geq b \Rightarrow (a \sim c) \geq (b \sim c)$).

2.3.2 The Interval Algebra

Let S' be the set of all intervals over S (i.e., $S' = \{[a,b] \mid a,b \in S \text{ and } a \leq b\}$). We introduce three interval operations over the set S': interval maximum ($\boxplus$), interval minimum ($\boxdot$), and interval blocking ($\simeq$). We now show that these interval operators have the same properties as their discrete counterparts.

Proof. We first examine the properties of the interval maximum operator. Since $\boxplus$ is defined in terms of $+$ which is commutative, associative, and idempotent, it is trivial to show that interval maximum is also (1) commutative, (2) associative, and (3) idempotent. (4) Given that λ is the identity element of $+$, $[\lambda,\lambda]$ is the identity element of $\boxplus$. (5) To prove that S' is closed over $\boxplus$, we must show that $[a,b] \boxplus [c,d] \in S'$ for all $[a,b],[c,d] \in S'$. By definition, $[a,b] \boxplus [c,d] = [a+c, b+d]$. Since $+$ is closed over S, $a+c$ and $b+d$ are in S. All that remains is to show that $a+c \leq b+d$. This follows from the definition of an interval ($[a,b]$ implies $a \leq b$) and the monotonicity of $+$. Hence, $[a,b] \boxplus [c,d] \in S'$. (6) To even be able to say whether $\boxplus$ is monotonic, we require an interval definition for the relational operator $\geq$. We define $\geq$ for intervals as follows:

$$[a,b] \geq [c,d] \iff a \geq c \text{ and } b \geq d.$$

Assuming this definition, $\boxplus$ is monotonic if

$$[a,b] \geq [c,d] \Longrightarrow ([a,b] \boxplus [e,f]) \geq ([c,d] \boxplus [e,f]) \text{ and } ([e,f] \boxplus [a,b]) \geq ([e,f] \boxplus [c,d]).$$

Applying the interval definition of $\geq$, this simplifies to

$$\begin{aligned} a \geq c \text{ and } b \geq d \quad \Longrightarrow \quad & a+e \geq c+e \text{ and } b+f \geq d+f \\ \text{and} \quad & e+a \geq e+c \text{ and } f+b \geq f+d. \end{aligned}$$

This follows from the monotonicity of $+$.

The properties of the interval minimum operator may be proved using similar arguments. Clearly, $\boxdot$ is (1) associative, (2) has identity $[\omega, \omega]$ and (3) annihilator $[\lambda, \lambda]$, (4) is closed, and (5) monotonic. Applying the definitions of $\boxdot$ and $\boxplus$ and noting that $\cdot$ distributes over $+$, we see that $\boxdot$ distributes over $\boxplus$.

Next, we examine the properties of the interval blocking operator $\simeq$. (1) To prove that $\simeq$ is idempotent, we must show that $[a,b] \simeq [a,b] = [a,b]$ for all $[a,b] \in S'$. This follows directly from the definition of $\simeq$ and the idempotence of $+$, since $[a,b] \simeq [a,b] = [a+a,b]$. (2) The next property is that $\simeq$ distributes over $\boxplus$. We must show that $([a,b] \boxplus [c,d]) \simeq [e,f] = ([a,b] \simeq [e,f]) \boxplus ([c,d] \simeq [e,f])$ for all $[a,b],[c,d],[e,f] \in S'$. We begin by applying the definitions of $\simeq$ and $\boxplus$ to expand each expression. Expanding the first expression yields the following:

$$([a,b] \boxplus [c,d]) \simeq [e,f] = \begin{cases} [a+c+e, b+d] & \text{if } b+d \geq e \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

Similarly, expanding the second expression gives:

$$([a,b] \simeq [e,f]) \boxplus ([c,d] \simeq [e,f]) = \begin{cases} [(a+e)+(c+e), b+d] & \text{if } b \geq e \text{ and } d \geq e \\ [a+e, b] & \text{if } b \geq e > d \\ [c+e, d] & \text{if } d \geq e > b \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

To determine whether these definitions are equivalent, we examine each of the four cases. In the first case, $b \geq e$ and $d \geq e$. Clearly, this implies $b+d \geq e$. Since $+$ is commutative, associative, and idempotent, $(a+e)+(c+e) = a+c+e$; thus, the expressions are equivalent. Second, consider the case $b \geq e > d$. Here also $b+d \geq e$; therefore, we must show that $[a+c+e, b+d] = [a+e, b]$. We are given $b > d$, thus $b+d = b$. Since $e > d \geq c$, $a+c+e = a+e$. The third case ($d \geq e > b$) is similar to the second and is proved by repeating the previous arguments with $[a,b]$ and $[c,d]$ exchanged. The final case is trivial, since both expressions yield $[\lambda, \lambda]$ when $b < e$ and $d < e$. Hence, both expressions are equivalent.

Next we prove that $\simeq$ distributes over $\boxdot$. We must show that $([a,b] \boxdot [c,d]) \simeq [e,f] = ([a,b] \simeq [e,f]) \boxdot ([c,d] \simeq [e,f])$ for all $[a,b],[c,d],[e,f] \in S'$. The argument here is similar. We begin by expanding each expression using the definitions of $\boxdot$ and $\simeq$ and then perform case analysis to determine equivalence. The first expression here

gives:

$$([a,b] \boxdot [c,d]) \simeq [e,f] = \begin{cases} [(a \cdot c) + e, b \cdot d] & \text{if } b \cdot d \geq e \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

Given that λ is the annihilator for minimum, the second expression yields the following when expanded:

$$([a,b] \simeq [e,f]) \boxdot ([c,d] \simeq [e,f]) = \begin{cases} [(a+e) \cdot (c+e), b \cdot d] & \text{if } b \geq e \text{ and } d \geq e \\ [\lambda, \lambda] & \text{otherwise} \end{cases}$$

At this point the proof becomes trivial, since these expressions are clearly equivalent.

Finally, we prove that (3) $\simeq$ is monotonic in its first argument (i.e., $[a,b] \geq [c,d] \Longrightarrow ([a,b] \simeq [e,f]) \geq ([c,d] \simeq [e,f])$). From the interval definition of $\geq$ and the assumption that $[a,b] \geq [c,d]$, $a \geq c$ and $b \geq d$; thus, we only have to examine three cases. In the first case, we have $b \geq d \geq e$; therefore, we must show that $a + e \geq c + e$. Clearly this is true, since $+$ is monotonic. The second case is trivial, since if $b \geq e > d$, then $[a+e, b] \geq [\lambda, \lambda]$. The last case occurs when both $b < e$ and $d < e$. In this case, both expressions yield $[\lambda, \lambda]$. We have thus shown that $\simeq$ is monotonic in its first argument.

This proves that our interval extensions to the operators in Bryant's algebra exhibit the same properties. Since Bryant proves that the algorithm works whenever these properties hold, we conclude that our interval algorithm employing these interval operators will correctly compute the steady-state of a transistor network with interval transistor strengths and node sizes. □

2.4 Performance

Our interval path analysis algorithm, Intssim, has been implemented in the C programming language and runs on several machines (VAX-11/780, Encore Multimax, and SUN SPARCstation). We ran both the Intssim and MOSSIM II programs on identical circuits and recorded their response times. Experiments show that the interval version takes about twice as long as MOSSIM II to compute the steady-state value of a transistor network. This is the expected result, since the interval algorithm performs two calculations for each minimum, maximum, or blocking operator encountered. The worst-case time complexity of both path analysis algorithms is $O(s+t)$, where t represents the number of transistors and s represents the number of distinct node sizes and transistor strengths.

2.5 Summary

In this chapter, we considered the problem of simulating transistor networks containing uncertain transistor strengths and node sizes. We represented uncertain transistor strengths by strength ranges and developed an interval algebra to manipulate these ranges. We further generalized an existing switch-level simulator, MOSSIM II, to create a simulation algorithm for networks with interval transistor strengths and node sizes. Our algorithm has a running time of about twice that of the original and a comparable worst-case time complexity. We also presented a formal proof of correctness for our method.

Chapter 3

Modeling Uncertainty in RC Timing Analysis I

Timing verification is an important aspect of VLSI design. Its purpose is to predict the operating speed of a VLSI chip before it is fabricated. This speed is a function of three components: structure, technology, and the manufacturing process. The first two of these are well known to chip designers. Consider, for example, carry-propagate and carry-lookahead adders. Although both perform the same function, they have different structures and hence operating speeds. A comparison of CMOS and nMOS technologies provides an example of the effects of technology on performance. By replacing slow depletion mode pull-ups by complementary logic, CMOS designs achieve faster operating speeds than their nMOS counterparts. The third component, the manufacturing process, is also an important factor in chip performance. During fabrication, even slight variations in any of a multitude of processing parameters (e.g., exposure time, temperature, gas pressure) produce variations in the process and consequently in device parameters. For example, over and under-etching, oxide growth rates, and ion implantation levels all affect the electrical characteristics of the resulting circuit. In particular, they affect the transistor sizes, threshold voltages, resistance factors, and capacitance factors used to perform timing analysis. As a result, the circuit specifications given for each technology are only approximations to the true values. These parameters actually have a range of possible values, typically varying between 1 and 20 percent from those stated. Currently, most timing verifiers use only the average values given to estimate circuit performance, but it is important to know how the circuit will

operate with other possible combinations of these values.

While there are many timing verifiers in use today [Hitc82, Joup83, McWi80, Oust83, Wall86], most do not address the problem of parameter uncertainty. A few [Hitc82, McWi80, Wall86], however, provide partial solutions to this problem. These logic-level timing verifiers use bounds on the delay of each macro-component (e.g., AND gates, OR gates, flip-flops) to compute bounds on the performance of the entire circuit, but none of them discusses how these component delays are derived initially. We present a method for RC timing analysis (i.e., computing the delay of a transistor network from the resistances and capacitances of its parts) that can be used to compute these macro-component delays.

The most widely-used approach for modeling parameter uncertainty within existing simulation systems is the Monte Carlo method [Sobo75]. With this technique, analysis is repeated for random combinations of values chosen from within the acceptable range of each parameter. Once computed, these results are tabulated and returned. Unfortunately, accurately determining bounds on the behavior of a circuit requires a large number of trials; thus, while Monte Carlo simulation can be effective, it is also very time consuming. Our goal is to achieve similar accuracy in less time.

In this chapter, we present a theoretical framework based on interval algebra for handling parameter uncertainty in RC analysis. We then illustrate our approach by modifying an existing timing analyzer, Crystal [Oust83, Oust84, Oust86], to create an RC analysis algorithm that incorporates this framework. Given a circuit description and bounds on all circuit parameters (transistor lengths, widths, threshold voltage, capacitances, and resistances), our algorithm computes best and worst-case bounds on the delay of a transistor network. Although we employ Crystal for these experiments, the framework may also be applied to other RC analysis algorithms. We conclude this chapter with an analysis of our method and a comparison with Monte Carlo simulation.

3.1 RC Analysis in Crystal

Crystal is a data-independent timing analyzer for sequential MOS VLSI circuits. Given a design description extracted from a mask layout, Crystal computes and returns a list of the *critical* paths through the circuit. This algorithm has three phases: breaking the circuit into pieces, estimating delay through the pieces (using RC analysis), and using these delays to locate critical paths.

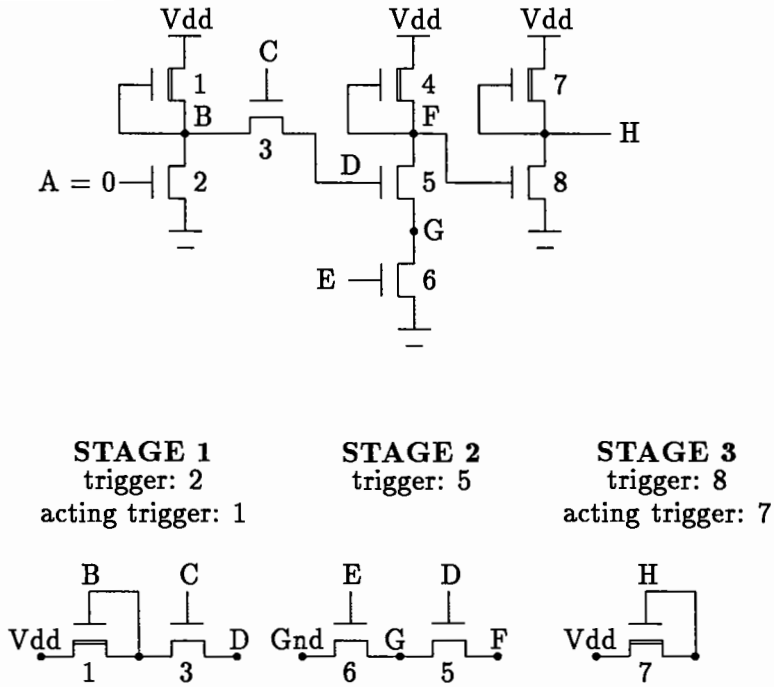


Figure 3.1: Decomposing a Circuit into Stages

In the first phase, Crystal decomposes the circuit into chains of transistors called *stages*. Each stage represents an electrical path through one or more transistors originating at a strong signal source (Vdd or Gnd) and terminating at an output node or transistor gate (called the *target*). A stage is activated by the last transistor in the path to switch on. This transistor is called the *trigger*. In nMOS stages containing depletion load transistors, the trigger transistor is not one of the transistors in the stage; therefore, the depletion load transistor closest to the target acts as the trigger for the stage. For example, stage 1 in Figure 3.1 is activated by a falling signal value at the gate of transistor 2. Since transistor 2 is not on the electrical path from the signal source Vdd to the target node D for the stage, depletion load transistor 1 acts as the trigger. The target transistor for one stage becomes the trigger transistor for the next stage. For an example of circuit decomposition, see Figure 3.1.

In the second phase of critical path analysis, Crystal estimates the delay of each stage. Given the sizes and types of the transistors in the stage, the parasitic resistances

and capacitances of the nodes, the trigger transistor, and the waveform at the gate of the trigger transistor, a delay modeler generates the resulting waveform at the target of the stage. Crystal supports three different models for computing stage delay: the linear RC model, the Slope model, and the PR-Slope model. Of the three models, the PR-Slope model is the most accurate. We will discuss this method in detail in the next section.

In the final phase of analysis, Crystal employs these stage delays to locate critical paths in the design using a longest-path algorithm with a depth-first scanning order. For further details, see [Oust83].

3.1.1 The PR-Slope Model

To estimate delay, the PR-Slope model begins by determining a resistance and capacitance for each node and transistor along the stage. Node capacitances and resistances can either be provided by the circuit extractor or, in some older systems, computed by Crystal directly from the geometry of the node. In the latter case, Crystal is given an area and perimeter for each substrate layer in the node and computes the capacitance and resistance contribution of each. The substrate resistances and capacitances are then summed to compute the resistance and capacitance of the node. In addition to the above contributions, the gate-to-source capacitances of adjacent branch transistors are also included in the node calculations.

A transistor is modeled as a perfect switch in series with a resistor. This resistance is fixed for non-trigger transistors and is computed from the transistor type and geometry. Specifically, a resistance factor is determined by table lookup on transistor type, usage, and signal value. This factor is then multiplied by the length/width ratio of the transistor to generate an estimate of its effective resistance.

The effective resistance of the trigger transistor is more accurately modeled. This resistance is not a constant, but a function of the input waveform at the gate of the transistor. In particular, resistance is a function of the *rise time* of the gate signal, which is the rate at which the signal is changing when it crosses the threshold voltage. The faster the gate signal changes, the lower the effective resistance across the transistor. To model this effect, the timing analyzer combines the rise time at the transistor gate, the load being driven, and the transistor size to compute a *rise time ratio.* It then consults a table indexed by these rise time ratios to find the resistance factor of the

trigger transistor. This factor is multiplied by the aspect ratio of the transistor to determine its resistance.

To calculate the effective capacitance of each transistor, Crystal determines the capacitance factor by table lookup on transistor type and multiplies this value by the area of the transistor. All transistor characterization tables used in these computations are generated by running SPICE simulations on sample circuits and compiling the results.

Having calculated the resistances and capacitances of each node and transistor in the path, Crystal uses these values to compute the total delay of the stage. The PR-Slope model employs the RC equations presented by Penfield and Rubinstein in [Rubi83]. Since a Crystal stage is just an RC line, these delay equations simplify to the following sum:

$$D = \sum_{t<i\leq T} \sum_{S<j\leq i} C_i R_j \tag{3.1}$$

where t represents the trigger transistor, T the target node, S the source node, R_j the resistance of element j, and C_i the capacitance at element i. For example, the delay of Stage 1 in Figure 3.1 is given by the following equation:

$$D = C_B(R_1 + R_B) + C_3(R_1 + R_B + R_3) + C_D(R_1 + R_B + R_3 + R_D). \tag{3.2}$$

In this example, the node capacitances and resistances are denoted by alphabetic subscripts and the transistor capacitances and resistances are denoted by numeric subscripts.

3.2 Bounds for RC Analysis

The PR-Slope model described above uses the average resistance and capacitance values of circuit devices to compute a single estimate of circuit delay. Our goal, however, is to model the effects of device parameter variation on delay; therefore, we must modify this RC analysis algorithm to compute delay bounds. We begin with a few definitions. Let $F(x_1, x_2, ..., x_n)$ be any real-valued, arithmetic function of n independent variables such that $F : \Re^n \rightarrow \Re$ and $x_i \in \Re$. Now assume that the value of each variable, x_i, is not precisely known. Instead, we are given a range of possible values for each x_i, $\hat{x}_i = [min_i, max_i]$. The tightest bounds on the solutions to F are found by evaluating the function over all possible combinations of input values and taking the union of the

results. This is called the *United Extension of F* [Moor79, p. 18] and is defined by the following:

$$\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \equiv \bigcup_{x_i \in \hat{x}_i} F(x_1, x_2, ..., x_n). \tag{3.3}$$

For timing analysis, $F(x_1, x_2, ..., x_n)$ represents delay Equation 3.1 and the x_i represent the capacitances, resistances, and transistor sizes used to compute this delay. If the value of any of these variables is not precisely known, then the tightest bounds on the solution may be found by computing $\hat{F}$. Because the intervals are defined over $\Re$, each contains an infinite number of possible assignments; therefore, we cannot compute $\hat{F}$ directly. We can approximate $\hat{F}$ by dividing each interval into $k-1$ subintervals and evaluating F at the k endpoints. Since each variable x_i can assume any one of these k possible values, we must evaluate function F k^n times to compute its United Extension. Even for $k = 2$, this computation requires 2^n invocations of F. Since the number of variables n tends to be large even for small stages (e.g., the PR-Slope Model requires six variables per transistor and a minimum of two variables per node), this is not a feasible approach.

3.2.1 A First Approximation

To approximate $\hat{F}$, we create an interval version of the delay equation. Let I be the set of all intervals over $\Re$; thus, $I \equiv \{[a,b] \mid a, b \in \Re\}$. To compute stage delay, we require an interval arithmetic. The needed interval arithmetic operations over the set I are addition ($\oplus$), subtraction ($\ominus$), multiplication ($\odot$), and division ($\oslash$) defined as follows:

$$[a,b] \oplus [c,d] \equiv [a+c, b+d] \tag{3.4}$$

$$[a,b] \ominus [c,d] \equiv [a-d, b-c] \tag{3.5}$$

$$[a,b] \odot [c,d] \equiv [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \tag{3.6}$$

$$[a,b] \oslash [c,d] \equiv [a,b] \odot [1/c, 1/d] \quad \textit{provided that } 0 \notin [c,d] \tag{3.7}$$

The properties of these operators are summarized in the introduction of this thesis. Now let $\mathcal{F} : I^n \rightarrow I$ be the interval equivalent of F derived by replacing the basic arithmetic operators $+$, $-$, $\cdot$, and $/$ in F by their respective interval counterparts $\oplus$, $\ominus$, $\odot$, and $\oslash$. This creates an interval version of the original function that returns bounds on the result when evaluated. We now show that $\mathcal{F}$ approximates $\hat{F}$.

Lemma 3.1 *If $\hat{x}_1, ..., \hat{x}_n \in I$ are degenerate, then $F(x_1, ..., x_n) = \mathcal{F}(\hat{x}_1, ..., \hat{x}_n)$.*

Proof. By induction on the number of operators (m) in F. By convention, if $\hat{x} = [x, x]$, then we write $\hat{x} = x$. Over the set of degenerate intervals, each interval operator is equivalent to its respective discrete operator as shown by the following: for all degenerate $\hat{a}, \hat{b} \in I$

$$\hat{a} \oplus \hat{b} = [a, a] \oplus [b, b] = [a + b, a + b] = a + b$$

$$\hat{a} \ominus \hat{b} = [a, a] \ominus [b, b] = [a - b, a - b] = a - b$$

$$\hat{a} \odot \hat{b} = [a, a] \odot [b, b] = [ab, ab] = ab$$

$$\hat{a} \oslash \hat{b} = [a, a] \oslash [b, b] = [a/b, a/b] = a/b.$$

Note also that the set of degenerate intervals is closed over these interval operators; thus, when evaluated over this set, $\mathcal{F}$ always returns a degenerate interval. For the basis ($m = 1$), F must be one of the following four functions: $F = a + b$, $F = a - b$, $F = ab$, or $F = a/b$. We have just demonstrated that $F = \mathcal{F}$ over the set of degenerate intervals for each of these cases. Now we are given that $F = \mathcal{F}$ over the set of degenerate intervals for all functions F consisting of less than m operators. We must show that $F = \mathcal{F}$ for functions with m operators. Any arithmetic function can be written as a binary parse tree, where each parent node contains one of the four arithmetic operators and each child node contains either a value or a subtree representing a subexpression of the function. If we remove the root of the parse tree, we are left with two subtrees representing subfunctions. If the original function had m operators, each of these subfunctions must contain less than m operators, since we removed the operator at the root of the parse tree. Let F represent the original function, let F_1 and F_2 represent these subfunctions, and let $*$ be the operator at the root of F. Clearly, $F = F_1 * F_2$. By the definition of $\mathcal{F}$, we know that $\mathcal{F} = \mathcal{F}_1 \circledast \mathcal{F}_2$ where $\circledast$ is the interval counterpart of the $*$ operator. Applying the assumption, we see that $\mathcal{F}_1 = F_1$ and $\mathcal{F}_2 = F_2$ over the set of degenerate intervals. Since $\circledast$ is equivalent to $*$ over the set of degenerate intervals, we conclude that $F = \mathcal{F}$ over this set. □

Theorem 3.1 $\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$

Proof. Let $y_1, y_2, ..., y_n$ be any legal assignment of values to the variables $x_1, x_2, ..., x_n$. For each y_i, $y_i = \hat{y}_i = [y_i, y_i] \in \hat{x}_i = [min_i, max_i]$; therefore, $\hat{y}_i \subseteq \hat{x}_i$. Hence, by the principle of monotonic inclusion, $\mathcal{F}(\hat{y}_1, \hat{y}_2, ..., \hat{y}_n) \subseteq \mathcal{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$. Applying

Lemma 3.1, we note that $F(y_1, y_2, ..., y_n) = \mathcal{F}(\hat{y}_1, \hat{y}_2, ..., \hat{y}_n)$. Since our choice of values was not special, we conclude that $F(y_1, y_2, ..., y_n) \subseteq \mathcal{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$ for all legal assignments $y_1, y_2, ..., y_n$. Since $\hat{F}$ is just the union of the values $F(y_1, y_2, ..., y_n)$, we know that $\hat{F} \subseteq \mathcal{F}$. □

According to the above theorem, we can approximate the United Extension of our delay function by replacing the discrete mathematical operators in the function with the corresponding interval ones. This substitution yields an interval version of the algorithm that computes bounds on the delay of the circuit much more efficiently than exhaustive enumeration.

3.2.2 Improving these Bounds

Up to this point, our discussion has been general and our proofs apply to any real-valued, arithmetic function F. Now, we discuss a particular example: delay equation 3.1. Let D denote this function, let $\hat{D}$ represent its united extension, and let $\mathcal{D}$ be the corresponding interval delay function. While $\mathcal{D}$ does indeed yield correct bounds on the delay of the circuit, they tend to be overly conservative. One reason for this is the lack of multiplicative inverses in the interval algebra. For example, let $\hat{x} = [a, b]$, then $\hat{x} \oslash \hat{x} = [a/b, b/a]$. This yields the expected result $[1, 1]$ only when $a = b$; however, $[1, 1] \subseteq \hat{x} \oslash \hat{x}$ for all $\hat{x}$. Crystal's delay equation contains several such instances. One occurs when we compute the RC contribution of transistor j as in the following:

$$\begin{aligned} C_j \odot R_j &= CperArea_j \odot Area_j \odot RperSquare_j \odot Aspect_j \\ &= CperArea_j \odot L_j \odot W_j \odot RperSquare_j \odot L_j \oslash W_j \\ &= CperArea_j \odot RperSquare_j \odot L_j^2 \odot W_j \oslash W_j \\ &\supseteq CperArea_j \odot RperSquare_j \odot L_j^2 \end{aligned}$$

To alleviate this problem, we must carefully tune the delay equations to eliminate any unnecessary divisions. Specifically, we rewrite function D to create an equivalent version D_2 by expanding the RC terms and eliminating as many of these divisions as possible. For example, the RC terms for transistor j are transformed as follows:

$$\begin{aligned} C_j(R_1 + ... + R_j) &= C_j(R_1 + ... + R_{j-1}) + C_j \cdot R_j \\ &= C_j(R_1 + ... + R_{j-1}) + CperArea_j \cdot RperSquare_j \cdot L_j^2 \end{aligned}$$

Similarly, the RC terms for node k are rewritten as follows:

$$\begin{aligned}
C_k(R_1 + ... + R_k) &= C_k(R_1 + ... + R_{k-2}) + C_k \cdot R_{k-1} + C_k \cdot R_k \\
&= C_k(R_1 + ... + R_{k-2}) + C_k \cdot R_k + \\
&\quad (CRest_k + CperWidth_{k-1} \cdot W_{k-1})R_{k-1} \\
&= C_k(R_1 + ... + R_{k-2}) + CRest_k \cdot R_{k-1} + C_k \cdot R_k + \\
&\quad CperWidth_{k-1} \cdot W_{k-1} \cdot R_{k-1} \\
&= C_k(R_1 + ... + R_{k-2}) + CRest_k \cdot R_{k-1} + C_k \cdot R_k + \\
&\quad CperWidth_{k-1} \cdot W_{k-1} \cdot RperSquare_{k-1} \cdot Aspect_{k-1} \\
&= C_k(R_1 + ... + R_{k-2}) + CRest_k \cdot R_{k-1} + C_k \cdot R_k + \\
&\quad CperWidth_{k-1} \cdot RperSquare_{k-1} \cdot L_{k-1}
\end{aligned}$$

where $C_k = CRest_k + CperWidth_{k-1} \cdot W_{k-1}$. In other words, $CperWidth_{k-1} \cdot W_{k-1}$ is the portion of node k's capacitance contributed by transistor $k-1$ and $CRest_k$ is the remainder.

Once we have D_2, we can replace the arithmetic operators in it with the corresponding interval operators to create an interval version, $\mathcal{D}_2$. Although the delay functions D and D_2 are equivalent, their interval counterparts $\mathcal{D}$ and $\mathcal{D}_2$ are not. Since $\mathcal{D}_2$ contains fewer interval divisions, the bounds that it produces are always as good as those produced by $\mathcal{D}$. Often they are tighter. In our next theorem, we show that $\mathcal{D}_2$ approximates $\hat{D}$, producing bounds that are at least as good as $\mathcal{D}$.

Theorem 3.2 $\hat{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}_2(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$

Proof. We begin by proving that $\hat{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}_2(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$. Applying Theorem 3.1, we note that $\hat{D}_2 \subseteq \mathcal{D}_2$. But $D = D_2$; therefore, $\hat{D} \subseteq \mathcal{D}_2$. Now we must show that $\mathcal{D}_2(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subseteq \mathcal{D}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$. Since all our variables (capacitance factors, resistance factors, transistor lengths and widths) are positive real numbers, the special case of the distributive property holds. Specifically,

$$\begin{aligned}
C_k \odot (R_1 \oplus ... \oplus R_k) &= C_k \odot (R_1 \oplus ... \oplus R_{k-1}) \ \oplus \ C_k \odot R_k \\
&= C_k \odot (R_1 \oplus ... \oplus R_{k-2}) \ \oplus \ C_k \odot R_{k-1} \ \oplus \ C_k \odot R_k.
\end{aligned}$$

Hence $\mathcal{D}$ only differs from $\mathcal{D}_2$ in the simplified RC terms. We noted above that $[1,1] \subseteq \hat{x} \oslash \hat{x}$; therefore, by monotonic inclusion,

$$CperArea_k \odot RperSquare_k \odot L_k^2 \subseteq CperArea_k \odot Area_k \odot RperSquare_k \odot Aspect_k$$

$$CperWidth_k \odot RperSquare_k \odot L_k \subseteq CperWidth_k \odot W_k \odot RperSquare_k \odot Aspect_k$$

Applying the principle of monotonic inclusion again, we conclude that $\mathcal{D}_2 \subseteq \mathcal{D}$. □

3.3 Performance

Using the revised interval delay function $\mathcal{D}_2$, we have implemented an interval version (Intcryst) of the RC analysis algorithm. We also implemented a Monte Carlo version of D for comparison. To select values from the range of each variable, the Monte Carlo program employs a random number generator with a uniform distribution. It then uses the original delay function D to compute the delay for each trial and returns the minimum, maximum, and average delays calculated. Both implementations were written in the C programming language and run on a SUN SPARCstation. We now present an analysis of the performance of our interval algorithm as compared to Monte Carlo simulation.

3.3.1 Quality of the Bounds

To determine the quality of the bounds produced by both the Monte Carlo (50,000 trials) and interval methods, we conducted a number of experiments comparing their bounds to the theoretical bounds ($\hat{D}$) for the given circuit. For these experiments, the theoretical bounds were approximated by running a Monte Carlo simulation with an unlimited number of trials. We also tested nominal delay (i.e., the delay computed using the average values of the input parameters). For all methods, delay was calculated in two stages. First, we determined the resistance and capacitance factors for all transistors by table lookup. Then we used the values obtained to compute delay.

The first experiment illustrates the relationship between delay and stage length (i.e., the number of transistors in a stage) for a family of nMOS circuits. As shown in Figure 3.2, the Monte Carlo and nominal delay methods always underestimate the theoretical bounds, while the interval approach always overestimates them. The graph in Figure 3.3 depicts the average percent error between the theoretical delay bounds and the bounds generated using the other methods (i.e., error = theoretical upper (lower) bound - estimated upper (lower) bound/ theoretical upper (lower) bound). As shown in Figure 3.3, both the Monte Carlo and interval methods produce bounds with similar accuracy, coming within 10% of the theoretical bounds for all stage lengths tested. The

Stage Length	Intcryst		Monte Carlo		Nominal	
	lower	upper	lower	upper	lower	upper
1	0	0	-4.42	3.91	-18.98	16.08
3	0.26	-0.24	-4.87	4.67	-16.23	13.49

Table 3.1: Average % Error for Upper and Lower Delay Bounds

graph further shows that the error of both methods grows with the length of the stage. Fortunately, stages encountered in most circuits are typically small, containing fewer than five transistors. As the graph demonstrates, nominal delay, the method currently employed in most timing analyzers, provides the poorest estimate of worst-case timing behaviors.

We also analyzed a portion of the Brown Systolic Array (B-SYS)[Hugh89], a 2-μ CMOS design. In particular, we surveyed 33 stages extracted from the ALU of the chip. The stages ranged in length from one to three transistors, with an average length of 2.45 transistors. The average percent error for each method is displayed in Table 3.1. As shown, Intcryst produced the tightest bounds, coming within 0.3% of the theoretical bounds on average. The Monte Carlo method (10,000 trials) produced an average error of about 4.6%, while the nominal delay had an average error of 15.6%.

The third example is an actual critical path computation taken from the B-SYS chip. In this case, we analyzed a portion of the circuit containing approximately 2,000 transistors. The critical path returned was determined to have 9 stages. Table 3.2 displays the delay bounds computed for each stage and the total for the critical path. Table 3.3 gives the percent error in stage delay for each method. As shown, the results are similar to those obtained in the previous experiment.

3.3.2 Running Time of the Algorithm

We used two measures to compare the speeds of the various implementations. First, we counted the occurrences of each arithmetic operation in each implementation. Table 3.4 contains a breakdown of operations for each of the three functions tested – the original delay function D used by Crystal and the Monte Carlo program, the revised delay function D_2, and the interval version, $\mathcal{D}_2$. In the analysis, N represents the number of transistors in a stage. The table shows that the interval version of the PR-Slope model

Stage	Size	Theoretical		Intcryst		Monte Carlo		Nom
		min	max	min	max	min	max	
1	2	6.8	9.3	6.8	9.3	7.2	8.7	8.0
2	3	54.3	72.9	54.1	73.1	56.6	69.5	63.0
3	2	10.3	14.1	10.3	14.1	10.8	13.4	12.1
4	2	10.6	14.4	10.6	14.4	11.0	13.9	12.4
5	2	1.2	1.6	1.2	1.6	1.2	1.5	1.4
6	1	0.7	1.0	0.7	1.0	0.8	0.9	0.8
7	1	0.8	1.1	0.8	1.1	0.8	1.0	0.9
8	3	20.4	27.4	20.3	27.5	21.2	26.5	23.7
9	1	2.0	2.7	2.0	2.7	2.0	2.6	2.3
Path total		107.1	144.5	106.8	144.8	111.7	138.1	124.5

Table 3.2: B-SYS Critical Path Computation

Stage	Size	Intcryst		Monte		Nominal	
		min	max	min	max	min	max
1	2	0.3	-0.3	-5.9	5.9	-16.7	14.0
2	3	0.3	-0.3	-4.2	4.7	-16.0	13.6
3	2	0.1	-0.1	-4.8	5.3	-17.0	14.3
4	2	0	-0.1	-3.8	3.7	-16.8	14.1
5	2	0.9	-0.6	-5.2	5.7	-18.1	15.4
6	1	0	0	-7.1	5.1	-18.6	15.3
7	1	0	0	-3.9	4.6	-19.5	15.6
8	3	0.4	-0.4	-4.1	3.1	-16.0	13.6
9	1	0	0	-1.5	1.5	-16.2	13.6
Path total		0.2	-0.3	-4.3	4.4	-16.3	13.8

Table 3.3: % Error in Critical Path Delay

Algorithm	Number of Operations			
	+	−	·	/
D (original)	$10N+1$	4	$9N+3$	$N+2$
D_2 (revised)	$18N-2$	4	$23N-10$	$2N+3$
$\mathcal{D}_2$ (intcryst)	$36N-4$	8	$46N-20$	$4N+6$

Table 3.4: Worst-Case Operations Count for a Single Stage with N Transistors

contains approximately four times as many operations in each category as the original delay function; therefore, we expect that this algorithm will require roughly four times longer to compute the stage delay.

To confirm this hypothesis, we ran all three versions on identical circuits and recorded their response times. Our experiments support this conjecture, showing that Intcryst is roughly four times slower than Crystal. Both implementations have $O(N)$ time complexity.

Since the Monte Carlo program uses the original function D, we note that our interval algorithm computes delay bounds for a stage in about the time required to perform four Monte Carlo trials. Our experiments, however, show that 50,000 Monte Carlo trials are often required to produce comparable delay bounds. Our interval algorithm, therefore, offers a vast improvement in speed. To illustrate this, Table 3.5 displays the amount of time each program took to compute delay bounds for the critical path computation presented in Table 3.2. To locate this critical path, Crystal examined almost 1,300 stages. It was able to compute the delay of all these stages in about 0.7 seconds. Intcryst required 2.8 seconds to compute delay bounds for these stages. The Monte Carlo program with 1,000 trials required 12 minutes for its analysis and with 50,000 trials took 10 hours to compute bounds.

3.4 Summary

In this chapter, we presented a method for modeling the effects of uncertainty in RC analysis. Representing uncertain parameters as intervals, we used interval algebra to create a mathematical framework for manipulating these uncertain values. We then modified an existing RC analysis algorithm (Crystal's PR-Slope model) to test our approach. Although Crystal was chosen for our experiments, the same techniques may

Method	No. of trials	Time
Crystal	1	0.7 sec
Intcryst	1	2.8 sec
Monte Carlo	1000	12.0 min
Monte Carlo	5000	1 hr
Monte Carlo	10000	2 hrs
Monte Carlo	50000	10 hrs

Table 3.5: Relative Speed of Each Algorithm for a Critical Path Computation

be applied to other timing analysis algorithms.

Given a transistor network with uncertain input parameters, our interval algorithm computes bounds on the delay. We claimed that the delay estimates produced by our algorithm approximate the theoretical delay bounds for the circuit and provided proofs to support our hypothesis. We then implemented our algorithm and tested its performance. Analysis shows that its operating speed is not significantly slower than that of the non-interval RC analysis algorithm.

When compared to Monte Carlo simulation, our algorithm is much more efficient, operating several orders of magnitude faster for the same quality results. For stages with few transistors, the accuracy of both methods is similar: both come within 10% of the theoretical bounds in the worst-case. In the next chapter, we present other interval-based methods that further improve the accuracy of these bounds.

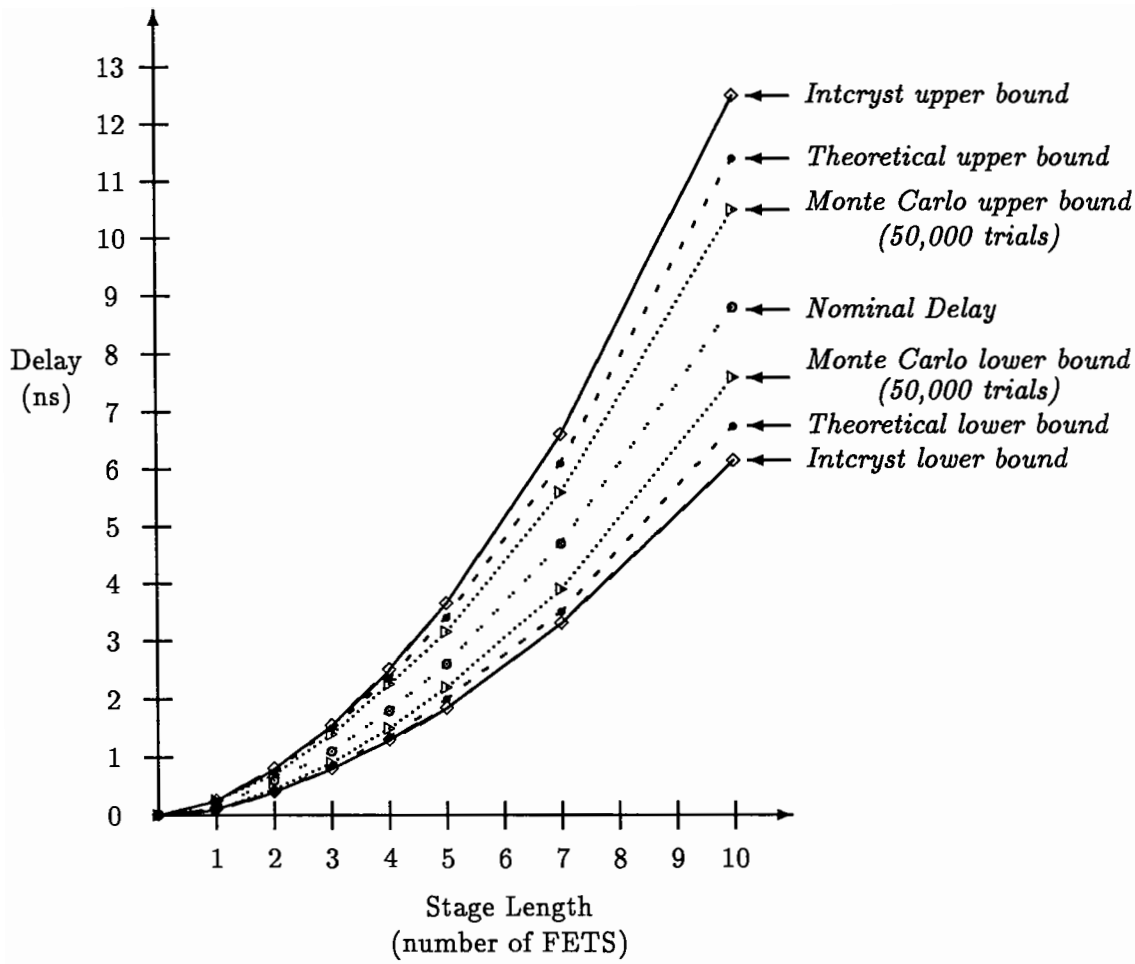


Figure 3.2: Delay vs. Stage Length

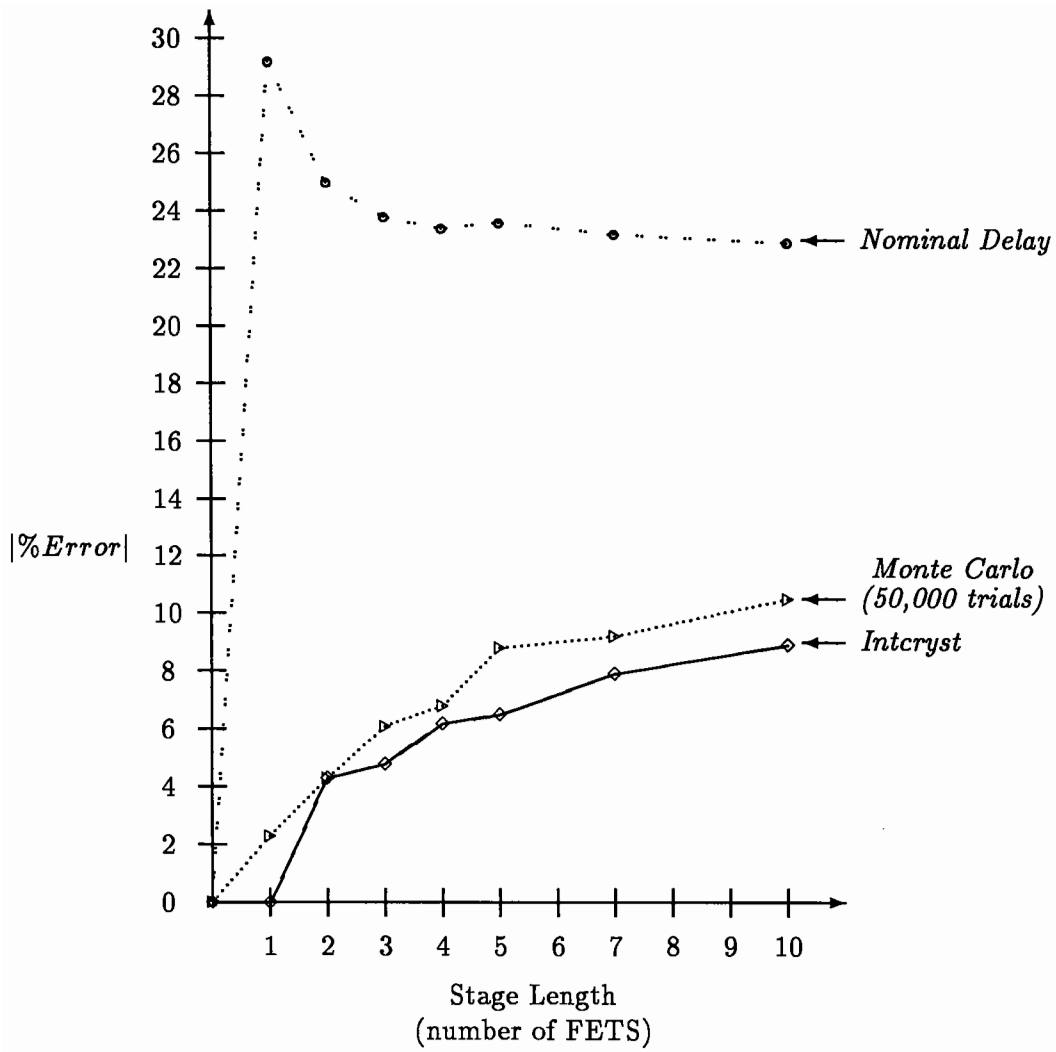


Figure 3.3: % Error vs. Stage Length

Chapter 4

Modeling Uncertainty in RC Timing Analysis II

In the previous chapter, we presented a method based on interval arithmetic for modeling the effects of uncertainty in RC analysis. Using this technique, we implemented Intcryst, an interval version of Crystal's PR-Slope model [Oust83, Oust84, Oust86], and compared the bounds that it generated to the theoretical bounds ($\hat{D}$) for several circuits. While the interval results always contain the theoretical bounds, they are often overly conservative. The reason for this is that the interval algebra ignores variable interdependencies. Each occurrence of a variable is considered independent of all other occurrences. For example, consider the function $F(x) = x + 1/x$ where $\hat{x} = [0.5, 2]$. The United Extension $\hat{F}$ of this function is $[2, 2.5]$, but the interval solution is $[1, 4]$. The interval upper bound of four results from using the maximum value of x for the first occurrence of the variable and its minimum value for the second (i.e., $2 + 1/0.5 = 4$). If we wish to produce tighter delay bounds, we must develop alternative methods that account for variable interdependence.

In this chapter, we explore two other *general* interval-based techniques for approximating the United Extension of a function. The first of these methods employs centred forms and the second uses interval differential calculus, an interval extension of real differential calculus. To illustrate these methods, we incorporate them into Crystal's PR-slope model to create two new algorithms for computing bounds on circuit delay. Although Crystal was chosen for these experiments, the same techniques can be applied to other timing verifiers. Finally, we analyze the performance of these methods,

comparing them to our original interval algorithm and with Monte Carlo simulation.

4.1 Krawczyk's Centred Form

Centred forms provide an alternative method for approximating the United Extension, $\hat{F}$. While there are many types of centred forms (see [Rats84]), all share a common framework and structure. Given a real-valued function $F(x_1, x_2, ..., x_n)$ and the interval range $\hat{x}_i$ of each variable x_i, all methods begin by choosing a particular set of values, $c_1, c_2, ..., c_n$, such that each $c_i \in \hat{x}_i$ and evaluating $F(c_1, c_2, ..., c_n)$. While any choice $c_i \in \hat{x}_i$ can be used, the midpoint of each interval is typically chosen. They then rewrite the function F to create an equivalent expression (denoted F_C) which has the following structure:

$$F_C(x_1, x_2, ..., x_n) \equiv F(c_1, c_2, ..., c_n) + G(x_1 - c_1, x_2 - c_2, ..., x_n - c_n) \tag{4.1}$$

This new representation is known as the *General Centred-Form of F.*

Centred forms differ from one another in their choice of the function G. For instance, *standard centred forms* use a Taylor series expansion of order k to create G. The *mean-value centred form* uses derivatives to create G; thus, $G \equiv F'(x) \cdot (x - c)$. *Krawczyk's centred form* uses interval slopes. With this method, there is no general closed form for G; instead G is computed directly from F by construction.

Having transformed F into centred form F_C, they then create an interval version of F_C. This is accomplished by replacing each variable x_i in F_C by its interval range $\hat{x}_i$ and each arithmetic operator by its corresponding interval operator. When evaluated, this new interval function produces symmetric bounds centered at $F(c_1, c_2, ..., c_n)$ containing $\hat{F}$. Although function F and its centred form F_C are equivalent, their interval extensions $\mathcal{F}$ and $\mathcal{F}_C$ are not. Since the resulting interval versions contain different numbers and combinations of the arithmetic operations, the accumulation of error is different. In most cases, the centred forms produce tighter bounds.

Each of the centred forms presented above yield bounds which subsume the true bounds $\hat{F}$; thus, $\hat{F} \subseteq \mathcal{F}_C$. Proofs of convergence for all of these centred methods are found in [Rats84] and will not be repeated here.

In choosing a centred form for RC timing analysis, we evaluated each of the methods presented in [Rats84]. With standard centred forms, we would have to precompute a different closed form function G for each function F. In RC analysis, the delay equation

varies with the number of transistors in the stage (called the *stage length*); therefore, to use a standard centred form, we must generate a closed centred expression for each possible stage length encountered. We require a more flexible approach. Mean-value centred forms are one alternative. Ratschek and Rokne, however, claim that this method produces poorer bounds estimates than the higher-order standard centred forms [Rats84, pp. 78-81]. Unlike the standard forms, Krawczyk's centred form does not have to be precomputed and can be easily programmed for any function. The flexibility of this format makes it an ideal candidate for the timing analysis problem.

Krawczyk's centred form [Rats84, pp. 59-62] calculates G directly from function F by construction. We therefore require a straight-line program for computing F. Let $B = \{b_1, b_2, ..., b_r\}$ be the set of real constants that appear in F, let $X = (x_1, x_2, ..., x_n)$ be the vector of independent variables in F, and let $U = \{u_1, u_2, ..., u_s\}$ be a finite set dependent variables used to compute F. A straight-line program is a sequence of s computational steps for calculating the value of a function and is defined as follows:

$u_i = x_i$	for $i = 1, ..., n$	/* load variables */
$u_i = b_{i-n}$	for $i = n+1, ..., n+r$	/* load constants */
$u_i = u_j *_i u_k$	for $i = n+r+1, ..., s$	/* calculate F */
	where $j, k < i$ and $*_i \in \{+, -, \cdot, /\}$	

For example, if $F(x) = x + 1/x$, then the following sequence of instructions constitutes one possible program for computing F:

$$\begin{aligned} u_1 &= x \\ u_2 &= 1 \\ u_3 &= u_2/u_1 \\ u_4 &= u_1 + u_3 \end{aligned}$$

Given a straight-line program for computing F, we employ it to compute the interval slope G. Let F_i denote the function corresponding to the ith step of the straight line program (e.g., $F_1 = x$, $F_2 = 1$, $F_3 = 1/x$, and $F_4 = x + 1/x$). Likewise, let $\mathcal{F}_i$ denote the interval extension of F_i (e.g., $\mathcal{F}_1 = \hat{x}$, $\mathcal{F}_2 = [1,1]$, $\mathcal{F}_3 = [1,1] \oslash \hat{x}$, and $\mathcal{F}_4 = \hat{x} \oplus [1,1] \oslash \hat{x}$). Define $E_1, E_2, ..., E_n \in I^n$ to be n-dimensional interval unit vectors such that $E_1 = ([1,1],[0,0],...,[0,0])$, $E_2 = ([0,0],[1,1],...,[0,0])$, ..., $E_n = ([0,0],...,[0,0],[1,1])$. Likewise, let $O = ([0,0],[0,0],...,[0,0]) \in I^n$ be an n-dimensional

i	u_i	$F_i(c)$	$\mathcal{F}_i(\hat{x})$	G_i
1	x	1.25	[0.5,2]	[1,1]
2	1	1	[1,1]	[0,0]
3	u_2/u_1	0.8	[0.5,2]	[-1.6,-0.4]
4	u_1+u_3	**2.05**	**[1,4]**	**[-0.6,0.6]**

Table 4.1: Computing the Interval Slope G

interval zero vector. The interval slope $G = (\hat{g}_1, \hat{g}_2, ..., \hat{g}_n)$ is defined recursively from the s-instruction straight-line program as follows:

$G_i = E_i$	if $u_i = x_i$	/* load variables */
$G_i = O$	if $u_i = b_{i-n}$	/* load constants */
$G_i = G_j \oplus G_k$	if $u_i = u_j + u_k$	/* addition */
$G_i = G_j \ominus G_k$	if $u_i = u_j - u_k$	/* subtraction */
$G_i = G_j \odot \mathcal{F}_k(\hat{X}) \oplus G_k \odot F_j(C)$	if $u_i = u_j u_k$	/* multiplication */
$G_i = G_j \oslash \mathcal{F}_k(\hat{X}) \ominus G_k \odot F_i(C) \oslash \mathcal{F}_k(\hat{X})$	if $u_i = u_j/u_k$	/* division */

The recursion terminates after s steps, the number of instructions in the program, and the interval slope $G = G_s$. For example, let $F(x) = x + 1/x$, $\hat{x} = [0.5, 2]$, and $c = 1.25$. Using the function procedure given above, we can derive the interval slope G for this function. The results of each computational step are shown in Table 4.1. The last row of the table displays the resulting interval slope.

Having found G, we can construct Krawczyk's centred form $\mathcal{F}_C$ for function F using the following formula:

$$\mathcal{F}_C(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \equiv F(c_1, c_2, ..., c_n) \oplus \sum_{i=1}^{n} \hat{g}_i \odot (\hat{x}_i \ominus c_i). \tag{4.2}$$

Using the results in Table 4.1, this equation yields $\mathcal{F}_C = [1.6, 2.5]$. This interval is centered at $F(c) = 2.05$ and represents the smallest such interval containing the theoretical bounds $[2, 2.5]$. Recall that the interval bounds $\mathcal{F}$ for the same function are $[1, 4]$. In this case, the centred form produces tighter bounds.

This algorithm has a time complexity of $O(sn)$ where s represents the number of instructions in the program for computing F and n represents the number of independent variables.

4.2 Interval Differential Calculus

4.2.1 Rall's Algorithm

The second method for approximating $\hat{F}$ employs interval differential calculus, which was introduced by Rall [Rall86]. Given a function $F(x)$ over $\Re$ and an interval $\hat{x} = [a, b]$, Rall uses differential calculus to compute $\hat{F}$. The derivative F' provides information about the monotonicity of the function which in turn can be used to obtain a better approximation to $\hat{F}$. For instance, if $F'(x) \geq 0$ for all $x \in \hat{x}$, then function F is monotone increasing over the interval $\hat{x}$ and

$$\hat{F}(\hat{x}) = [F(a), F(b)]. \tag{4.3}$$

Likewise, if $F'(x) \leq 0$ for all $x \in \hat{x}$, then the function is monotone decreasing over the interval $\hat{x}$ and

$$\hat{F}(\hat{x}) = [F(b), F(a)]. \tag{4.4}$$

Rall further demonstrates that $F'(x)$ for all $x \in \hat{x}$ can be approximated using interval differential arithmetic, a natural extension of real differential arithmetic.

The operands for real differential arithmetic are ordered pairs in $\Re^2$ such that the first element of the pair contains a variable or function and the second element its derivative. If $X = (x, x')$ and $Y = (y, y')$ are two such operands, then the rules of real differential arithmetic are:

$$X + Y = (x, x') + (y, y') = (x + y, x' + y') \tag{4.5}$$

$$X - Y = (x, x') - (y, y') = (x - y, x' - y') \tag{4.6}$$

$$X \cdot Y = (x, x') \cdot (y, y') = (x \cdot y, x \cdot y' + y \cdot x') \tag{4.7}$$

$$X/Y = (x, x')/(y, y') = (x/y, (x' - (x/y) \cdot y')/y), \quad y \neq 0 \tag{4.8}$$

The interval differential algebra is similar except that all operands are interval pairs and the basic arithmetic operators $+$, $-$, $\cdot$, and $/$ have been replaced by their interval equivalents $\oplus$, $\ominus$, $\odot$, and $\oslash$. If $\hat{X} = (\hat{x}, \hat{x}')$ and $\hat{Y} = (\hat{y}, \hat{y}')$, then the interval differential operators are:

$$\hat{X} \oplus \hat{Y} = (\hat{x}, \hat{x}') \oplus (\hat{y}, \hat{y}') = (\hat{x} \oplus \hat{y}, \hat{x}' \oplus \hat{y}') \tag{4.9}$$

$$\hat{X} \ominus \hat{Y} = (\hat{x}, \hat{x}') \ominus (\hat{y}, \hat{y}') = (\hat{x} \ominus \hat{y}, \hat{x}' \ominus \hat{y}') \tag{4.10}$$

$$\hat{X} \odot \hat{Y} = (\hat{x}, \hat{x}') \odot (\hat{y}, \hat{y}') = (\hat{x} \odot \hat{y}, \hat{x} \odot \hat{y}' \oplus \hat{y} \odot \hat{x}') \tag{4.11}$$

```
Current bounds = F((a + b)/2)
Add interval x̂ = [a, b] to evaluation queue
While queue not empty
    Get next (sub)interval ŷ from queue
    Compute F'(ŷ)
    Check range of F'(ŷ)
    If monotone
        Compute exact (sub)interval bounds using Equation 4.3 or 4.4
        Update current bounds
    Else if possible, bisect ŷ into halves ŷ1 and ŷ2
        Add intervals ŷ1 and ŷ2 to evaluation queue
    Else (interval ŷ is too small to bisect)
        Approximate (sub)interval bounds using F(ŷ)
        Update current bounds
Return current bounds
```

Figure 4.1: Rall's Interval Differential Algorithm

$$\hat{X} \oslash \hat{Y} = (\hat{x}, \hat{x}') \oslash (\hat{y}, \hat{y}') = (\hat{x} \oslash \hat{y}, (\hat{x}' \ominus (\hat{x} \oslash \hat{y}) \odot \hat{y}') \oslash \hat{y}) \quad (4.12)$$
$$\textit{where } [0,0] \notin \hat{y}$$

We use these interval differential operators to compute the interval derivative of F, $\mathcal{F}'(\hat{x})$.

Having introduced interval differential calculus, Rall incorporates it into his algorithm to estimate $\hat{F}(\hat{x})$ as summarized in Figure 4.1. We begin by computing the interval derivative of function F. If this derivative indicates that the function is monotone over the interval $\hat{x}$, then we compute $\hat{F}(\hat{x})$ directly using Equation 4.3 or 4.4; otherwise, we partition the interval $\hat{x}$ into two halves and repeat the analysis on each half. The number of interval bisections performed by this algorithm is specified by the user. When this number is exceeded, the bounds must be estimated. In this case, Rall uses $\mathcal{F}$ to estimate subinterval bounds. The final bounds on the function are found by taking the union of all subinterval bounds. The number of estimated subinterval components indicates the quality of the final result. In particular, the fewer the number of non-monotone subintervals, the better the resulting bounds are likely to be.

To illustrate this approach, consider our previous example $F(x) = x + 1/x$ with $\hat{x} = [0.5, 2.0]$ and a bisection limit of 3. The results of each stage of this computation are displayed in Table 4.2. The last column in the table shows the subinterval bounds (if any) computed for each stage. Taking the union of these bounds yields a final result

$\hat{x}$	$\mathcal{F}(\hat{x})$	$\mathcal{F}'(\hat{x})$	Action	$\mathcal{F}_I(\hat{x})$
0.50, 2.00	1.00,4.00	-3.00,0.75	bisect	
0.50, 1.25	1.30,3.25	-3.00,0.36	bisect	
1.25, 2.00	1.75,2.80	0.36,0.75	compute bounds	2.05,2.50
0.50, 0.88	1.64,2.88	-3.00,-0.31	compute bounds	2.02,2.50
0.88, 1.25	1.67,2.39	-0.31,0.36	bisect	
0.88, 1.06	1.82,2.21	-0.31,0.11	estimate bounds	1.82,2.21
1.06, 1.25	1.86,2.19	0.11,0.36	compute bounds	2.00,2.05
TOTAL				**1.82,2.50**

Table 4.2: An Interval Differential Bounds Computation

of $[1.82, 2.50]$. The theoretical bounds for this function are $[2, 2.5]$ and the interval bounds are $[1, 4]$. For this function, the interval differential method $\mathcal{F}_I(\hat{x})$ produced much better results than the original interval approach.

4.2.2 Extensions to Rall's Algorithm

Rall's algorithm only works for single-variable functions, so we have generalized it to create one for multi-variable functions. Given a function $F(x_1, x_2, ..., x_n)$ over $\Re$ and the range of values for each variable $x_i \in \hat{x}_i = [min_i, max_i]$, we wish to approximate $\hat{F}$ using interval differential calculus. For multi-variable functions, we use *partial* derivatives. Let $L = (l_1, l_2, ..., l_n)$ and $U = (u_1, u_2, ..., u_n)$ be lower and upper bound vectors over $\Re^n$. If $\partial F/\partial x_i \geq 0$ for all legal value assignments, then F is monotone increasing with respect to x_i and

$$\begin{aligned} l_i &= min_i \\ u_i &= max_i. \end{aligned} \tag{4.13}$$

Likewise, if $\partial F/\partial x_i \leq 0$ for all legal value assignments, then F is monotone decreasing with respect to x_i and

$$\begin{aligned} l_i &= max_i \\ u_i &= min_i. \end{aligned} \tag{4.14}$$

If the function F is monotone over the ranges of all variables $x_1, x_2, ..., x_n$, then we can precisely compute $\hat{F}$ as follows:

$$\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) = [F(l_1, l_2, ..., l_n), F(u_1, u_2, ..., u_n)] \tag{4.15}$$

We can approximate each partial derivative $\partial F/\partial x_i$ evaluated at all possible assignments $x_j \in \hat{x}_j$ with an interval partial derivative, $\partial \mathcal{F}/\partial x_i$. The interval partial differential algebra needed for this computation is similar to the algebra presented in the previous section. As before, the operands are variable-derivative pairs, but in this case, the derivative is represented as a vector of n partial derivatives (e.g., $\hat{X} = (\hat{x}, (\hat{x}'_1, \hat{x}'_2, ..., \hat{x}'_n))$ where $\hat{x}'_i$ denotes the partial derivative of the i^{th} variable with respect to x). The arithmetic operators for this calculus are defined as follows:

$$\hat{X} \oplus \hat{Y} = (\hat{x} \oplus \hat{y}, (\hat{x}'_1 \oplus \hat{y}'_1, ..., \hat{x}'_n \oplus \hat{y}'_n)) \tag{4.16}$$

$$\hat{X} \ominus \hat{Y} = (\hat{x} \ominus \hat{y}, (\hat{x}'_1 \ominus \hat{y}'_1, ..., \hat{x}'_n \ominus \hat{y}'_n)) \tag{4.17}$$

$$\hat{X} \odot \hat{Y} = (\hat{x} \odot \hat{y}, (\hat{x} \odot \hat{y}'_1 \ \oplus \ \hat{y} \odot \hat{x}'_1, ..., \hat{x} \odot \hat{y}'_n \ \oplus \ \hat{y} \odot \hat{x}'_n)) \tag{4.18}$$

$$\hat{X} \oslash \hat{Y} = (\hat{x} \oslash \hat{y}, ((\hat{x}'_1 \ \ominus \ (\hat{x} \oslash \hat{y}) \odot \hat{y}'_1) \oslash \hat{y}, ..., (\hat{x}'_n \ \ominus \ (\hat{x} \oslash \hat{y}) \odot \hat{y}'_n) \oslash \hat{y})) \quad where \ [0,0] \notin \hat{y} \tag{4.19}$$

Now when F is evaluated using these interval differential operations, the result is the operand $(\mathcal{F}, (\partial \mathcal{F}/\partial x_1, \partial \mathcal{F}/\partial x_2, ..., \partial \mathcal{F}/\partial x_n))$. The C language code for each of these operators in presented in Appendix C.

We contend that these interval partial derivatives may be used to determine the monotonicity of F. Recall that F is monotone increasing (decreasing) with respect to the variable x_i, if $\partial F/\partial x_i \geq 0$ (≤ 0) for all possible assignments $x_j \in \hat{x}_j$. By definition, $\partial F/\partial x_i$ evaluated at all possible variable values is just the United Extension of $\partial F/\partial x_i$. Furthermore, Theorem 3.1 states that the interval extension of a function produces bounds which contain its United Extension; therefore,

$$\hat{\partial F/\partial x_i}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \ \subseteq \ \partial \mathcal{F}/\partial x_i(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n). \tag{4.20}$$

Hence, if $\partial \mathcal{F}/\partial x_i \geq 0$, then $\hat{\partial F/\partial x_i} \geq 0$ and F is monotone increasing with respect to x_i. We thus conclude that the interval partial derivatives computed by our algebra may be employed to determine the monotonicity of the function F.

Having defined the interval partial differential algebra, we are now ready to present our version of the bounds algorithm (See Figure 4.2). The algorithm begins by computing interval partial derivatives for each variable. It then checks the ranges of all partial derivatives for monotonicity, loading the lower L and upper U bounds arrays with the appropriate values. If the function is monotone with respect to all variables, then the true bounds on the function can be precisely found. If, however, the function

```
Current bounds = F(c_1, ..., c_n) where c_i = (min_i + max_i)/2
Add interval vector X̂ = (x̂_1, x̂_2, ..., x̂_n) to evaluation queue
While queue not empty
    Get next entry Ŷ = (ŷ_1, ..., ŷ_n) from queue
    Compute ∂F/∂y_i for all i = 1, ..., n
    Check ranges of all partial derivatives, loading bound arrays L and U
    If F is monotone for all y_i
        Compute exact (sub)interval bounds using Equation 4.15
        Update current bounds
    Else if possible, bisect Ŷ into Ŷ_1 and Ŷ_2
        Add vectors Ŷ_1 and Ŷ_2 to evaluation queue
    Else (intervals are too small to bisect)
        Approximate (sub)interval bounds using F(Ŷ)
        Update current bounds
Return current bounds
```

Figure 4.2: Multi-Variable Interval Differential Algorithm

is non-monotone over the range of some variable, we bisect the intervals and repeat the analysis on the pieces. As before, $\mathcal{F}$ is used to estimate the bounds when the subintervals become too small.

The partitioning method employed in this algorithm is crucial to both its efficiency and correctness. If we subdivide and carelessly group the intervals, we risk underestimating the true bounds. On the other hand, if we are too conservative (e.g., calculate all combinations of subintervals), the problem becomes computationally intractable. We therefore seek a balance between these two extremes. Our approach is to bisect only the non-monotone intervals and group the resulting subintervals such that all the values that are likely to produce upper bounds on the solution are collected together and all the values that are likely to produce lower bounds are similarly collected. More formally, if non-monotone interval $\hat{x}_i = [min_i, max_i]$, then let $\hat{x}_{i,1} = [min_i, mid_i]$ and $\hat{x}_{i,2} = [mid_i, max_i]$ where $mid_i = (min_i + max_i)/2$. Also let $\hat{L}$ represent the set of subintervals $\hat{x}_{i,j}$ that are likely to produce lower bounds and $\hat{U}$ represent the set of subintervals that are likely to produce upper bounds. The goal is to produce a partitioning $(\hat{L}, \hat{U})$ of the problem space. The difficulty here is that F is non-monotone with respect to variable x_i and we do not know which values of x_i minimize or maximize F. Here again, we use the partial derivatives to make an educated guess. Specifically, the magnitude of the derivative indicates the slope of the curve, while its sign denotes

$\hat{x}_1$	$\hat{x}_2$	$\hat{x}_3$	$\partial\mathcal{F}/\partial x_1$	$\partial\mathcal{F}/\partial x_2$	$\partial\mathcal{F}/\partial x_3$	Action	$\mathcal{F}_I$
0.5, 1.5	1.0,2.0	4.0,5.0	-7.0,0.6	-4.3,1.0	0.5,1.0	bisect	
0.5, 1.0	1.0,1.5	4.0,5.0	-5.0,0.0	-4.0,0.2	0.7,1.0	bisect	
1.0, 1.5	1.5,2.0	4.0,5.0	-1.0,0.3	-1.6,0.0	0.5,0.7	bisect	
0.5, 1.0	1.0,1.25	4.0,5.0	-4.0,0.0	-4.0,-0.6	0.8,1.0	compute	5.45,7.50
0.5, 1.0	1.25,1.5	4.0,5.0	-5.0,-0.3	-2.2,0.2	0.7,0.8	bisect	
1.0, 1.25	1.5,2.0	4.0,5.0	-1.0,0.1	-1.4,0.0	0.5,0.7	bisect	
1.25, 1.5	1.5,2.0	4.0,5.0	-0.3,0.3	-1.6,-0.2	0.5,0.7	bisect	
0.5, 1.0	1.25,1.38	4.0,5.0	-4.5,-0.3	-2.2,-0.1	0.7,0.8	compute	5.28,7.00
0.5, 1.0	1.38,1.5	4.0,5.0	-5.0,-0.4	-1.6,0.2	0.7,0.7	estimate	4.54,7.64
1.0, 1.12	1.5,2.0	4.0,5.0	-1.0,-0.2	-1.3,0.0	0.5,0.7	compute	4.90,5.83
1.12, 1.25	1.5,2.0	4.0,5.0	-0.6,0.1	-1.4,-0.1	0.5,0.7	estimate	4.32,6.36
1.25, 1.38	1.5,2.0	4.0,5.0	-0.3,0.2	-1.5,-0.2	0.5,0.7	estimate	4.34,6.31
1.38, 1.5	1.5,2.0	4.0,5.0	-0.1,0.3	-1.6,-0.3	0.5,0.7	estimate	4.38,6.29
TOTAL							**4.32,7.64**

Table 4.3: A Multi-Variable Interval Differential Bounds Computation

the direction (whether the function is increasing or decreasing). For non-monotone x_i, its upper and lower derivative bounds specify the maximum increasing and decreasing slopes respectively. A large upper bound implies that the function rapidly increases over some values of $\hat{x}_i$. Likewise, a large lower bound implies that the function rapidly decreases over some values of $\hat{x}_i$. Therefore, if the magnitude of the upper bound is greater than that of the lower, we know that the function increases faster than it decreases and assign $\hat{x}_{i,1}$ to $\hat{L}$ and $\hat{x}_{i,2}$ to $\hat{U}$. If the magnitude of the lower bound is the greater, then we assume that the function decreases faster than it increases and assign $\hat{x}_{i,2}$ to $\hat{L}$ and $\hat{x}_{i,1}$ to $\hat{L}$. In practice, this partitioning method works very well, efficiently yielding tight bounds that approximate the theoretical solution.

To illustrate this algorithm, consider $F(x_1, x_2, x_3) = x_1 + x_2/x_1 + x_3/x_2$ where $\hat{x}_1 = [0.5, 1.5]$, $\hat{x}_2 = [1, 2]$, $\hat{x}_3 = [4, 5]$, and the bisection limit is 3. The results of each stage of the computation are displayed in Table 4.3. As the table shows, the interval differential bounds $\mathcal{F}_I$ for this function are $[4.32, 7.64]$. The theoretical bounds $\hat{F}$ are $[4.83, 7.50]$ and the interval bounds $\mathcal{F}$ are $[3.17, 10.50]$. Clearly, the interval differential approach produced a significantly better approximation to $\hat{F}$ than $\mathcal{F}$ did.

Because we partition and group non-monotonic variables, the outer loop of our interval differential algorithm (see Figure 4.2) is executed a maximum of $2^{b+1} - 1$ times, where b represents the maximum number of bisections allowed. Therefore, the worst-case computational complexity of this algorithm is $O(2^{b+1}sn)$, where b is the

bisection limit, s is the number of arithmetic operations in F, and n is the number of independent variables. Fortunately, the algorithm achieves fairly good results with small values of b. In this paper, all interval differential bounds were generated with $b = 3$.

4.3 Improved Delay Estimates for Timing Analysis

Having described two general interval techniques for approximating the United Extension, we now apply these methods to the timing analysis problem. As in the previous chapter, we have chosen Crystal's PR-slope model [Oust83, Oust84, Oust86].

4.3.1 The Centred Version

First, we produce a Krawczyk's centred-form version of the PR-slope delay modeler. For this problem, the function $D(x_1, x_2, ..., x_n)$ is the delay function given in Equation 3.1 and the variables $x_1, x_2, ..., x_n$ are the resistance factors, capacitance factors, and transistor sizes needed to compute the delay. The transistor resistance and capacitance factors are predetermined by table lookup.[1]

Computing the centred form of this delay function is accomplished in two phases. First, we calculate the interval slope $\hat{g}_i$ for each variable x_i, then we use these interval slopes to find bounds on the delay. To calculate the interval slope, we require a straight-line program for computing D. We therefore encode the delay equation within a computer program and employ it to compute the interval slope vector G. Once we have determined G, we use Equation 4.2 to calculate the final delay bounds.

4.3.2 The Interval Differential Version

To compute interval differential delay bounds, we must first create an interval differential version of the delay function D. This is accomplished by replacing each arithmetic operator $+$, $-$, $\cdot$, and $/$ in D by its interval differential equivalent specified in Equations 4.16 through 4.19 and each variable x_i in D by an interval operand $\hat{X}_i$ indicating its value and derivative. When evaluated, this new function returns the interval partial

[1]Because Crystal uses table lookup to determine the trigger transistor resistance factor, any dependency between this value and the other variables is hidden from the algorithm. Both the centred and interval differential methods model variable interdependence if all dependent values are explicitly expressed as functions of the independent variables. Because this is not the case in Crystal, we chose to precompute this factor and treat it as an independent variable for the delay computation.

derivative $\partial \mathcal{F}/\partial x_i$ of each x_i. Then we follow the algorithm given in Figure 4.2 to compute delay bounds.

4.4 Performance

We implemented both centred-form (Cencryst) and interval differential (Idcryst) versions of Crystal's PR-Slope model. To test these methods, we compared them against the original interval algorithm (Intcryst), a Monte Carlo version (Crystal with 10,000 trials), and the nominal delay (Crystal). To select variable values for each Monte Carlo trial, we employed a random number generator with a uniform distribution. In Idcryst, the maximum number of bisections was set to three. All implementations were written in the C programming language and run on a SUN SPARCstation. In this section, we present an analysis of the performance of each algorithm.

4.4.1 Quality of the Bounds

To determine the quality of the bounds, we conducted a number of experiments comparing the generated bounds to the theoretical bounds ($\hat{D}$) for a given circuit. As before, the theoretical bounds were approximated by Monte Carlo simulation with an unlimited number of trials. For all methods, the trigger transistor resistance was precomputed and treated as an independent variable in the delay function. Figures 4.3 and 4.4 illustrate the relationship between delay and stage length (i.e., the number of transistors in the stage) for a family of nMOS stages. These graphs display the percentage error for both the upper and lower bounds generated by each method. As these graphs demonstrate, the Monte Carlo and nominal value techniques always underestimate the bounds, while the interval approaches always overestimate them. Of all the methods, Idcryst produces the best bounds, coming within about 4% of the desired result. Note that the centred method generates a skewed interval, producing a tight upper bound and a poor lower bound. Since centred-forms yield an interval centered around the nominal value, we conclude that the nominal delay does not fall in the middle of the true bounds, thus causing the skewed centred values that we observe. Intcryst produces bounds within 8% of the theoretical values and the Monte Carlo technique comes within 12%. The nominal delay produces the poorest estimate of best- and worst-case delay. For most of these methods, the error grows with the

Stage	Size	Theoretical		Cencryst		Idcryst		Intcryst	
		min	max	min	max	min	max	min	max
1	2	6.8	9.3	6.7	9.3	6.8	9.3	6.8	9.3
2	3	54.3	72.9	53.1	72.9	54.3	72.9	54.1	73.1
3	2	10.3	14.1	10.1	14.1	10.3	14.1	10.3	14.1
4	2	10.6	14.4	10.4	14.4	10.6	14.4	10.6	14.4
5	2	1.2	1.6	1.1	1.6	1.2	1.6	1.2	1.6
6	1	0.7	1.0	0.7	1.0	0.7	1.0	0.7	1.0
7	1	0.8	1.1	0.7	1.1	0.8	1.1	0.8	1.1
8	3	20.4	27.4	19.9	27.4	20.4	27.4	20.3	27.5
9	1	2.0	2.7	1.9	2.7	2.0	2.7	2.0	2.7
Path total		107.1	144.5	104.6	144.5	107.1	144.5	106.8	144.8

Table 4.4: A B-SYS Critical Path Delay Computation (ns)

length of the stage. Fortunately, the stages encountered in most circuits are typically small, containing fewer than five transistors.

In addition to the above experiments, we also analyzed a portion of the Brown Systolic Array (B-SYS)[Hugh89]. We extracted a 2,000 transistor piece of the circuit and computed its critical path. This path consists of nine stages, each ranging in length from one to three transistors. Table 4.4 displays the delay bounds generated by each interval method for the entire critical path and for each individual stage in the path. Table 4.5 gives the percentage error in the bounds for each method, including the Monte Carlo and Nominal results displayed in the previous chapter. As the tables shows, the interval differential version produces the tightest delay bounds, achieving the theoretical bounds for all stages. Because the function is monotone over the ranges of all variables, this interval technique can precisely calculate the bounds. As in the previous experiments, Cencryst produces very tight upper bounds (matching the theoretical ones), but poorer lower bounds (only coming within 2.3% of the desired result). On average, Intcryst comes within 0.3% of both upper and lower bounds. The Monte Carlo method produces an average error of about 4.3%, while the nominal delay has an average error of about 15%.

4.4.2 Running Time of the Algorithm

The graph in Figure 4.5 depicts the relative speed of each of the implementations. As expected, the Monte Carlo method is the least efficient of the bounding techniques. Of

Stage	Size	Cencryst		Idcryst		Intcryst		Monte Carlo		Nominal	
		min	max	min	max	min	max	min	max	min	max
1	2	2.3	-0.1	0	0	0.3	-0.3	-5.9	5.9	-16.7	14.0
2	3	2.3	0	0	0	0.3	-0.3	-4.2	4.7	-16.0	13.6
3	2	2.4	-0.1	0	0	0.1	-0.1	-4.8	5.3	-17.0	14.3
4	2	2.4	0	0	0	0	-0.1	-3.8	3.7	-16.8	14.1
5	2	3.5	0	0	0	0.9	-0.6	-5.2	5.7	-18.1	15.4
6	1	2.9	0	0	0	0	0	-7.1	5.1	-18.6	15.3
7	1	3.9	0	0	0	0	0	-3.9	4.6	-19.5	15.6
8	3	2.3	-0.1	0	0	0.4	-0.4	-4.1	3.1	-16.0	13.6
9	1	2.0	0	0	0	0	0	-1.5	1.5	-16.2	13.6
Path total		2.3	0	0	0	0.2	-0.3	-4.3	4.4	-16.3	13.8

Table 4.5: % Error in Delay Bounds for the Critical Path

the interval-based techniques, the original interval algorithm is the most efficient. As stated in the previous chapter, this method runs roughly four times slower than the non-interval PR-slope model in Crystal and has a time complexity of $O(n)$, where n is the number of independent variables.

The centred-form method is the next fastest interval method. As previously stated, its running time is $0(sn)$, where s is the number of instructions in the straight-line program for computing the given function. For Crystal's delay equation, s is proportional to the number of variables in Equation 3.1; therefore, the running time of Cencryst is $O(n^2)$.

The interval differential method is the slowest of the interval algorithms. Since it computes derivatives for each of the n variables, this method requires $O(n^2)$ time for each function evaluation. If the function is monotone over the range of all variables, then the algorithm evaluates this function only once. If, however, the function is non-monotone, then we must bisect the intervals and evaluate the function over the subintervals. In the worst-case, the algorithm may evaluate the function $2^{b+1}-1$ times, where b represents the maximum number of interval bisections allowed. Therefore, this algorithm has a worst-case running time of $O(2^{b+1}n^2)$.

4.5 Summary

In this chapter, we presented two interval-based methods that improve on our original technique for modeling uncertainty in RC timing analysis. The first of these uses

Krawczyk's centred form, while the second employs an extension of Rall's interval differential calculus.

A comparison of the three interval-based approaches reveals a classic trade-off between efficiency and precision. Our experiments demonstrated that both the centred (Cencryst) and interval differential (Idcryst) algorithms produce tighter bounds than the original interval method (Intcryst). The primary reason for this is that the new methods take into account variable interdependence, while the original does not. The original, however, requires less time to compute its bounds.

The centred method is the second fastest interval approach, but generates bounds of inconsistent quality. In all of our experiments, this algorithm yielded a tight upper bound and a poor lower bound. These skewed bounds may be attributed to the fact that the nominal delay did not fall in the center of the true interval. Since the resulting centred interval is centered at this nominal value, Cencryst must overestimate the lower bound in order to encompass the theoretical upper bound.

In general, the interval differential approach is the slowest of the algorithms, but generates the tightest bounds. In our experiments, this algorithm produced bounds within about 4% of the true results using at most three bisections. It is also the only one of the three methods that provides a measure of the goodness of its estimate; the number of estimated subinterval bounds indicates the precision of the result.

All three interval-based approaches are much more efficient than Monte Carlo simulation. In experiments with 10,000 Monte Carlo trials, all three also generated significantly tighter bounds.

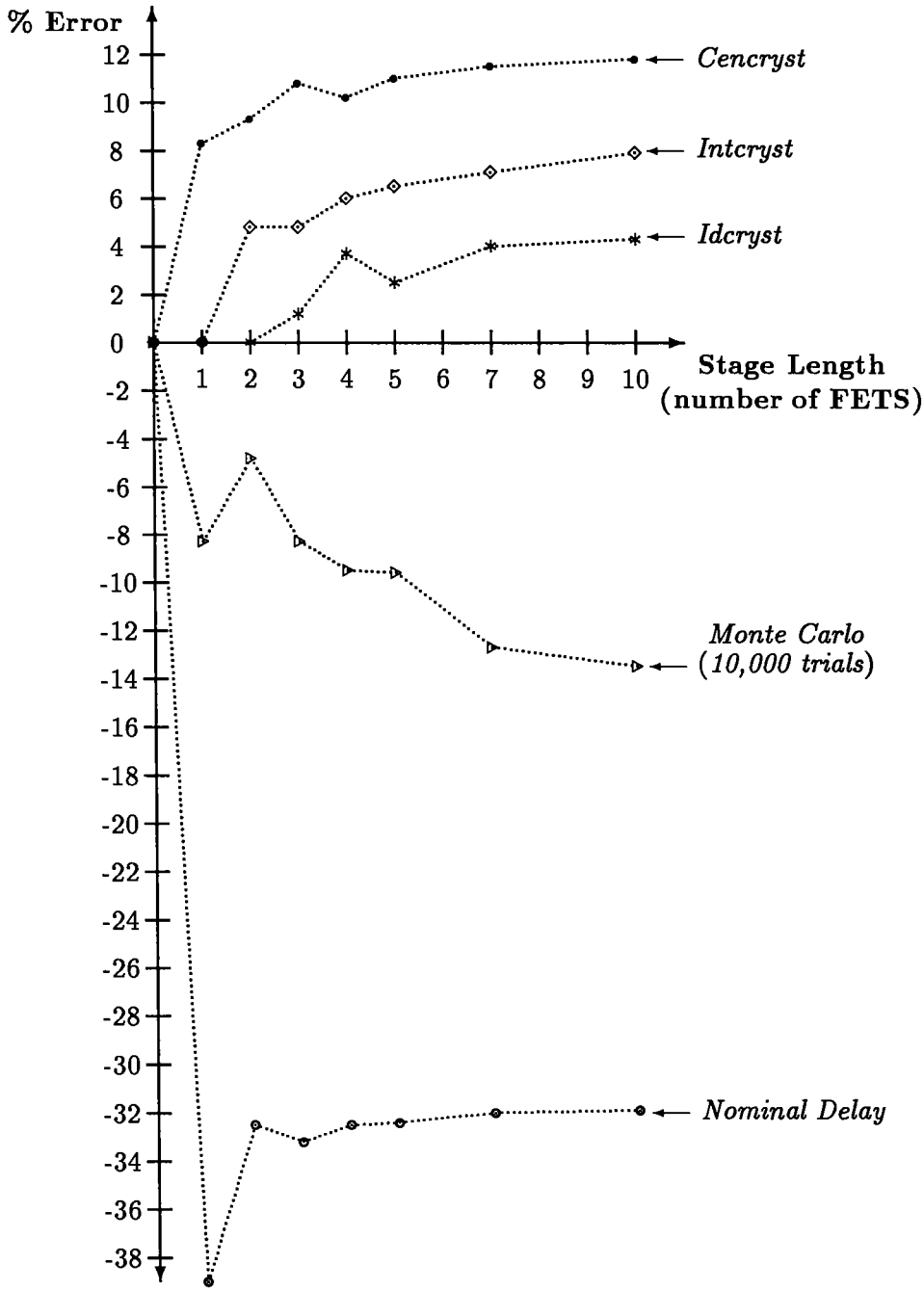


Figure 4.3: %Error vs. Stage Length (Lower Bounds)

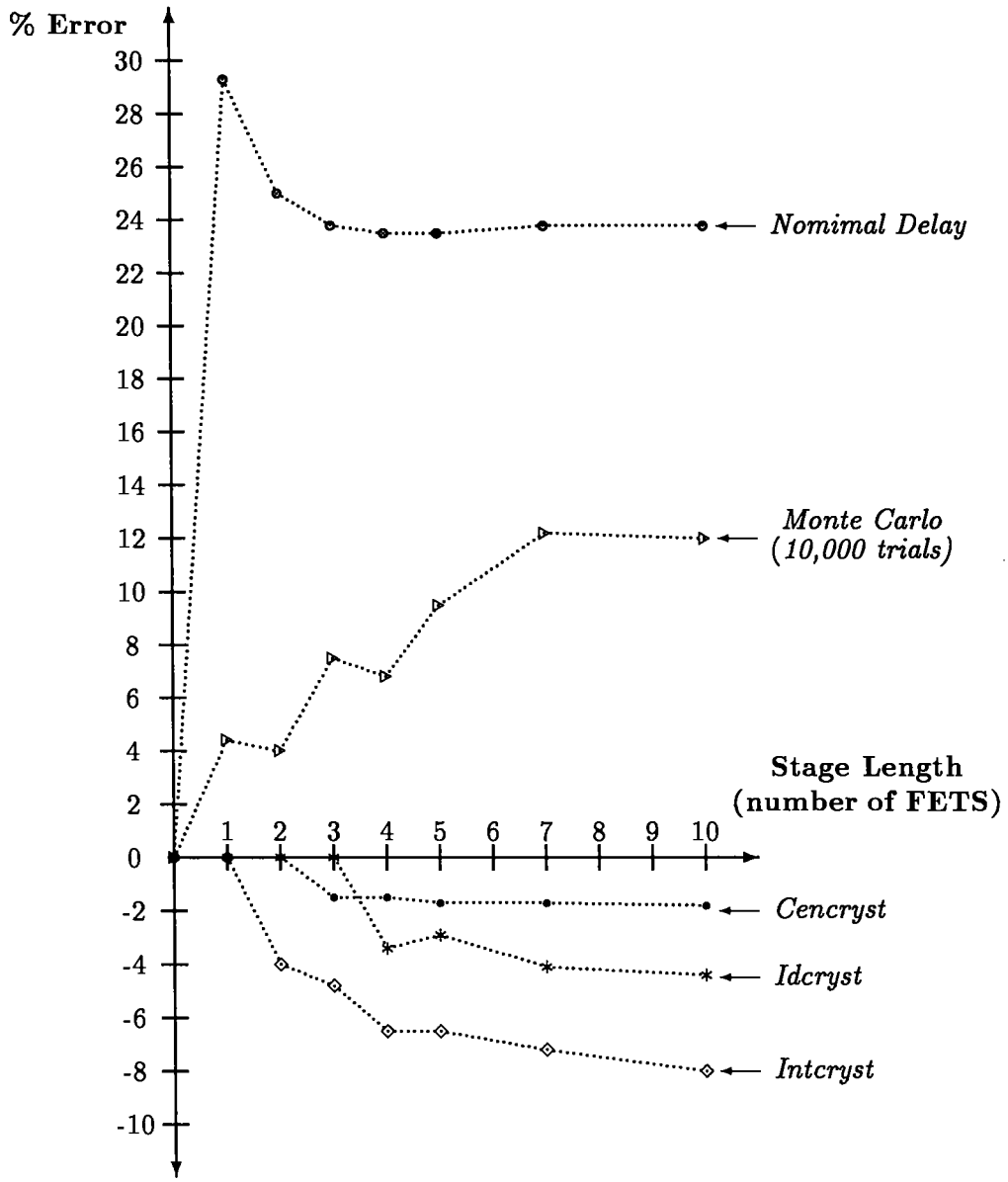


Figure 4.4: %Error vs. Stage Length (Upper Bounds)

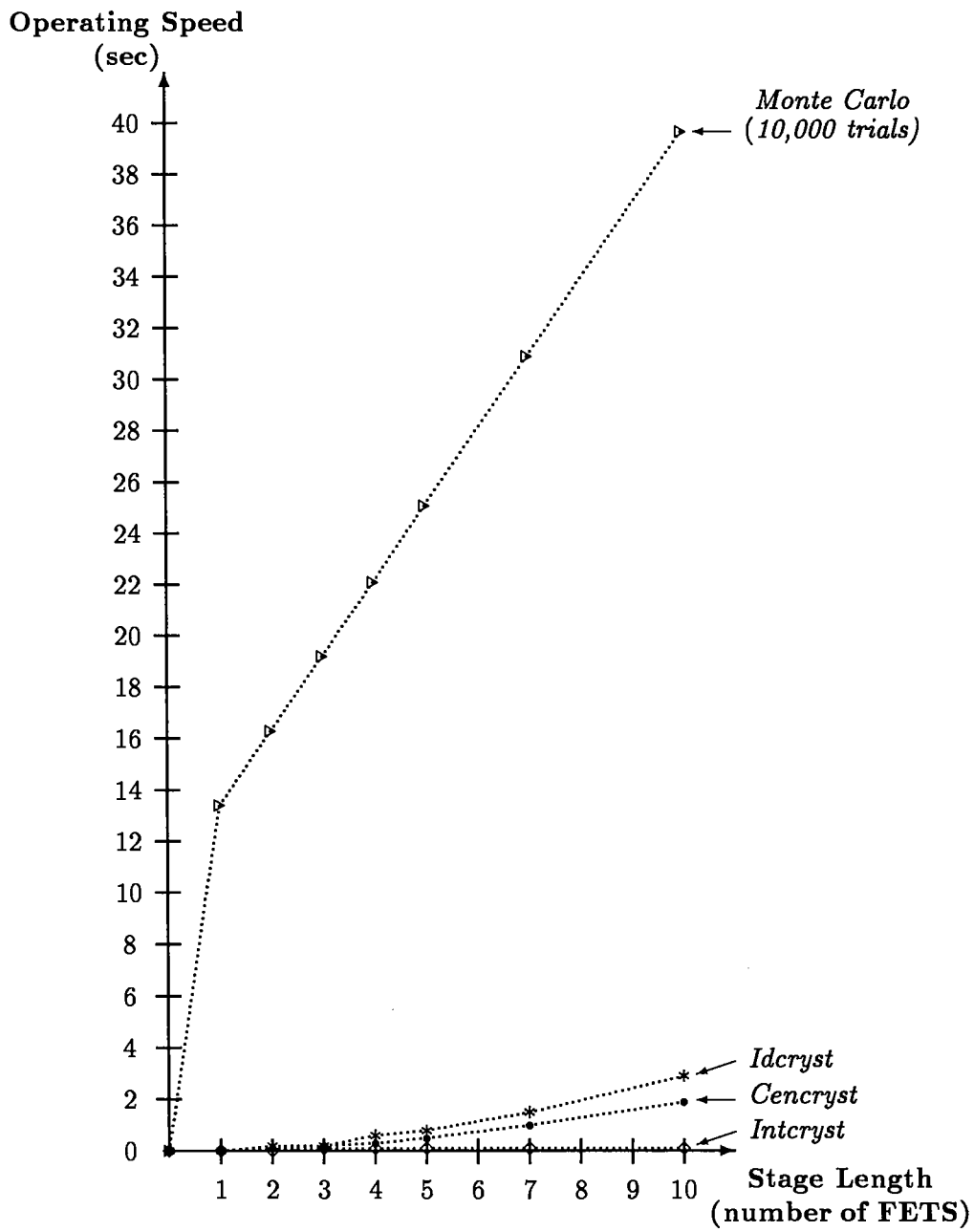


Figure 4.5: Operating Speed vs. Stage Length

Chapter 5

Uncertainty and the Longest Path Problem

In previous chapters, we discussed the problem of uncertainty in timing analysis and presented interval methods for computing bounds on stage delay. In this chapter, we address the second half of the problem – how to combine these uncertain stage delays to compute bounds on the delay of the entire circuit. To solve this problem, we employ a longest path algorithm.

The *classical* longest/shortest path problem[Tarj83, pp. 85-96] can be summarized as follows: Let $G = (V, E)$ be a directed graph with weighted edges, such that $length(v, w) \in \Re$ for all edges $(v, w) \in E$. Also designate $s \in V$ to be the source vertex. A path p is defined as a sequence of consecutive graph edges originating at the source vertex s. The length of a path p, denoted $length(p)$, is defined as the sum of the lengths of its edges. The longest (shortest) path from vertex s to any other vertex $v \in V$ reachable from s is the path from s to v whose length is maximum (minimum). If the graph contains a positive (negative) cycle, then no longest (shortest) path exists to vertices reachable from the cycle. For a given vertex $v \in V$, the standard algorithm returns the length $a \in \Re$ of the longest (shortest) path to vertex v and one representative path p from s to v such that $length(p) = a$.

For timing analysis, graph G depicts the timing dependencies between nodes in the circuit. In Crystal[Oust83, Oust84, Oust86], the vertices represent the input, output, and target nodes in the circuit. The edges of the graph represent stages activated by signal changes at these nodes. Each graph edge is labeled with the delay for that

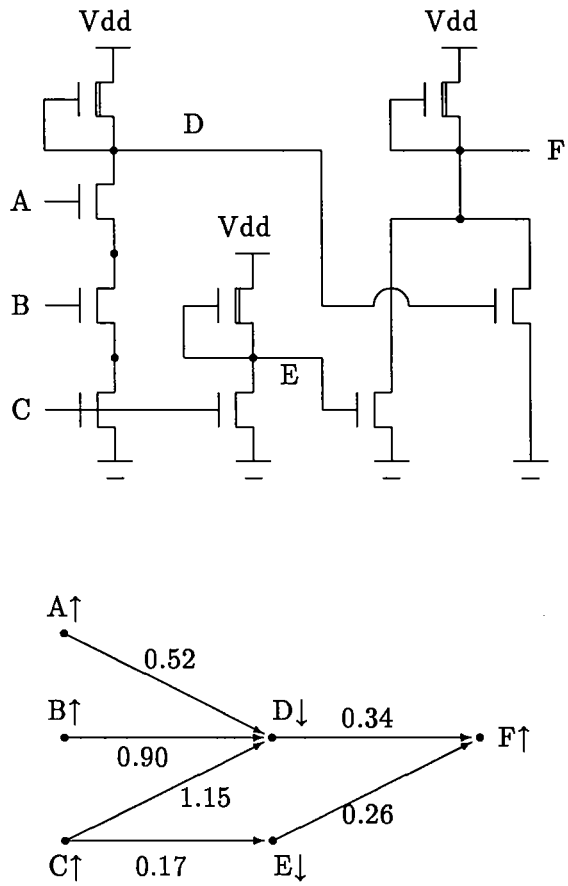


Figure 5.1: Timing Dependency Graph for a Transistor Network

stage computed by RC analysis. Figure 5.1 presents a nMOS transistor network and one possible timing dependency graph for this circuit. The vertices in the graph correspond to the labeled nodes in the circuit and the edges represent stages. For instance, a rising signal at node A could allow current to flow from GND to node D, thus producing a falling signal value at node D. When analyzed using Crystal's PR-Slope model, this stage has a delay of 0.52 ns. To represent the stage, the graph contains a directed edge from A to D with a weight of 0.52 ns. All the delays shown in this graph are computed using Crystal's PR-Slope model.

For timing verification at the logic level, similar dependency graphs are created [Wall86]. Here, the graph vertices represent the inputs and outputs of logic blocks and

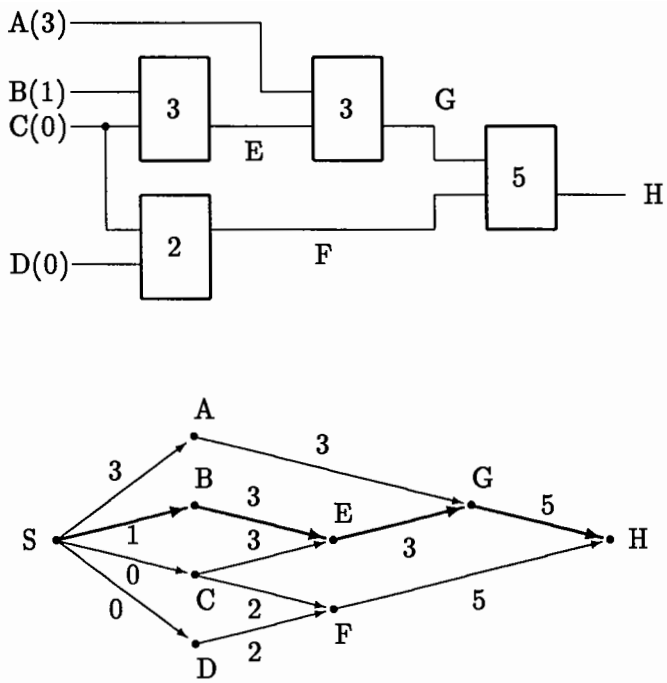


Figure 5.2: Timing Dependency Graph for a Logic Circuit

the edges represent the timing dependencies between them. Associated with each edge is the delay through the logic block for that input/output pair. For example, Figure 5.2 presents a simple logic circuit. Each circuit input is labeled with its *input arrival time* in parentheses and each logic block is labeled with its associated delay (ns). Below the circuit is its associated timing dependency graph. As before, the vertices in the graph correspond to the labeled nodes in the circuit and the edges are labeled with the delay through the circuit between the nodes. Note that we added a source vertex S to the graph and edges from this source vertex to each input node. We labeled each of these edges with the appropriate arrival time for each input.

Once created, these timing dependency graphs are used to estimate the operating speed of the circuit and to detect potential timing errors. The longest path through the graph indicates the path through the circuit with maximum delay. For combinational circuits, this is used to determine minimum operating speed. For instance, the circuit in Figure 5.2 has a maximum delay of 12ns as determined by the path through vertices

SBEGH. For sequential circuits, the maximum delay between clocked components is used to determine the clock rate. Similarly, the shortest path through the graph indicates minimal delay and is used to locate race conditions. Race conditions occur when the delays along different paths through the circuit cause conflicting signals to arrive at a node.

Placement provides another instance of the longest path problem. Here, the graph (called a *channel position graph*) indicates the relative position of the cells in the layout [Prea79]. Each vertex in the graph corresponds to a boundary between a cell block and a routing channel. Each edge represents one of these channels or logic cells and is labeled with its physical dimensions. Figure 5.3 depicts a sample layout and a vertical channel position graph for this layout[1]. Although routing is not shown in the placement diagram, the channels are sized to allow the necessary connections between blocks. The graph presents the relative positions of the blocks. Each edge in the graph is labeled with a block or channel identifier followed by its width (measured in mils). Block identifiers are denoted by the letter B and the corresponding number of the block. Channels are denoted by the letter C and are numbered from left to right as they are encountered in the layout. The longest path through a channel position graph defines the minimum length or width of the final layout. For instance, the minimum length of the layout in Figure 5.3 is 92 mils as determined by the path through blocks B1 and B8. Shortening the longest path length is tantamount to achieving a more compact design.

As these examples illustrate, solving the longest/shortest path problem is an important part of many VLSI design tools. For many of these applications, the weights on the graph edges are not exactly known. In timing analysis, the delays of circuit components cannot be precisely predicted prior to fabrication. The channel and cell dimensions needed for automatic placement cannot always be precisely determined. In hierarchical placement systems, the sizes of the circuit components at higher levels in the hierarchy are determined from cell layouts at lower levels. If these lower-level cells are not completely placed and routed, the dimensions of the higher-level components containing them are not known exactly, but it is possible to determine ranges for these values. Also full routing is required to accurately calculate channel width. Because of its computational expense, it is not practical to fully route each trial placement when

[1]This layout was adapted from a circuit in [Prea79].

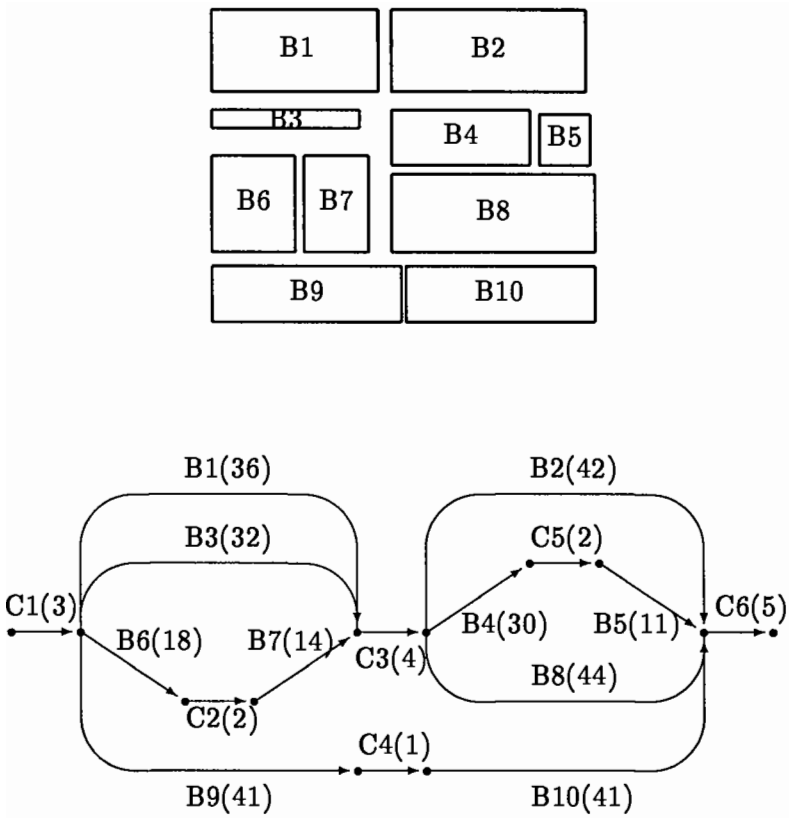


Figure 5.3: Vertical Channel Position Graph for Placement

exploring layout alternatives. Instead, channel width must be estimated until an acceptable placement is found. In these cases, we require a longest path algorithm for graphs with uncertain edge weights.

To model uncertainty, some timing verifiers [McWi80, Wall86] represent delay by the upper and lower bounds on its value. Some others [Hitc82, Wall86] represent uncertain delay with statistical distributions, employing a mean value and standard deviation for each delay. While these timing verifiers provide rules for combining uncertain delays to compute bounds on the delay of the circuit, the algebraic structure behind these rules is only intuitively defined. One of our goals is to formalize this underlying algebraic structure. Second, none of these systems returns the paths that produce the delay bounds, but these paths are needed to improve the speed of the chip. Our second goal is to return path as well as delay information.

The classical longest path algorithm returns one representative longest path to a given vertex. For many VLSI applications, one representative path is not sufficient. In timing analysis, it is not uncommon for several paths to have similar or identical delays. If the timing algorithm returns only one sample path, the designer may waste time reducing the delay on this path, only to discover a second path as bad as the first. It is therefore desirable to return the set of all paths whose delay values are within a certain tolerance of the worst-delay path. Since these paths may contain common edges, the designer can increase the speed of the entire design by reducing the delay through these common pieces. In our longest path algorithm, we return a *set* of longest paths, each of which could potentially be longest.

In this chapter, we present a theoretical framework based on interval algebra for solving the longest/shortest path problem with uncertain edge lengths. More formally, let I be the set of intervals over $\Re$ and let $G = (V, E)$ be a directed graph with interval-weighted edges such that $length(v, w) \in I$ for all edges $(v, w) \in E$. Also let $s \in V$ be designated as the source vertex. For any vertex $v \in V$, we return bounds $\hat{a} \in I$ on the length of the longest (shortest) path to vertex v and the set of all paths $\{p_1, p_2, ..., p_k\}$ from s to v in G such that $length(p_i) \cap \hat{a} \neq \emptyset$ for each path p_i. In other words, we return the set of all paths that have maximum (minimum) length under some possible assignment of values to the edge lengths. To solve this problem, we develop an interval algebra for manipulating uncertain edge lengths and then modify the standard longest/shortest path algorithm [Tarj83] to incorporate this framework. We conclude this chapter by analyzing the performance of the resulting interval path algorithms.

5.1 Classical Solutions to the Longest Path Problem

We begin with a brief review of the standard longest path problem adapted from Tarjan [Tarj83, pp. 85-96]. Given a directed graph with weighted edges, we wish to find paths through the graph with maximum length. There are four variations on this basic problem: single pair, single source, single sink, and all pairs. In the first of these, we seek the longest path between a given pair of vertices. In the single source problem, we are given only the source vertex s and must determine the longest path from s to v for every vertex v. The single sink problem is analogous: given a destination vertex t, find the longest path from v to t for every vertex v. The last version is the all pairs problem. As its name suggests, we seek to locate a longest path from s to t for every pair of vertices s and t. Since the single source problem can be adapted to solve all the other variations [Tarj83, p. 85], we will only discuss solutions to it.

In [Tarj83], Tarjan presents a number of solutions to the single source longest path problem. Although several versions of this algorithm are given, they all share the same basic structure and differ only in the search strategy employed. We therefore present the basic algorithm here and then discuss the different search options. Formally stated, we are given a directed graph $G = (V, E)$ with weighted edges $(v, w) \in E$ such that $length(v, w) \in \Re$ and a source vertex $s \in V$. For each vertex $v \in V$, the algorithm returns the length $a \in \Re$ of the longest path from s to v and a representative path p from s to v such that $length(p) = a$. The longest paths are returned as a *longest path tree* (i.e., a spanning tree rooted at source vertex s each of whose edges is part of a longest path to some vertex v). To compute longest paths, the algorithm partitions the vertices into three groups or states: *unlabeled*, *labeled*, and *scanned*. The unlabeled vertices are those that have not yet been examined, the labeled vertices are those that we wish to examine, and the scanned vertices are those that have already been examined. At the start of processing, the source vertex is marked labeled and all other vertices are marked unlabeled. In the course of the algorithm, labeled vertices are chosen, processed, and marked scanned. When a vertex is analyzed, it will cause other vertices to become labeled. The algorithm finishes when all the vertices have been scanned. These state transitions are summarized in Figure 5.4. In addition to state, each vertex v has two other associated attributes: *dist(v)* and *p(v)*. The first of these attributes is the current maximum path length from source s to vertex v and the second is its current parent in the longest path tree. These attributes are updated as we examine labeled vertices in

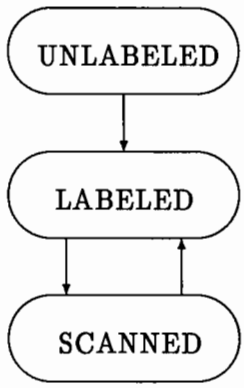


Figure 5.4: State Transitions for Graph Vertices

the graph. When processing concludes, the parent attributes $p(v)$ denote the longest path tree. It is important to note that if the graph contains a positive cycle, no longest path exists. In this case, the algorithm will never halt; therefore, we must check for positive cycles. The algorithm is summarized in Figure 5.5.

As previously mentioned, there are several versions of the above algorithm. Each one differs from the others only in the strategy used to select vertices from the labeled set. The aim of these methods is to increase the efficiency of the algorithm by choosing an efficient scanning order. Some common scanning orders for this problem are topological, breadth-first, and depth-first. The topological approach orders the vertices such that if (v, w) is a directed edge in the graph, v appears before w in the ordering. This approach works best for acyclic graphs and has a running time proportional to m, the number of edges [Tarj83, p. 89]. The breadth-first scanning strategy chooses the least recently labeled vertex. The running time of this algorithm is $O(nm)$ [Tarj83, p. 93]. The third scanning order is depth-first (i.e., we choose the most recently labeled vertex). The worst-case time complexity of this strategy is exponential in the number of vertices [Wall86, p. 684]. For example, suppose that vertex x is on the path from source vertex s to target vertex t. Also suppose that there exists a second path from s to t that also passes through vertex x such that the path length from s to x along this second path is greater than the path length from s to x along the first. Using a depth-first scanning order, we follow the first path marking the vertices scanned as we go. Later in the search, we follow the second path until we reach vertex x. Because this new

```
Initialize the vertices:
  Mark vertex s labeled
  Mark all other vertices v unlabeled
  dist(s) = 0
  For all other vertices v, dist(v) = -∞
  For all vertices v, p(v) = ∅
Until there are no more labeled vertices or we detect a positive cycle
  Select a labeled vertex v
  Mark vertex v scanned
  For each outgoing edge (v, w) for this vertex
    new dist = max(dist(v) + length(v, w), dist(w))
    If new dist ≠ dist(w)
      Update vertex w:
        Mark vertex w labeled
        dist(w) = new dist
        p(w) = v
    End if
  End for
End until
```

Figure 5.5: The Classical Longest Path Algorithm (Tarjan)

path has greater length, the maximum distance from s to x has changed. This change at vertex x will affect all remaining vertices on paths from s to t containing x; thus, vertex x and all its "descendants" must now be re-examined. In the worst-case, we could end up enumerating all paths through the graph; therefore, this algorithm has an exponential worst-case time complexity. Fortunately, the worst-case is rarely encountered in practice. Many timing verifiers, including Crystal[Oust83, Oust84, Oust86], employ this strategy for critical path analysis. Crystal's PR-Slope model uses the rise time from the previous stage to compute the delay of the next stage. Because the depth-first method traces the paths through the graph, it is a natural choice for this type of analysis. Table 5.1 illustrates each of these scanning orders for the graph in Figure 5.2.

5.2 Longest Paths with Uncertain Edge Length

Now we are given a directed graph $G = (V, E)$ with interval weighted edges such that $length(v, w) \in I$ for all edges $(v, w) \in E$ and a source vertex $s \in V$. For each vertex $v \in V$, the interval algorithm must return bounds $\hat{a} \in I$ on the length of the longest

Method	Scanning Order
Topological	SABCDEFGH
Breadth-First	SABCDGEFHGH
Depth-First	SAGHBEGHCFD

Table 5.1: Survey of Scanning Strategies

path from s to v and the set of all paths $\{p_1, p_2, ..., p_k\}$ from s to v such that for each path p_i, $length(p_i) \cap \hat{a} \neq \emptyset$. In effect, the resulting longest "path" to vertex v is actually a set of paths, each of which has maximal length under some legal assignment of values to the graph edges. To illustrate this, consider the directed graph with uncertain edge lengths shown in Figure 5.6. For this example, assume that the intervals are defined over the integers. By enumerating all possible edge length assignments, we can compute the set of longest paths from vertex A to I. Table 5.2 displays the longest path from A to I for each possible combination of edge values. The final totals are calculated by taking the union of the results from each trial. As shown, there are two possible longest paths from A to I (ABEHI and ABECFHI) and the bounds on longest path length are $[21, 23]$. Because we must execute the longest path algorithm for each possible combination of value assignments, this method has time complexity that is exponential in the number of unknown edge lengths even when using a constant time longest path algorithm. A quick method for computing bounds would be to run the standard longest path algorithm twice – once with all minimum edge weights and again with maximum values. While this does yield bounds, it does not return all the paths. As indicated by the first and last entries in the enumeration table, this method would return bounds of $[21, 23]$ for vertex I, but only one longest path (ABECFHI). To locate all longest paths, we present the following interval-based longest path algorithm.

5.2.1 An Interval Longest Path Algorithm

To create an interval longest path algorithm, we must modify the original algorithm in two areas. First, we must create and incorporate an interval algebra for manipulating uncertain edge lengths. Second, we must store multiple longest paths.

To address the first problem, we define two interval operations $\oplus$ (addition) and

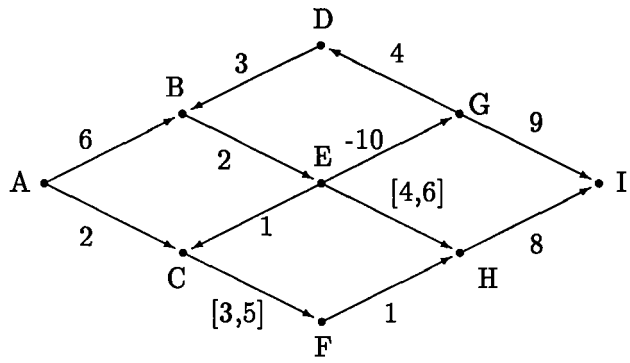


Figure 5.6: Directed Graph with Uncertain Edge Lengths

Edge		Longest Path A → I	
(C,F)	(E,H)	Length	Path
3	4	21	ABECFHI
3	5	21	ABEHI or ABECFHI
3	6	22	ABEHI
4	4	22	ABECFHI
4	5	22	ABECFHI
4	6	22	ABEHI or ABECFHI
5	4	23	ABECFHI
5	5	23	ABECFHI
5	6	23	ABECFHI
Total		[21,23]	{ABEHI, ABECFHI}

Table 5.2: Computing Longest Paths by Exhaustive Enumeration

MAX (maximum) over the set I as follows:

$$[a,b] \oplus [c,d] \equiv [a+c, b+d] \tag{5.1}$$

$$MAX([a,b],[c,d]) \equiv [max(a,c), max(b,d)] \tag{5.2}$$

These interval operators are the logical interval extensions of the discrete operators $+$ and max respectively and are derived from them using the following standard formula: $F(\hat{x},\hat{y}) = \{f(x,y) \mid x \in \hat{x} \text{ and } y \in \hat{y}\}$ [Moor79]. As $+$ is used to compute path length in the original algorithm, $\oplus$ is used to compute bounds on path length in the interval algorithm. Similarly, MAX is used in place of max to compute bounds on longest path length. We also need the interval relational operator $\neq$ to compare interval path lengths. This operator is defined as follows:

$$[a,b] \neq [c,d] \Longleftrightarrow a \neq c \text{ or } b \neq d \tag{5.3}$$

Having derived these equivalent interval operators, we substitute them into the path analysis algorithm to handle unknown edge lengths.

The simple substitution suggested above generates a longest path algorithm which returns bounds on the longest path length, but we also wish to return the set of paths that generate these bounds. To accomplish this task, we must save some additional information. In the original algorithm, the longest paths are saved in the parent attributes of the vertices. If the longest path from s to w contains edge (v,w) then $p(w) = v$. Because we need to store multiple paths, $p(w)$ now contains a set of possible parents for vertex w. We also associate a key (denoted $key(v)$) with each parent in the set, denoting the current maximum path length to vertex w along the path containing this parent. More formally, let path p be a path from s to w containing edge (v,w) such that $length(p) \cap dist(w) \neq \emptyset$; therefore, path p is a potential longest path to vertex w. To store this path, we save parent vertex v with the upper bound of $length(p)$ as its key. Whenever $dist(w)$ is updated, the parent set $p(w)$ must be purged of all parents whose keys are no longer elements of $dist(w)$. $Key(v) \notin dist(w)$ if and only if $length(p) < dist(w)$; therefore, path p through parent v cannot be a longest path to vertex w. When execution completes, these parent sets indicate the resulting set of potential longest paths for the graph.

Parent sets may be implemented in a variety of ways. The simplest implementation is a linked list of parents sorted in increasing order by key. With this method, each parent deletion requires $O(1)$ time, because the parent with the smallest key is always

```
Initialize the vertices:
  Mark source vertex s labeled
  Mark all other vertices v unlabeled
  dist(s) = [0, 0]
  For all other vertices v, dist(v) = [−∞, −∞]
  For all vertices v, p(v) = ∅
Until there are no more labeled vertices or we detect a positive cycle
  Select a labeled vertex v
  Mark vertex v scanned
  For each outgoing edge (v, w) for this vertex
    New path length = dist(v) ⊕ length(v, w)
    key(v) = upper bound of new path length
    New dist = MAX(new path length, dist(w))
    If new dist ≠ dist(w)
      Update vertex w:
        Mark vertex w labeled
        dist(w) = new dist
        Purge p(w) of all parents u such that key(u) ∉ dist(w)
        Add vertex v with key key(v) to p(w)
    Else if key(v) ∈ dist(w)
      Add vertex v with key key(v) to p(w)
    End if
  End for
End until
```

Figure 5.7: The Interval Longest Path Algorithm

found at the front of the list. Insertions, however, have a worst-case time of $O(n)$, since we may have to traverse the entire linked list. A more clever implementation for this set is a heap with the condition that the key of each node is less than that of its children. For this implementation, insertions cost $O(\log n)$, the depth of the heap. Since the parent with the smallest key is always on the top, deletions require constant time. Restoring the heap condition after a deletion, however, requires $O(\log n)$ time, where n represents the number of parents in the set. In the worst-case, n is the number of vertices in the graph. Fortunately, this situation rarely occurs, since in practice most parent sets contain relatively few entries.

As with the original algorithm, any of the scanning strategies given in the previous section may be employed to order the labeled vertices for evaluation. The changes made to incorporate intervals do not affect the scanning strategy at all. The interval algorithm is described in Figure 5.7.

5.2.2 Shortest Paths

The algorithm just presented can be easily altered to compute shortest paths. First, we require an interval minimum operator MIN to compute bounds on shortest path length. MIN is the interval extension of the discrete operator min and is defined as follows:

$$MIN([a,b],[c,d]) \equiv [min(a,c), min(b,d)] \tag{5.4}$$

As MAX and $\oplus$ are used to compute bounds on longest path length, MIN and $\oplus$ are similarly used to compute bounds on shortest path length. Second, we must change the key associated with each vertex in a parent set. Since we are computing shortest paths, we use the lower bound on path length as the key. Hence, if $key(v) \notin dist(w)$, then $length(p) > dist(w)$ and path p through parent v cannot be a shortest path for vertex w. No further changes are required.

5.2.3 Using the Results

The interval longest path algorithm returns the set of longest paths as the subgraph $G' = (V, E')$ of G such that $(v, w) \in E'$ if (v, w) is an edge on some potential longest path p in G. The parent sets indicate these edges; thus, if $v \in p(w)$ then edge $(v, w) \in E'$. To illustrate this, consider the directed graph in Figure 5.6 with source vertex A. The above algorithm computes the set of longest paths from A to each vertex and bounds on the longest path lengths. The resulting longest path graph G' is displayed in Figure 5.8. Each vertex in the graph is labeled with its maximum distance from source node A. Once created, G' can be used to extract and display useful information about the longest paths in G. Two possible applications are enumerating longest paths and finding common edges.

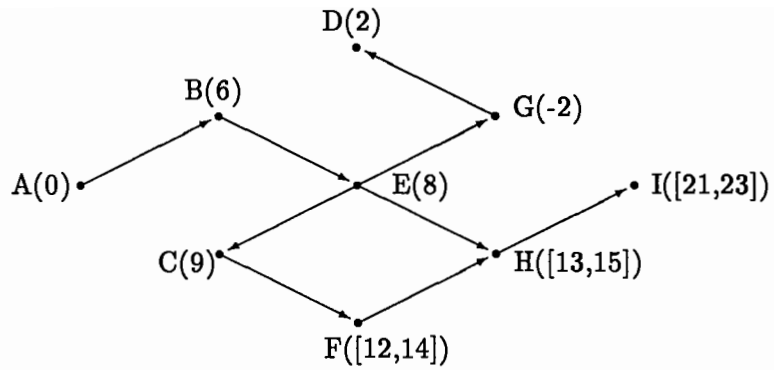
Timing analysis programs typically return the critical path through the circuit in addition to its delay. G' can be used to locate and return a list of potential longest paths from source vertex s to a given destination vertex t. To enumerate these paths, we start at the destination node t and trace the parent pointers back through the graph to source vertex s. Since this algorithm enumerates all possible paths from s to t in G', it has an exponential worst-case time complexity. In the worst-case, the number of paths is $O(k^n)$ where k is the maximum outdegree of a vertex in G' and n is the number of vertices. Because it assumes that each path p from s to t passes through all graph vertices and that each vertex has maximum outdegree, this worst scenario

rarely occurs. For example, the longest path graph in Figure 5.8 has nine vertices with a maximum outdegree of three and yet contains at most two longest paths to any given vertex. Beneath this graph is a detailed listing of the longest paths to vertex I. As shown, there are two possible longest paths from A to I (ABEHI and ABECFHI) and the bounds on longest path length are $[21, 23]$. Note that these results match those given in Table 5.2.

To increase the operating speed of a circuit, the chip designer must reduce the delay along the critical paths through the circuit. If all critical paths pass through the same area of the chip, the designer can reduce the delay on all critical paths by reducing the delay through this common piece. In our longest path algorithm, we therefore wish to locate the edges common to all longest paths between vertices s and t. Assume that we have a connected graph representing the longest paths between these vertices. If an edge is common to all longest paths, its removal will split the graph into two disjoint subgraphs (See Figure 5.9). We, thus, can modify an algorithm for finding articulation points to locate these common edges.[2] To solve this problem, we first construct an undirected graph $G''(V'', E'')$ from G'. This new graph differs from G' in two respects. First, it contains only those vertices and edges in G' that are on some longest path between vertices s and t. Second, we insert a special vertex along each edge of the resulting graph. For instance, if directed edge $(v, w) \in E'$ is on some longest path between s and t, then we create a new vertex u', add vertices v, w and u' to V'', and add undirected edges (v, u') and (u', w) to E''. The graph G'' can be constructed using depth-first search on G' in $O(m)$ time, where m is the number of edges in E'. To locate common edges, we now employ standard algorithms to locate the articulation points in graph G'' [Aho74, pp. 176-187]. A special vertex $u' \in V''$ is an articulation point of G'' if and only if its corresponding edge (v, w) in G' is a common edge. Finding articulation points requires $O(m')$ time, where m' is the number of edges in G''. Since m' is at most $2m$, we conclude that common edges can be located in time $O(m)$, where m is the number of edges in E'.

Figure 5.10 illustrates this algorithm for our previous longest path example. The algorithm begins with the longest path graph G' shown in the upper left-hand corner of the diagram. From this graph, an undirected graph G'' is constructed, containing only the paths from source A to vertex I. Special vertices representing the graph edges have

[2] From a private communication with Roberto Tamassia.




```
The longest path to vertex I has length [ 21.00, 23.00]
The set of all possible longest paths:
  From vertex A to B with edge length [  6.00,  6.00] = [  6.00,  6.00]
  From vertex B to E with edge length [  2.00,  2.00] = [  8.00,  8.00]
  From vertex E to C with edge length [  1.00,  1.00] = [  9.00,  9.00]
  From vertex C to F with edge length [  3.00,  5.00] = [ 12.00, 14.00]
  From vertex F to H with edge length [  1.00,  1.00] = [ 13.00, 15.00]
  From vertex H to I with edge length [  8.00,  8.00] = [ 21.00, 23.00]
    Path length = [ 21.00, 23.00]

  From vertex A to B with edge length [  6.00,  6.00] = [  6.00,  6.00]
  From vertex B to E with edge length [  2.00,  2.00] = [  8.00,  8.00]
  From vertex E to H with edge length [  4.00,  6.00] = [ 12.00, 14.00]
  From vertex H to I with edge length [  8.00,  8.00] = [ 20.00, 22.00]
    Path length = [20.00,22.00]
```

Figure 5.8: Computing Longest Paths with Intervals

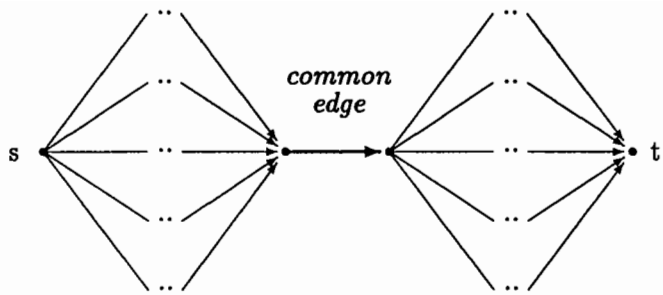


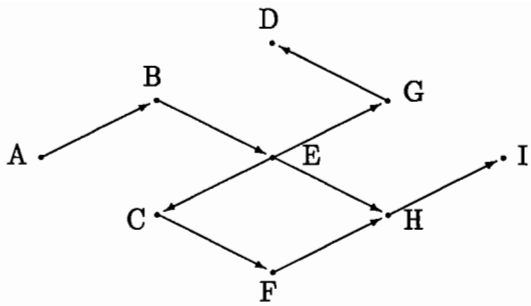
Figure 5.9: Common Edges

also been added to the graph. These vertices are assigned unique identifiers (in lower-case letters) denoting their edge endpoints. The third graph displays the articulation points in graph G''. As shown, vertices B, E, H, ab', be', and hi' are all articulation points. The special vertices among these articulation points represent common edges; therefore, edges (A,B), (B,E), and (H,I) are the edges common to all longest paths between vertices A and I. These edges are highlighted in the final graph.

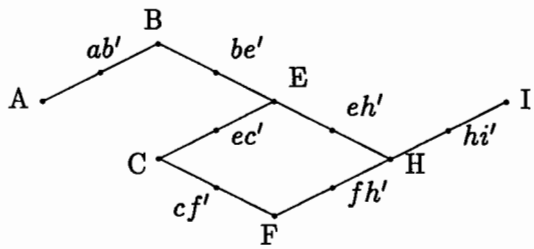
5.2.4 A Timing Example

In Figure 5.11, we present the critical path analysis for the nMOS circuit in Figure 5.1. The first graph in Figure 5.11 depicts the timing dependencies between the labeled nodes in the circuit. The graph vertices represent the input, output, and target nodes in the circuit and the edges represent the timing dependencies between these nodes. For instance, a rising signal value at node C allows current to flow from GND to node E, thus producing a falling signal value at node E. Associated with each edge (v, w) is the delay (ns) along the path through the circuit to vertex w triggered by the signal change at vertex v. These delays were computed using the interval RC analysis algorithm presented in Chapter 3. Because the original timing dependency graph does not have a single source node, we add source vertex S and a dashed edge from S to each of the circuit inputs (A,B, and C). These dashed edges are labeled with the *input arrival times* for each input node.

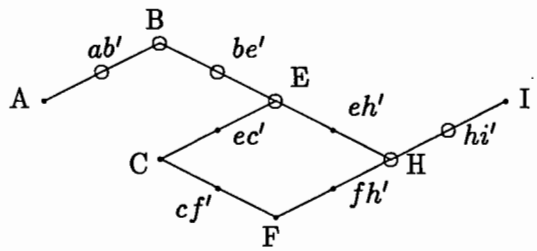
Given the timing dependency graph and the source vertex S, we used our interval longest path algorithm to compute the critical paths through the network. The second



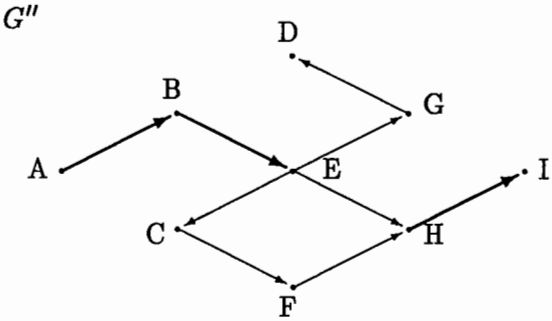
Longest Path Graph G' with source vertex A



Graph G'' for paths to vertex I



Articulation Points in G''



Common Edges in Longest Paths from A to I

Figure 5.10: Finding Common Edges

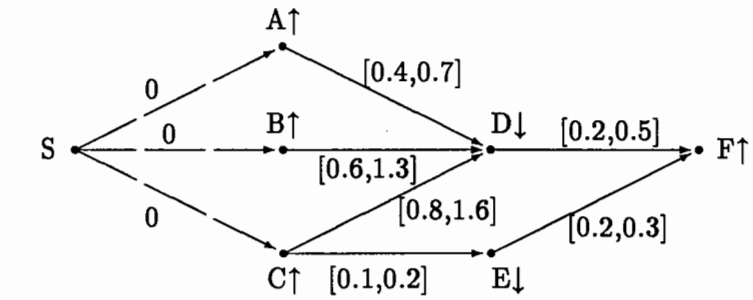
graph in Figure 5.11 presents the results of this computation. This graph depicts the set of longest paths and the maximum delay (in ns) from the source vertex to each node in the graph. The listing beneath this graph gives the details of the longest delay paths from S to F. As shown, there are two longest paths to node F. One is activated by the signal change at node B and the second by the change at node C. The maximum signal delay to node F is $[1.0, 2.1]$ ns. This means that a rising signal at the inputs of the circuit will require from 1.0 to 2.1 ns to propagate to the output through the slowest paths in the circuit.

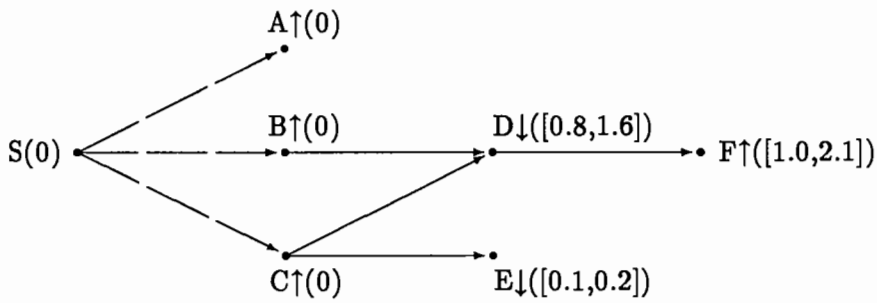
5.3 Proof of the Method

In this section, we formally prove that our interval algorithms compute the correct bounds on longest or shortest path length on a graph with uncertain edge weights. The classical shortest path problem is an algebraic path problem defined over the closed semi-ring $(\Re, min, +, \infty, 0)$, where $\Re$ includes ∞ and $-\infty$. In [Aho74, pp. 195-201], Aho, Hopcroft and Ullman prove that the classical algorithm for computing shortest path length is correct by the properties of this algebra. Let I denote the set of intervals over $\Re$. We claim that our interval algebra $(I, MIN, \oplus, [\infty, \infty], [0, 0])$ also forms a closed semi-ring. Hence, we conclude that if the original algorithm using the algebra $(\Re, min, +, \infty, 0)$ correctly computes shortest path lengths on graphs with discrete edge weights, then our interval version employing the algebra $(I, MIN, \oplus, [\infty, \infty], [0, 0])$ will correctly calculate bounds on shortest path length on graphs with interval-weighted edges. A similar argument can be made for the algebra $(I, MAX, \oplus, [-\infty, -\infty], [0, 0])$ used when computing longest paths. We now prove that our interval algebra forms a closed semi-ring.

Theorem 5.1 $(I, MIN, \oplus, [\infty, \infty], [0, 0])$ *forms a closed semi-ring.*

Proof. First, we must show that $(I, MIN, [\infty, \infty])$ is a monoid. As shown in Appendix A, MIN is closed, associative, and has identity $[\infty, \infty]$. Similarly, $(I, \oplus, [0, 0])$ is also a monoid. Since ∞ is the annihilator for $+$, $[\infty, \infty]$ is the annihilator for $\oplus$. From the properties of min, we see that MIN is commutative and idempotent. Next, we must show that $\oplus$ distributes over MIN; thus, we must show that $MIN([a, b], [c, d]) \oplus [e, f] = MIN([a, b] \oplus [e, f], [c, d] \oplus [e, f])$ and $[e, f] \oplus MIN([a, b], [c, d]) = MIN([e, f] \oplus [a, b], [e, f] \oplus [c, d])$ for all $[a, b], [c, d], [e, f] \in I$. Applying the definitions of MIN and $\oplus$,






```
The longest path to vertex F has length [  1.00,  2.10]
The set of all possible longest paths:
  From vertex S to C with edge length [  0.00,  0.00] = [  0.00,  0.00]
  From vertex C to D with edge length [  0.80,  1.60] = [  0.80,  1.60]
  From vertex D to F with edge length [  0.20,  0.50] = [  1.00,  2.10]
    Path length = [  1.00,  2.10]

  From vertex S to B with edge length [  0.00,  0.00] = [  0.00,  0.00]
  From vertex B to D with edge length [  0.60,  1.30] = [  0.60,  1.30]
  From vertex D to F with edge length [  0.20,  0.50] = [  0.80,  1.80]
    Path length = [  0.80,  1.80]
```

Figure 5.11: Longest Path Analysis for Timing Verification

we must show that $[min(a,c)+e, min(b,d)+f] = [min(a+e, c+e), min(b+f, d+f)]$ and $[e + min(a,c), f + min(b,d)] = [min(e+a, e+c), min(f+b, f+d)]$. Because $+$ distributes over min, this is obviously true. Next, we must show that if $\hat{a}_1, \hat{a}_2, ...$ is countable sequence of elements in I, then $MIN(\hat{a}_1, \hat{a}_2, ...)$ exists and is unique. Furthermore, we must show that MIN is associative, commutative, and idempotent over infinite as well as finite sequences. Since these properties hold for min and MIN is defined in terms of min, these properties hold for MIN. Finally, $\oplus$ must distribute over MIN for countably infinite sequences as well as finite ones. This follows from the definitions of the interval operators and the fact that $+$ distributes over min for countably infinite sequences. □

5.4 Performance

Both the longest and shortest path algorithms described in this chapter were implemented in the C programming language and run on a SUN SPARCstation. Since most VLSI applications of the longest path algorithm employ either breadth or depth-first scanning orders, we incorporated both of them in our implementations. In this section, we discuss the performance of the algorithms. In particular, we examine their running time and the quality of the bounds they produce.

5.4.1 Quality of the Bounds

In timing verification, the longest path algorithm is employed to compute the delay through a circuit. Here, edge lengths correspond to stage delays and are calculated using RC timing analysis. As we stated in Chapter 3, the interval methods for computing stage delays may produce overly conservative bounds for some circuits. An interesting question is how does this "error" in edge length (stage delay) affect path length (circuit delay)? We now show that edge error does not accumulate. Let p be any path through graph G such that path p consists of k edges with actual lengths $\hat{l}_1, \hat{l}_2, ..., \hat{l}_k$. For each edge i, let l_i represent the upper (lower) bound of $\hat{l}_i$. Now let $\hat{c}_1, \hat{c}_2, ..., \hat{c}_k$ be the lengths that we are given for those edges and let c_i denote the upper (lower) bound of $\hat{c}_i$. The percent error e_i in the upper (lower) bound of each edge i is $e_i = \frac{l_i - c_i}{l_i} \cdot 100$. Let L represent the actual upper (lower) bound on the length of path p; thus, $L = \sum_{i=1}^{k} l_i$. Likewise, let C be the computed value for path p; thus, $C = \sum_{i=1}^{k} c_i$. Let E represent

the percent error in the upper (lower) bound on path length for p. We now prove that path length error E is bounded by maximum edge length error; thus,

Theorem 5.2 $E \leq max(e_1, e_2, ..., e_k)$

Proof. By definition, $E = \frac{L-C}{L} \cdot 100$. Expanding L and C, $E = \frac{(l_1+...+l_k)-(c_1+...+c_k)}{L} \cdot 100$. This equation can be rewritten as $E = \frac{(l_1-c_1)+...+(l_k-c_k)}{L} \cdot 100$; therefore, $E = \sum_{i=1}^{k} \frac{(l_i-c_i)}{L} \cdot 100$. Multiplying each term by $\frac{l_i}{l_i}$ and rearranging yields $E = \sum_{i=1}^{k} \frac{l_i}{L}(\frac{l_i-c_i}{l_i} \cdot 100)$. But $\frac{l_i-c_i}{l_i} \cdot 100$ is just e_i; therefore, $E = \sum_{i=1}^{k} \frac{l_i}{L} e_i$. This equation shows that path length error is a weighted sum of the edge length errors. Let $\overline{e} = max(e_1, e_2, ..., e_k)$. Clearly, $E \leq \sum_{i=1}^{k} \frac{l_i}{L} \overline{e}$. But $\sum_{i=1}^{k} \frac{l_i}{L} = 1$; therefore, $E \leq \overline{e} \cdot 1$. We have just proved that $E \leq max(e_1, e_2, ..., e_k)$. □

To illustrate this theorem, Tables 5.3 and 5.4 present an example from timing analysis. Table 5.3 displays the stage delays and total path delay from an actual critical path calculation for a portion of the B-SYS chip [Hugh89]. The *Theoretical* column contains the theoretical stage and path delay bounds and the remaining columns contain bounds generated by various interval and non-interval methods. In Table 5.4, we present the errors in stage and path delay for each method. As expected, the path delay error for each case is less than or equal to the maximum error of its individual stages. Using this theorem, we also conclude that if the bounds on edge length are correct, then our algorithm is guaranteed to find the tightest possible bounds on path length.

5.4.2 Running Time of the Algorithm

Our interval algorithms differ from the non-interval versions in two main areas. First, our interval versions employ interval operations. Since we must perform two discrete operations to evaluate each interval one, we expect that our interval algorithms will run twice as slow as the originals. Second, the interval versions associate a set of parents with each vertex, instead of a single parent. The cost of maintaining these sets decreases the efficiency of the interval versions. If we implement parent sets using heaps, then insertions and deletions each cost $O(\log n)$, where n indicates the size of the heap. In the worst-case, n is the number of vertices in the graph. Hence, with this implementation, the operating speed of the interval algorithms will decrease by a factor of $O(\log n)$ in the worst case.

Stage	Size	Theoretical		Intcryst		Monte Carlo		Nom
		min	max	min	max	min	max	
1	2	6.84	9.28	6.82	9.31	7.24	8.73	7.98
2	3	54.28	72.90	54.14	73.08	56.55	69.49	62.98
3	2	10.34	14.12	10.33	14.14	10.84	13.37	12.10
4	2	10.61	14.43	10.61	14.44	11.01	13.90	12.39
5	2	1.16	1.62	1.15	1.63	1.22	1.53	1.37
6	1	0.70	0.98	0.70	0.98	0.75	0.93	0.83
7	1	0.77	1.09	0.77	1.09	0.80	1.04	0.92
8	3	20.41	27.40	20.33	27.51	21.24	26.54	23.68
9	1	1.97	2.65	1.97	2.65	2.00	2.61	2.29
Path total		107.08	144.47	106.82	144.83	111.65	138.14	124.54

Table 5.3: A Critical Path Delay Calculation (ns)

Stage	Size	Intcryst		Monte Carlo		Nominal	
		min	max	min	max	min	max
1	2	0.29	-0.32	-5.85	5.93	-16.67	14.01
2	3	0.26	-0.25	-4.18	4.68	-16.03	13.61
3	2	0.10	-0.14	-4.84	5.31	-17.02	14.31
4	2	0	-0.07	-3.77	3.67	-16.78	14.14
5	2	0.86	-0.62	-5.17	5.65	-18.10	15.43
6	1	0	0	-7.14	5.10	-18.57	15.31
7	1	0	0	-3.90	4.59	-19.48	15.60
8	3	0.39	-0.40	-4.07	3.14	-16.02	13.58
9	1	0	0	-1.52	1.51	-16.24	13.58
Path total		0.24	-0.25	-4.27	4.38	-16.31	13.80

Table 5.4: % Error in Critical Path Delay Bounds

To confirm this hypothesis, we compared the operating speeds of our algorithms with those of the non-interval versions. As search strategy greatly affects performance, we tested implementations containing both breadth and depth-first scanning orders. We ran both the interval and non-interval implementations of each algorithm on identical graphs and compared their response times. Our tests showed that the interval implementations ran roughly twice as slow as their non-interval counterparts. These results suggest that the increase in operating speed was due largely to the interval operations, rather than the parent lists. A survey of these graphs confirmed this suspicion, demonstrating that most parent lists contain only a few entries.

5.5 Summary

In this chapter, we presented a variation on the classical longest/shortest path problem by introducing uncertain edge lengths. Instances of this problem occur in many areas of VLSI design, including placement and timing verification. To solve this problem, we represented uncertain edge lengths as intervals and created an interval algebra for computing bounds on path length. We then used this algebra to create an algorithm which not only computes bounds on path length, but also returns the set of paths that generated these bounds. We then presented a proof of correctness for our approach.

After implementing our algorithms, we tested their performance. We proved that path length error is bounded by the maximum error of its edge lengths; therefore, if the edge lengths are correct, our algorithms generate the tightest possible bounds on path length. We also tested the speed of these algorithms as compared to their non-interval counterparts. Analysis suggests that the interval versions may be slower than the non-interval ones by $O(\log n)$ in the worst case. Experiments on several graphs, however, demonstrated that the interval versions are not substantially slower than their non-interval counterparts.

Chapter 6

Placement using Uncertain Costs

Rapid increases in circuit size and complexity in the past ten years have made automatic placement an essential element of the VLSI design tool suite. The goal of automatic placement is to find an arrangement of the circuit components that facilitates routing and optimizes certain aspects of the circuit. These goals are encoded within an *objective function*, which provides a measure of the quality of each configuration. [Prea86] presents an overview of the most common automatic placement techniques. These can be divided into two major categories: constructive and iterative. The constructive approaches begin with a partial placement and build a complete placement from it. Typical examples of these algorithms include cluster growth [Dai89], min-cut partitioning [Breu77, Fidu82, Kern70, MS89, Tric86], global placement [Jack86], and branch and bound [Dai89, Prea79, Tric86]. Iterative techniques, such as pairwise interchange and simulated annealing [Kirk83], start with a complete placement and attempt to improve it. A more detailed description of each of these placement methods appears in [Prea86].

In spite of their differences, all placement algorithms share a number of common features. They all evaluate an objective function for each placement generated, compare the values returned by the objective functions to chose the best configuration of elements, and save the best result found. While there are almost as many different objective functions as placement programs, the most common measures of placement quality are circuit area, channel density, maximum wire length, and circuit delay. The

first of these metrics is layout area. To be accurate, this computation must include component and routing area. A problem, however, arises when these areas are not precisely known. In top-down or hierarchical placement, the dimensions of higher-level cells are determined by placements at lower-levels in the hierarchy. If the lower-level cells are not completely placed and routed, the exact dimensions of the higher-level cells containing them may not be known. Similarly, channel width can only be accurately determined by doing a complete routing. This calculation is computationally expensive and thus is practical only for the final placement configuration. While exploring placement alternatives, the routing area is usually estimated.

The second common placement objective is to minimize wire congestion over the circuit. Channels containing many nets have to be wider to accommodate the interconnections and may be difficult to route. Reducing wire congestion increases routability and reduces circuit area. Congestion is calculated by summing the channel density of all channels. Channel density is typically computed by counting the number of nets that cross a routing channel.

Reducing total wire length is another standard goal. Total wire length is defined as the sum over the entire circuit of the lengths of all nets. As with channel width, this can only be accurately calculated after full routing; therefore, wire length for a single wire is usually approximated by computing the half perimeter of the rectangular bounding box containing the net.

A fourth common layout metric is circuit delay. A placement with long wire delays along its critical paths can significantly degrade the performance of a circuit; therefore, many current placement algorithms incorporate delay information in their objective functions. One such method locates the critical delay paths through the circuit and uses these delays to weight the nets. Those nets along the critical path then receive a higher priority in the placement algorithm. By carefully placing the components from this worst-delay path, the designer hopes to reduce wire delay along the critical path and consequently the total delay of the circuit. The delays used to compute these weighting factors are subject to variation. Consequently, logic-level timing verifiers [McWi80, Hitc82] frequently represent component and wire delays as min-max delay bounds or statistical distributions.

This brief survey of the most common measures of placement quality reveals that many objective functions contain parameters whose exact values cannot be predicted in advance. Among them are cell area, routing area, wire length, and circuit delay. While

these values can be estimated, they cannot be precisely known before the layout is finished or the chip is fabricated. In spite of the prevalence of uncertainty in placement metrics, current placement algorithms use only single-valued estimates in their objective functions.

Placement programs that account for uncertainty offer several advantages over current systems. One advantage is a more comprehensive measure of placement quality. Since the parameters used in objective functions have variable values, different combinations of their values will result in different placement costs and possibly a different "best" configuration. In contrast to conventional placement algorithms which return a single optimal configuration, algorithms incorporating uncertainty will return a set of placements, each of which is optimal for some assignment of values. As the placement process continues, the values of these parameters may be refined, allowing the designer to chose one placement from the set of potentially optimal placements. Since it is unlikely that the final values of each parameter will exactly match the precomputed single-valued estimates, the final layout may be quite different from the one generated using a conventional placement algorithm.

Additional information about the best layout is the second advantage to including uncertainty. Systems that return a single configuration give no details about the features of the layout that make it optimal. By comparing the layouts from a set of best configurations, we begin to discover the similarities between them. These similarities reveal the factors that produced the best layouts for a particular circuit. For instance, certain components may appear in the same location in all potentially optimal placements. Also, strongly connected components may always appear in the same relative position. In addition to increasing our understanding of the design, these invariants may also indicate critical areas on the chip, which can be changed to further improve the placement.

A third advantage of incorporating uncertainty is that it facilitates iterative improvement without costly recomputation. As placement proceeds, the values of some uncertain parameters may be refined or may change. For instance, in hierarchical placement, the dimensions of the blocks at higher-levels will be refined as the layouts are completed at the lower-levels of the hierarchy. Similarly, channel width and wire length estimates are updated after full routing. Using current placement algorithms, the designer must re-execute the complete algorithm each time these values change. Since finding a good placement is a computationally expensive process, this is not practical.

Using an algorithm that models uncertainty, we may not have to completely recompute the layout after these changes. As long as the updated values or ranges lie within the ranges initially given for each parameter, the previous placement results are still valid. Instead of running the placement algorithm again, we simply recompute the cost of the placements in the optimal set and discard those that no longer belong.

For these reasons, we propose a framework for placement with uncertain objective functions. Traditional systems employ only single-valued estimates of parameter values in their objective functions, returning one possible cost for each configuration. Using these values, the algorithm returns the configuration whose cost in minimum. Instead of ignoring parameter variation, systems incorporating uncertainty use the full range of possible values for each variable to compute bounds on the cost of a configuration. Using these cost ranges, it returns a set of placements, each of which is optimal for some combination of input values.

In this chapter, we present one solution to this problem. As before, we represent placement costs as intervals and create an algebra for manipulating these uncertain values. We then incorporate this algebra within an existing placement algorithm [Prea79] to create one that handles uncertain objective functions. Although a branch and bound placement algorithm was chosen, the general principles employed apply to other placement strategies. In particular, we address the common issues of evaluating an uncertain objective function, comparing two placements with uncertain costs to find the best configurations, and storing the resulting minimum-cost placements.

6.1 A Branch and Bound Placement Algorithm

In [Prea79], Preas and vanCleemput present a branch and bound placement algorithm for arbitrarily-sized blocks. Because of its exponential computational complexity, the branch and bound technique is only practical for circuits with relatively few components (i.e., fewer than 20). Hence the placement algorithm begins by partitioning the circuit to create several smaller layout problems of a suitable size. The result of top-down partitioning is a hierarchical circuit description, where each level contains a collection of rectangular circuit components and the interconnections between them. Beginning at the bottom levels of the hierarchy, a constructive branch and bound algorithm is applied recursively to compute an initial placement. Because of the size of the search space for some larger circuits, the constructive algorithm may require an unreasonably

long time to find the optimal solution. For these cases, Preas and vanCleemput propose an iterative improvement scheme based on the same branch and bound strategy to improve an existing layout.

6.1.1 The Objective Function

The goal of this placement algorithm is to minimize circuit area; thus, we must estimate the size of each trial placement. Preas and vanCleemput begin by creating the *horizontal* and *vertical channel position graphs* that define the layout. A channel position graph is a directed acyclic graph whose nodes represent the boundaries between circuit components and channels and whose edges represent the channels and circuit blocks themselves. There are two special nodes in each graph depicting the right and left (top and bottom) boundaries of the layout. Associated with each edge is a *length*, representing the physical dimensions of the corresponding block or channel. At the lowest level of the hierarchy, the dimensions of the circuit components are given. At higher levels, circuit block size is computed from layouts generated at lower levels of the hierarchy. Channel width is estimated for all trial placements, but full routing is used to determine the channel width for the final configuration.

Figure 6.1 depicts a sample layout and its corresponding horizontal and vertical channel position graphs[1]. The graph edges corresponding to circuit components are denoted by block identifiers and edges corresponding to channels are labeled with the letter H (horizontal channels) or V (vertical channels) and are given unique identifiers. Each graph edge is also labeled with the estimated length or width of its associated block or channel in parentheses. All dimensions are given in mils.

Once the channel position graphs for a particular placement are created, we traverse the graphs to find the longest paths between the boundary nodes. The length of this longest path gives the minimum length (or width) of the layout. These dimensions are then multiplied to determine circuit area. For the example in Figure 6.1, the longest path through the vertical channel position graph has length 92 mils and determines the minimum length of the circuit. Likewise, the longest path through the horizontal channel position graph has length 71 mils and determines the minimum width of the layout. Hence, the minimum area of this configuration is 6532 square mils.

[1]This layout was adapted from a circuit in [Prea79]

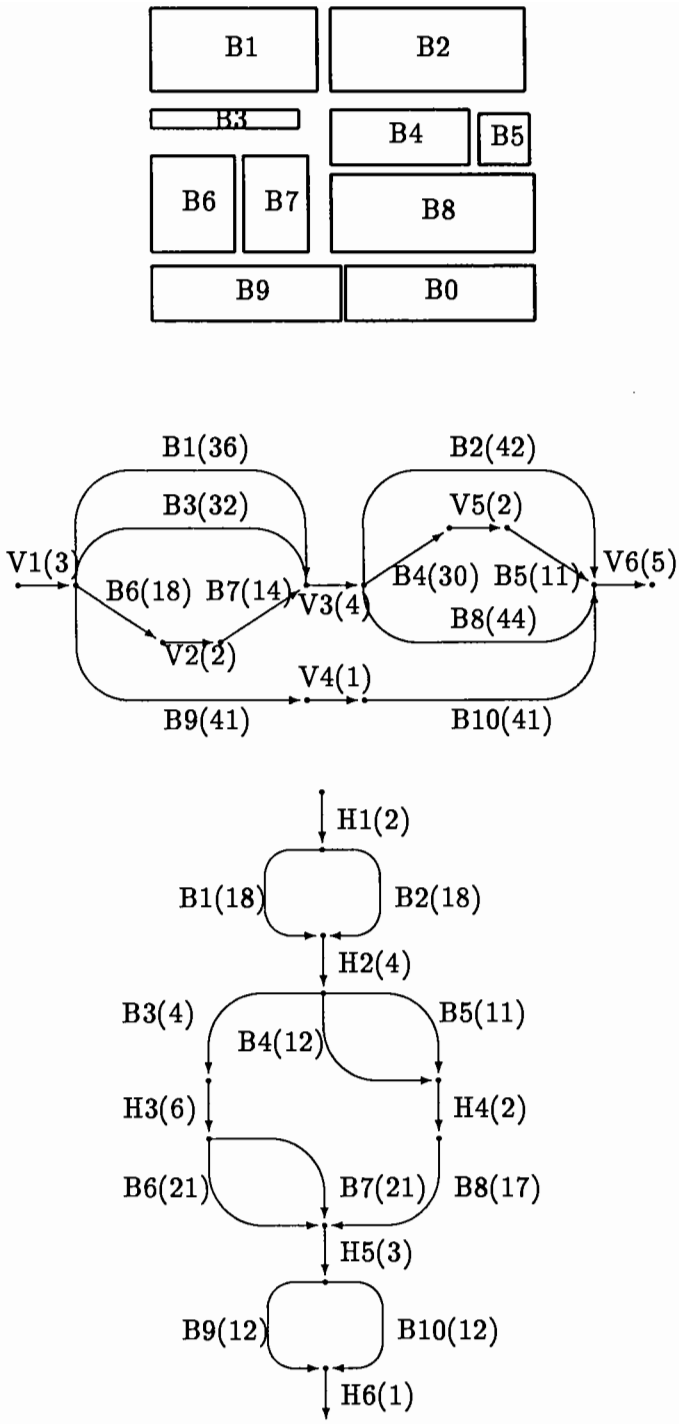


Figure 6.1: Horizontal and Vertical Channel Position Graphs

6.1.2 Branch and Bound Placement

Given a set of circuit blocks and their interconnections, Preas and vanCleemput employ branch and bound techniques to find the configuration with minimum area. Branch and bound is a well-known potentially-exhaustive search strategy [Aho83, pp. 330-336] that traverses a search tree to locate the minimum cost solution. Each node in the search tree represents a set of solutions. At each successive level of the tree, those solutions are refined, until each leaf node contains only a single solution. Associated with each node is a number, indicating a lower bound on its cost. When visiting a node, we compare its cost to that of the current best complete solution. If the cost of the node is greater than or equal to the current minimum cost, we prune that solution from the search tree; otherwise, we consider each of its children and compute their costs. At the leaf nodes, we locate and save the one with the lowest cost. When all the nodes have been either visited or pruned, the search is complete and the best solution is returned.

For the placement problem, the internal nodes of the search tree represent partial placements and the leaf nodes represent complete placements. The root of the tree contains a *seed* placement, consisting of one or more blocks and their positions relative to each other. By adding unplaced blocks to a partial placement, we generate its child configurations. Since our objective function is circuit area, the cost of each (partial or complete) placement is its size as computed from its channel position graphs. The area of the smallest complete placement found in the course of the search is used to prune successive configurations. Initially, we require an estimate of minimum layout area to prune partial placements until a complete placement with lower cost is found. This initial estimate is either derived from designer specifications or computed from a given complete layout. The branch and bound placement algorithm is summarized in Figure 6.2.

To illustrate this algorithm, Figure 6.3 displays a complete branch and bound search tree for a circuit consisting of three components with fixed pin positions and connections. The average dimensions of the three circuit components are shown in Table 6.1. Each node in the tree represents a partial or complete placement. Below each configuration is a placement identifier (e.g., P1, P2, ..., Pn) followed by placement cost. Although routing is not shown in the diagram, its effects are evident in the cost metric. This explains why symmetric placement configurations have slightly different costs.

```
Initialize minimum cost
Generate seed placement
Add placement to search tree
While there are unvisited placements on search tree
    Choose placement from search tree
    If placement cost ≥ minimum cost
        Prune placement from search tree
    Else if placement is complete
        Minimum cost = placement cost
        Save placement as current best
    Else (placement is partial and unpruned, so expand it)
        Choose unplaced block
        For each possible position in partial placement
            Create child placement by adding unplaced block at position
            Compute child placement cost
            If child cost ≥ minimum cost
                Prune child from search tree
        End for
End while
Return the minimum placement
```

Figure 6.2: Branch and Bound Placement Algorithm (Preas and vanCleemput)

The seed placement P1 contains a single block. Its child placements P2-5 are created by adding a second block to the seed placement in all possible positions. The rest of the configurations are leaf nodes representing complete placements consisting of all three blocks. An X below a layout indicates that the placement was pruned in the course of the search. Assume that we are given an initial cost bound of 1200. After examining nodes P1, P2, and P6, we find a complete placement with cost 1100. P6 now becomes the minimum-cost placement. The only other child of P2 with lower cost is P11, which then becomes the best placement. We now examine P3 and its children. Finding none with lower cost, we proceed to P4 and find P25 with cost 960. Since P5's cost is greater that of the current best solution, we prune P5 and do not generate any of its children. At the end of the search, the optimal configuration is P25 with cost 960.

Figure 6.4 presents the results of a layout computation on a circuit with ten cell blocks[2]. The average dimensions (in mils) of each block are displayed in Table 6.3. After generating and examining more than 180,000 partial and complete configurations, the algorithm returned the minimum-cost placement displayed in Figure 6.4. The resulting configuration has an area of 1998 square mils. Below the layout, the horizontal and vertical channel position graphs defining this optimal placement are displayed. As shown the longest paths through the horizontal and vertical channel position graphs are 37 and 54 mils respectively.

6.2 An Interval Placement Algorithm

Having described the branch and bound placement algorithm, we now discuss the changes that must be made to this algorithm to incorporate uncertain objective functions. While its general structure remains unaffected, three aspects of the algorithm do change. They are

- Computing uncertain cost functions
- Comparing values returned by these cost functions
- Saving minimum-cost placements

[2]This example was adapted from a circuit in [Cies84]

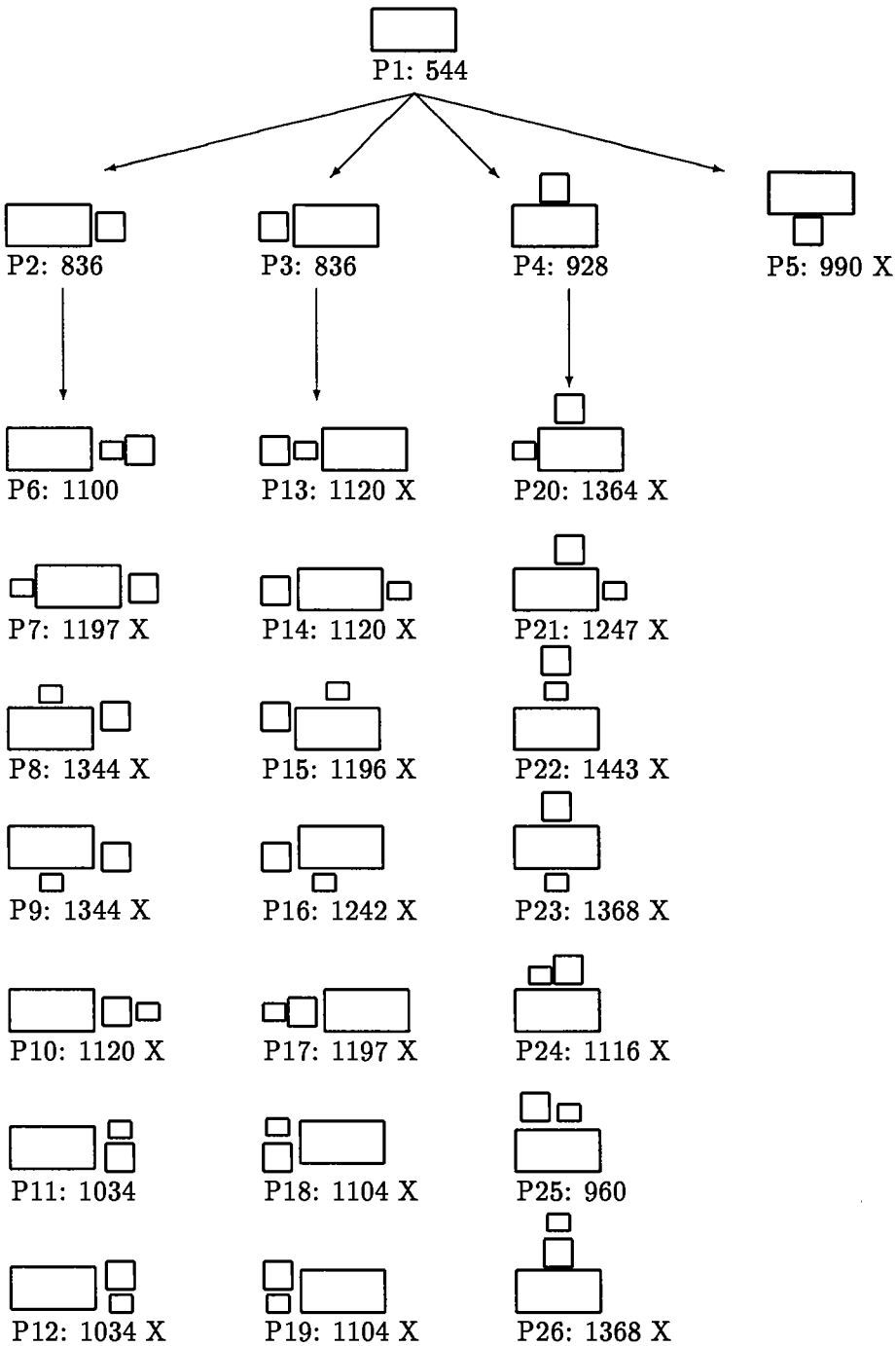


Figure 6.3: Branch and Bound Search Tree for Placement

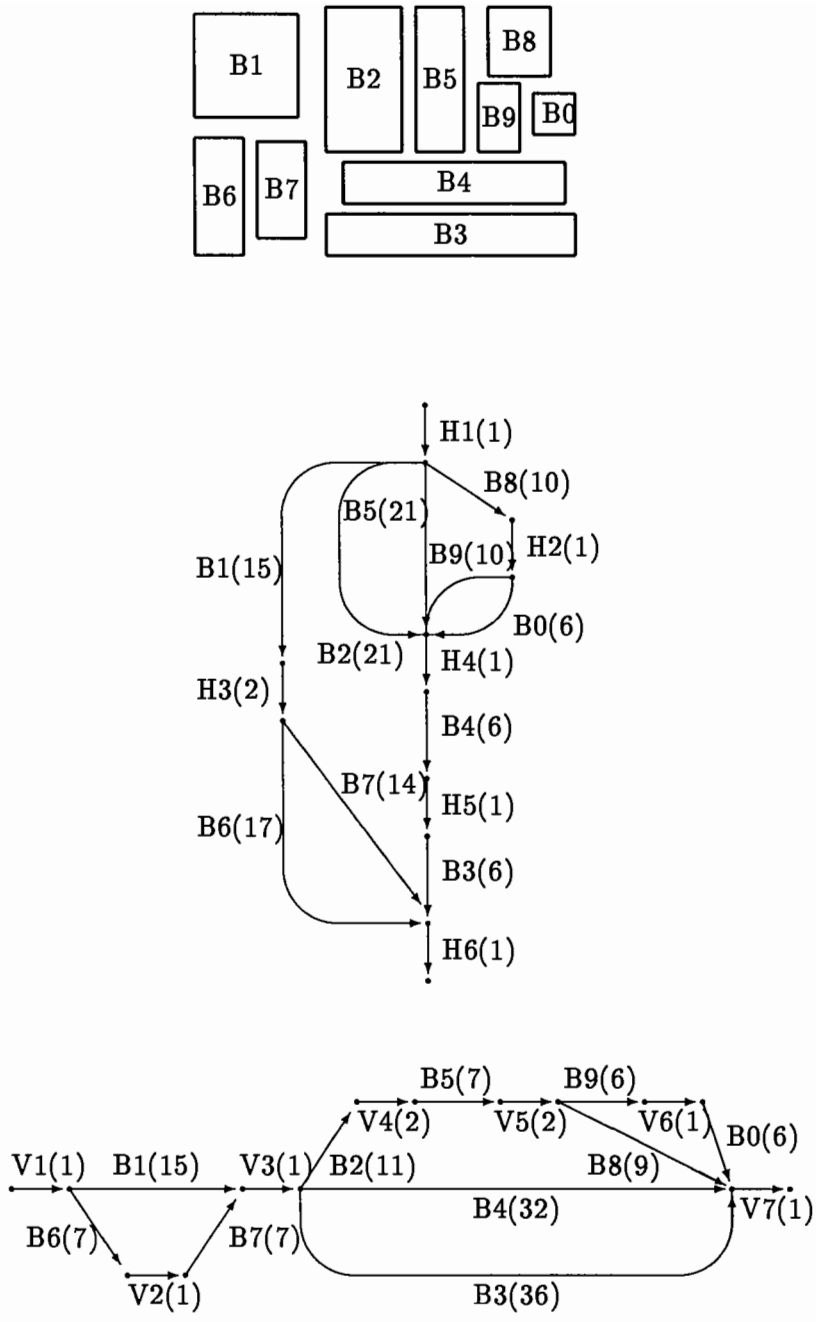


Figure 6.4: Resulting Minimum-Cost Placement with Channel Position Graphs

6.2.1 Computing an Interval Objective Function

The first of these issues is how to calculate an interval objective function. Since there are many different objective functions in current placement programs, it would be impractical to enumerate them all. Instead, we present a general methodology for creating an interval objective function and then illustrate it with a single example (circuit area).

To compute an interval objective function, we represent all uncertain values (e.g., channel widths, component sizes, wire lengths, component delays) in the cost function as intervals. We then create an application-specific interval algebra for manipulating these ranges. The interval operators in this algebra are the logical interval extensions of the discrete operators employed to compute the objective function. These interval operators F are derived from their non-interval counterparts f using the following standard formula [Moor79]:

$$F(\hat{x}, \hat{y}) \equiv \{f(x, y) \mid x \in \hat{x} \text{ and } y \in \hat{y}\}. \tag{6.1}$$

Having derived the equivalent interval operators, we then substitute them into the objective function to create an interval version of it. The resulting interval objective function returns bounds on its value when given interval inputs.

To illustrate this method, suppose that our objective function is layout area. Since the cell dimensions and channel widths needed to compute area may not be known exactly, we represent their values as intervals. Now we must create an interval algebra for manipulating these ranges. The non-interval objective function described in section 6.1.1 employs three operations to compute layout area: $+$ to sum the edge lengths in the graph, max to compare path lengths to determine the longest one, and $\cdot$ to multiply the longest path lengths to calculate the area. For the interval version, we derive the interval extensions $\oplus$ (addition), MAX (maximum), and $\odot$ (multiplication) of these operators using Equation 6.1. The resulting interval operators are defined as follows:

$$[a, b] \oplus [c, d] \equiv [a + c, b + d] \tag{6.2}$$

$$[a, b] \odot [c, d] \equiv [min(ac, ad, bc, bd), max(ac, ad, bc, bd)] \tag{6.3}$$

$$MAX([a, b], [c, d]) \equiv [max(a, c), max(b, d)] \tag{6.4}$$

To compute the dimensions of a given layout, the non-interval version employs a standard longest path algorithm[Tarj83, pp. 85-96] to calculate the length of the

longest path through each channel position graph. By replacing the discrete operators in this algorithm with their interval counterparts, we create an interval longest path algorithm that returns bounds on the length of the longest paths through a directed graph (See Chapter 5 for details). Once bounds on the length and width of the layout are found, the $\odot$ operator can be used to compute bounds on circuit area.

6.2.2 Comparing Placement Cost

Comparing the values returned by the objective functions is the second issue that we must address. Placement costs are used to prune unpromising placements from the search and to choose the best placement. To perform these tasks, we must be able to compare two interval placement costs to determine which is smaller.

In the first instance, we compare a placement cost to the current minimum cost, pruning the placement from the search if its cost exceeds the current minimum value. With interval cost functions, we need an interval comparator. We define the interval comparator $>$ as follows:

$$[a, b] > [c, d] \Longleftrightarrow a > d. \tag{6.5}$$

According to this definition, interval $[a, b]$ is greater than $[c, d]$ if and only if each member of the first interval range is greater than all members of the second. In our interval placement problem, a placement is thus pruned from the search tree if and only if all members of its cost range are greater than all members of the current minimum bound. This guarantees that we already have a better solution for all possible placement cost values; therefore, this placement cannot be a best placement and thus can be discarded.

Second, these metrics are used to compute the minimum placement cost. In our interval placement algorithm, the costs are now intervals; therefore, we need an interval MIN operator to calculate the minimum of two placement costs. This operator is derived from the discrete operator *min* and is defined as follows:

$$MIN([a, b], [c, d]) \equiv [min(a, c), min(b, d)]. \tag{6.6}$$

The interval returned by this operator represents the set of minimum costs obtained by comparing two placements over all possible combinations of their values.

6.2.3 Saving the Minimum-Cost Results

It is not enough to know the minimum placement cost, we must also save the placement configurations themselves. The original placement algorithm computes the minimum placement cost $a \in \Re$ and returns a single representative configuration with cost a. In reality, there may be several configurations with minimum or close to minimum cost, any one of which could be used in the final design. Our interval version of the placement algorithm returns bounds $\hat{a} \in I$ on minimum placement cost and a *set* of configurations such that each placement in this set has a cost that lies within $\hat{a}$. Under some scenario, any of these placements could be optimal. Since each of these placements represents a potential best configuration, we must modify the algorithm to save them. Whenever we locate a complete placement with cost $\hat{c}$ such that $\hat{a} \cap \hat{c} \neq \emptyset$, we first update the minimum placement cost using the MIN operator described above and add the new placement to the set. Since the minimum placement cost may have changed, we must update the set of optimal placements by deleting all placements whose costs now exceed the new minimum cost bound. To perform this function, we use the previously-defined interval $>$ operator. For example, let $\hat{b}$ represent the cost of a particular configuration in the optimal set and let $\hat{a}$ represent the updated minimum-cost bounds. If $\hat{b} > \hat{a}$, then for each element of $\hat{b}$ there exists another configuration in the optimal set with smaller cost. Since this placement cannot be an optimal placement under any scenario, we must delete it from the set.

The set of optimal configurations can be implemented in a variety of ways, including linked lists, binary trees, and heaps. One possibility is a linear linked list sorted in decreasing order by the lower bound on placement cost. Inserting a placement in this list has a worst-case time complexity of $O(n)$, where n is the number of configurations in the set. Deleting an item is much more efficient, since the placements with the largest areas are located at the front of the list and can be retrieved in $O(1)$ time.

Another possible implementation is the binary tree. In this case, placements are ordered such that the minimum cost of the left child is less than that of its parent. Likewise, the minimum cost of the right child is greater than that of its parent. This implementation yields a worst-case insertion time of $O(n)$ and an average time of $O(\log n)$. The deletion times are similar. With a little extra computational effort, we can maintain a balanced binary tree, thus guaranteeing insertion and deletion costs of $O(\log n)$.

```
Initialize minimum cost bound
Generate seed placement
Add placement to search tree
While there are unvisited placements on search tree
    Choose placement from search tree
    If placement cost > minimum cost
        Prune placement from search tree
    Else if placement is complete
        Minimum cost = MIN(minimum cost, placement cost)
        Prune minimum list of all placements whose cost > minimum cost
        Add new placement to list of minimum placements
    Else (placement is partial and unpruned, so expand it)
        Choose unplaced block
        For each possible position in partial placement
            Create child placement by adding unplaced block at position
            Compute child placement cost
            If child cost > minimum cost
                Prune child from search tree
        End for
End while
Return the minimum placements
```

Figure 6.5: Interval Branch and Bound Placement Algorithm

The third and best of these set implementations is a heap with the condition that the minimum cost of each parent configuration is greater than those of its children. The cost of inserting a placement in this structure is $O(\log n)$. Since the placement with the greatest cost is kept at the top of the heap, this item can be retrieved in $O(1)$ time. Restoring the heap property after a deletion requires $O(\log n)$ time. As a result, the total cost of deleting a placement is also $O(\log n)$.

6.2.4 The Interval Algorithm

Incorporating the changes described above yields a new interval branch and bound placement algorithm, as summarized in Figure 6.5.

Figure 6.6 illustrates this interval placement technique. The diagram depicts a branch and bound search tree for a circuit with three components. Table 6.1 gives the interval dimensions of these components. As before, each node represents a partial or complete placement and is labeled with a unique identifier and its associated cost.

Block	Length			Width		
	min	max	avg	min	max	avg
1	29.7	30.3	30.0	14.85	15.15	15.0
2	9.9	10.1	10.0	9.9	10.1	10.0
3	7.92	8.08	8.0	5.94	6.06	6.0

Table 6.1: Component Dimensions for the Search Tree Example

In this example, the costs are interval ranges. In Table 6.2, we show how the set of minimum-cost placements is updated during the search. For each placement visited, the table displays the resulting minimum-cost bound and the contents of the placement set. For each configuration in the optimal set, we also display the lower bound on its cost in parentheses. This value is used to prune placements from the set whenever the minimum cost changes. At the end of the search, the set of minimum-cost placements contains two configurations P25 and P12 and has a minimum cost in the range $[941, 979]$.

For our second example, we re-examine the ten-cell circuit in Figure 6.4. Instead of using only average values, we represent uncertain cell and channel sizes by their interval ranges. The interval dimensions of each block are displayed in Table 6.3. In the course of the search, the algorithm examined over 3 million partial and complete placements and returned a set of five optimal configurations for the circuit. The results of this computation are displayed in Figure 6.7. As shown, the best layouts had an area bound of $[1980.2, 2016.4]$ square mils. A closer examination of the set of minimum-cost placements reveals many similarities. For instance, elements B2, B3, B4 and B5 remain in the same relative position in all best placements. Other elements tend to cluster together, such as B8, B9 and B0. Some components appear in the same position in all five configurations. For example, block B1 is in the upper left-hand corner in all optimal solutions. These invariants indicate areas on the chip which can be improved to reduce the area of all optimal configurations. For example, blocks B2, B3, B4 and B5 all lie on critical paths in the horizontal channel position graphs of each configuration; therefore, these blocks and the channels between them define the width of the layout in all instances. By reducing the area of these blocks or by improving the routing between them, we can reduce the area of all five solutions.

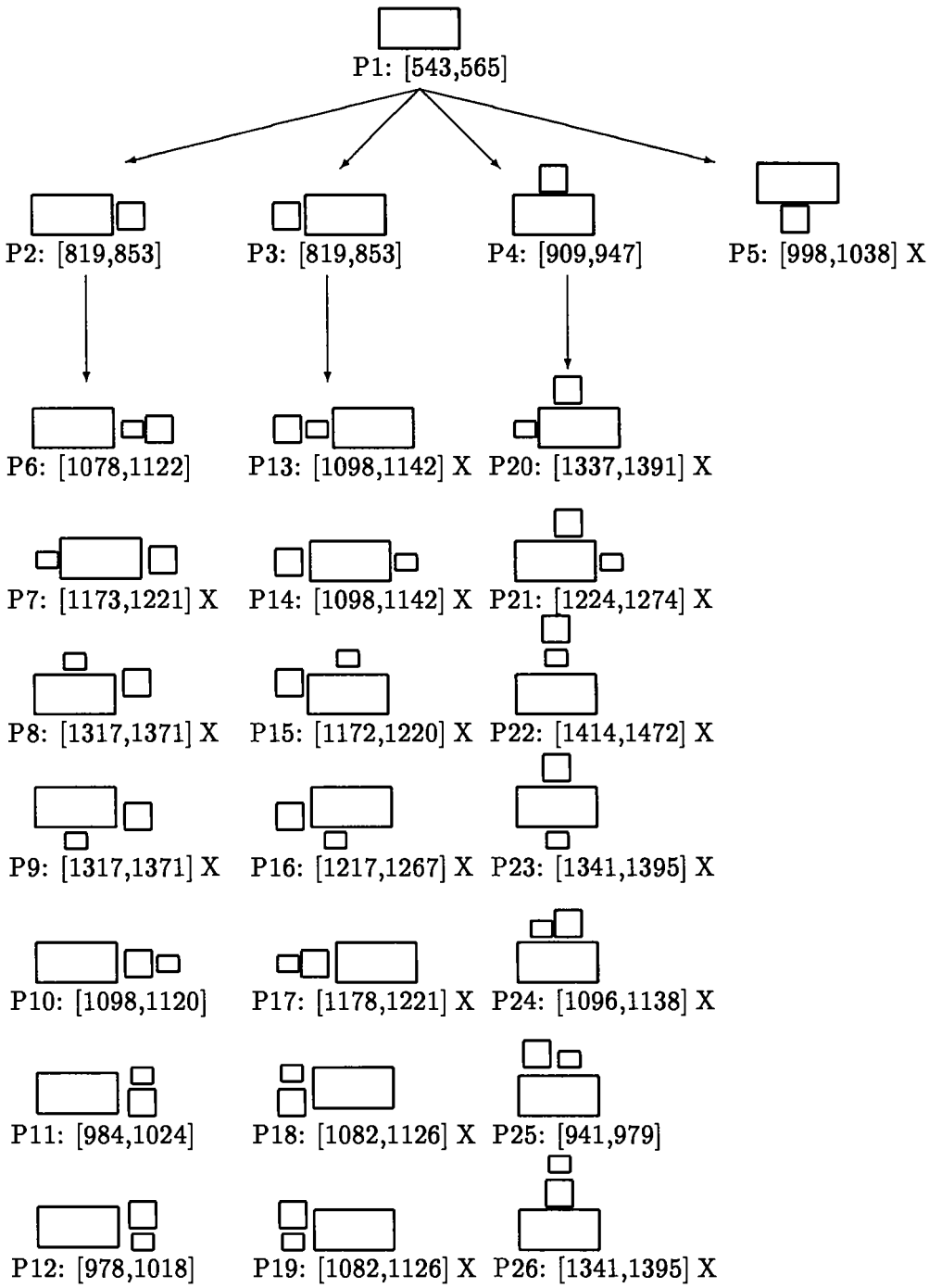
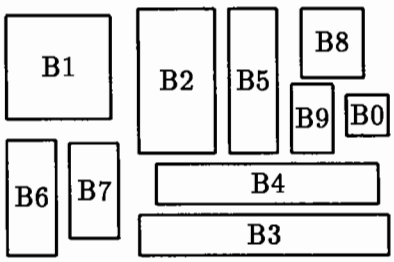


Figure 6.6: Interval Branch and Bound Search Tree for Placement

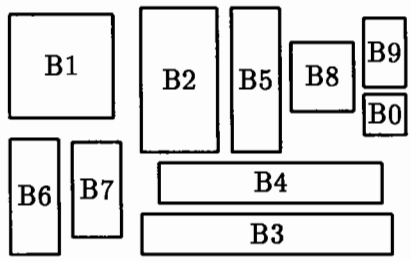
Visit			Minimum-Cost Results	
Placement	Cost	Type	Cost Bounds	Placement Set
P1	[543, 565]	partial	[1200, 1200]	
P2	[819, 853]	partial	[1200, 1200]	
P6	[1078, 1122]	complete	[1078, 1122]	$\{P6(1078)\}$
P7	[1173, 1221]	complete	[1078, 1122]	$\{P6(1078)\}$
P8	[1317, 1371]	complete	[1078, 1122]	$\{P6(1078)\}$
P9	[1317, 1371]	complete	[1078, 1122]	$\{P6(1078)\}$
P10	[1098, 1120]	complete	[1078, 1120]	$\{P10(1098), P6(1078)\}$
P11	[984, 1024]	complete	[984, 1024]	$\{P11(984)\}$
P12	[978, 1018]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P3	[819, 853]	partial	[978, 1018]	$\{P11(984), P12(978)\}$
P13	[1098, 1142]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P14	[1098, 1142]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P15	[1172, 1220]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P16	[1217, 1267]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P17	[1178, 1221]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P18	[1082, 1126]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P19	[1082, 1126]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P4	[909, 947]	partial	[978, 1018]	$\{P11(984), P12(978)\}$
P20	[1337, 1391]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P21	[1224, 1274]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P22	[1414, 1472]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P23	[1341, 1395]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P24	[1096, 1138]	complete	[978, 1018]	$\{P11(984), P12(978)\}$
P25	[941, 979]	complete	[941, 979]	$\{P12(978), P25(941)\}$
P26	[1341, 1395]	complete	[941, 979]	$\{P12(978), P25(941)\}$
P5	[998, 1038]	partial	[941, 979]	$\{P12(978), P25(941)\}$
Final			[941, 979]	$\{P12(978), P25(941)\}$

Table 6.2: Finding the Set of Minimum-Cost Placements

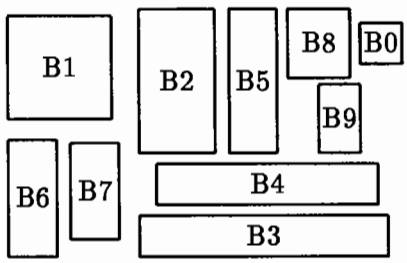
The Minimum-Cost Placements have cost bound = [1980.2, 2016.4]
The set of these placements follows:



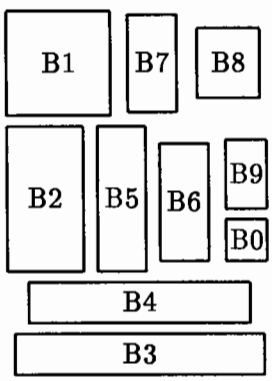
Cost = [1980.2, 2016.4]



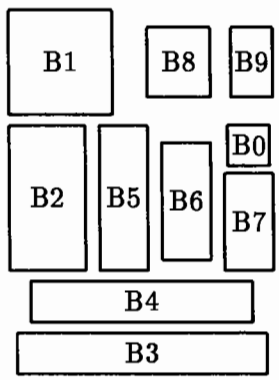
Cost = [1981.9, 2016.4]



Cost = [1982.2, 2016.4]



Cost = [1995.4, 2032.7]



Cost = [1995.6, 2032.7]

Figure 6.7: The Set of Minimum-Cost Placements for a Circuit with 10 Components

Block	Length			Width		
	min	max	avg	min	max	avg
B1	14.92	15.08	15.0	14.92	15.08	15.0
B2	10.94	11.06	11.0	20.89	21.11	21.0
B3	35.82	36.18	36.0	5.97	6.03	6.0
B4	31.84	32.16	32.0	5.97	6.03	6.0
B5	6.96	7.04	7.0	20.89	21.11	21.0
B6	6.96	7.04	7.0	16.91	17.09	17.0
B7	6.96	7.04	7.0	13.93	14.07	14.0
B8	8.95	9.05	9.0	9.95	10.05	10.0
B9	5.97	6.03	6.0	9.95	10.05	10.0
B0	5.97	6.03	6.0	5.97	6.03	6.0

Table 6.3: Block Dimensions for the 10-Component Circuit

Figure 6.8 presents another placement example. This circuit contains eight components with uncertain dimensions (See Table 6.4). Using expected values only, the original branch and bound search algorithm generated over 190,000 partial and complete placements and returned a single solution. The resulting configuration has a area of 4326.0 square mils and appears in the upper left-hand corner of Figure 6.8. Using the interval bounds on the component sizes, the interval branch and bound search algorithm generated almost 310,000 partial and complete placements in the search for the optimal solutions. The results of this search are shown in Figure 6.8. In this case, there are four optimal solutions with a minimum-cost bound of $[4285.4, 4366.8]$ square mils. As shown in the diagram, three of the four best configurations are very similar. In all four solutions, block B6 always appears to the right of block B1 and block B2 always appears in the upper left-hand corner of each layout.

6.3 Proof of the Method

Having presented our interval placement algorithm, we now prove that this method accurately computes bounds on minimum placement area. This proof is divided into two independent parts: computing the interval objective functions and comparing their values. In the first part, we must show that our algorithm for computing the interval objective function is correct. Since the algebra employed to evaluate objective functions is application-specific, this proof applies only to our particular cost function – circuit

The Minimum-Cost Placements have cost bound = [4285.4, 4366.8]
The set of these placements follows:

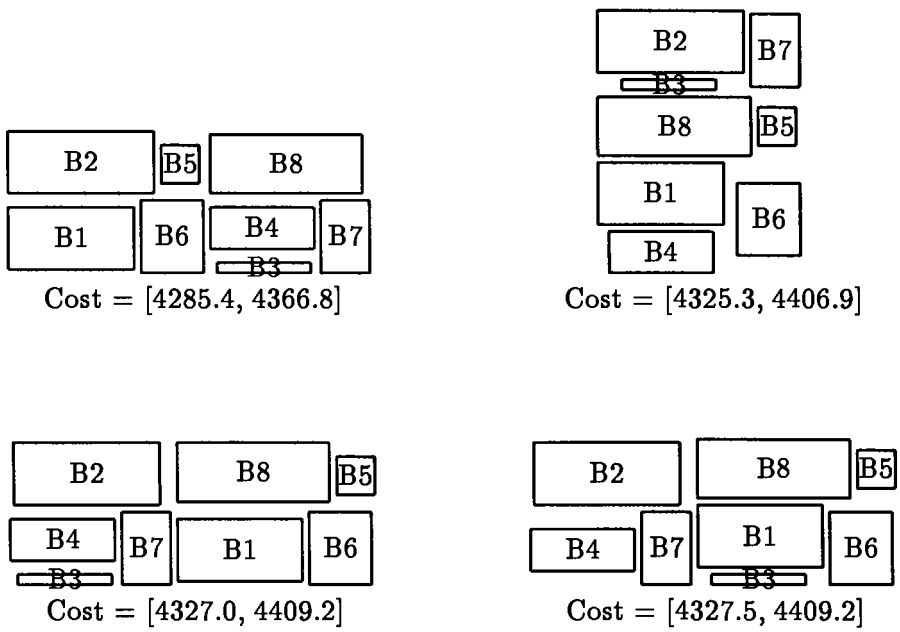


Figure 6.8: The Set of Minimum-Cost Placements for a Circuit with 8 Components

Block	Length			Width		
	min	max	avg	min	max	avg
B1	35.82	36.18	36.0	17.91	18.09	18.0
B2	41.79	42.21	42.0	17.91	18.09	18.0
B3	26.86	27.14	27.0	2.98	3.02	3.0
B4	29.85	30.15	30.0	11.94	12.06	12.0
B5	10.94	11.06	11.0	10.94	11.06	11.0
B6	17.91	18.09	18.0	20.89	21.11	21.0
B7	13.93	14.07	14.0	20.89	21.11	21.0
B8	43.78	44.22	44.0	16.91	17.09	17.0

Table 6.4: Block Dimensions for 8-Component Example

area. To compute the length and width of a given layout, our interval placement program employs an interval longest path algorithm. In Chapter 5, we presented a proof showing that this interval longest path algorithm correctly computes bounds on longest path length. Once the bounds on the dimensions of the layout have been determined, they are multiplied using the interval product operator ($\odot$) to calculate bounds on circuit area. The derivation of this interval product operator and its proof are presented by Moore in [Moor79]. From these proofs, we conclude that our interval objective function correctly computes bounds on circuit area.

Second, we must show that the interval algorithm for computing minimum-cost placements is also correct. Unlike the first part of the analysis, this proof applies to all interval objective functions and also to all interval-based placement strategies. The primary purpose of all placement algorithms is to find the placement that minimizes some cost metric. This reduces to the problem of choosing the smallest of a finite set of elements and may be solved using the algebra $(\Re, min, \infty)$, where $\Re$ also contains the element ∞. Since partial and complete placements may be generated and compared in any order, *min* must be associative and commutative. By definition, all real numbers are less than ∞; therefore, ∞ must be the identity element for *min*. Since the minimum value returned must be a member of the set, we further require that $\Re$ be closed over *min*. Clearly, this algebra has all of these properties.

Now let I be the set of intervals over the augmented set $\Re$. We must show that the extended algebra $(I, MIN, [\infty, \infty])$ exhibits the same properties as the non-interval original. In Appendix A, we demonstrate that the commutative, associative, closure,

and identity $[\infty, \infty]$ properties hold for MIN. Since the interval algebra is derived from the non-interval one and exhibits the same properties, we conclude that the algorithm employing this algebra will correctly compute the minimum of a collection of interval elements.

6.4 Performance

Both the interval branch and bound placement algorithm (Intplace) and the non-interval version were implemented in the C programming language and run on a SUN SPARCstation. In this section, we discuss and compare the performance of these two approaches.

6.4.1 Running Time

We begin by analyzing the operating speed of both implementations. Because it is an exhaustive search strategy, the original branch and bound algorithm may generate all possible configurations of the circuit components before finding the best solution; therefore, this algorithm has worst-case running time that is exponential in the number of circuit elements. Since the interval solution is based on the same approach, it too will have a exponential worst-case running time.

Intplace differs from the non-interval version in two main areas: the use of interval operations and the storage of minimum-cost solutions. First, Intplace employs interval operations to perform its calculations and comparisons. For each addition, multiplication, minimum, and maximum operation performed in the original version, we perform two such operations in the interval version. This is particularly evident in the longest path computation used to calculate placement cost; hence, we expect that this function will require twice the time in the interval implementation.

As noted in the previous chapter, the interval longest path algorithm also requires additional time to save the longest paths. Our placement cost calculations, however, need only the longest path lengths, not the paths themselves. By not saving this extraneous path information, we increase the efficiency of the layout cost computations for both interval and non-interval placement algorithms. Without this additional bookkeeping, the longest-path algorithms employed in both placement programs differ only in their choice of interval vs. non-interval operators.

Computing the minimum-cost solution is the second major difference between the interval and non-interval placement algorithms. Instead of returning a single configuration, Intplace returns a set of potential minimum-cost placements. Maintaining this set adds to the operating speed of the interval algorithm. Specifically, whenever the minimum-cost bound is updated in the course of the search, we must insert and delete items from this set. Using a heap to implement the set requires $O(\log n)$ time for each insertion and deletion, where n is the number of placements in the set. The total number of possible complete configurations is exponential in m, the number of circuit elements. In the worst-case, the set of minimum-cost placements could contain all of these possible configurations, yielding insertion and deletion times of $O(m)$. This decreases the operating speed of the interval algorithm by a factor of m in the worst-case. Fortunately, the list of minimum-cost placements is typically small, containing a fraction of the total number of possible complete placements. For example, consider the circuit in Figure 6.7. Of the several billion possible complete placements in the unpruned search tree, the optimal solution contained only five configurations.

We also tested both the interval and non-interval implementations on identical circuit descriptions and recorded their response times. The circuits tested contained up to ten components. For the larger circuits, the programs generated more than 180,000 distinct partial and complete configurations in the search for the best solution. Even so, this figure represents a significantly pruned portion of the almost 98 billion possible configurations in the search space for that problem. In spite of the large numbers of configurations explored, these experiments showed no appreciable difference in the operating speeds of the two algorithms. The apparent similarity in running times is attributed to the observation that most of the computation time is spent generating child placements, rather than evaluating them. The routines for generating child configurations are identical in both implementations.

6.4.2 Interval Width

Since an interval cost is used to prune the search tree in the branch and bound placement algorithm, the width of these intervals will effect the efficiency of the search. Recall that the algorithm prunes a partial placement from the tree when its cost exceeds the current minimum cost. In the interval version, placement cost is greater than minimum cost if and only if the lower bound on placement cost is greater than the

upper bound on minimum cost. Obviously, wider intervals yield larger upper bounds. With a larger upper bound on minimum layout cost, we prune fewer nodes from the search tree and increase the time needed to find the optimal solutions. Even in the non-interval algorithm, pruning efficiency is heavily dependent on circuit structure. In circuits with many blocks of similar shape and size, the standard algorithm may not be able to prune many nodes from the search tree, resulting in a nearly exhaustive enumeration of placement alternatives. For circuit descriptions containing blocks with widely varying sizes, the algorithm can prune a larger portion of the nodes from the search tree. Because pruning efficiency also depends on relative block size, a precise formula indicating the relationship between interval width and pruning efficiency is difficult to determine. We can, however, present several examples illustrating this correlation.

Figure 6.9 depicts the effects of interval width on search efficiency for several circuit descriptions containing five elements. In this graph, interval width is measured by the percent variance from the center of the interval (i.e., interval = center $\pm$ center $\cdot$ variance). Search efficiency is measured by the percentage of the 6565 possible partial and complete configurations generated and examined by the placement algorithm. In this experiment, we started with degenerate intervals and counted the number of partial and complete placements examined in the course of the search. For each circuit description, we varied the width of the intervals by a given percentage and recorded the resulting number of placements examined in locating the optimal solution. The graph displays these results for three different circuit descriptions. The first circuit (denoted by dots with dashed edges) contains blocks with a wide variation of sizes, the second circuit (denoted by triangles and solid lines) contains five blocks with three different sizes, and the third (denoted by diamonds and dotted edges) contains five identical blocks. As the graph shows, the increase in the number of configurations produced was modest for small cost interval widths ($\pm$5%). For large interval widths ($\pm$25%), the algorithm degenerated to an exhaustive enumeration of placement alternatives. Therefore, this algorithm should not be used, if the intervals given for circuit parameters are too wide.

As expected, interval width also affects the size of the optimal placement set. Wider intervals yield wider minimum cost bounds. With a larger upper bound on minimum layout cost, we prune fewer configurations from the optimal solution set. Figure 6.10 presents a graph illustrating the effects of interval width on the size of the optimal placement set for our three circuit examples. In each case, we varied the width of the

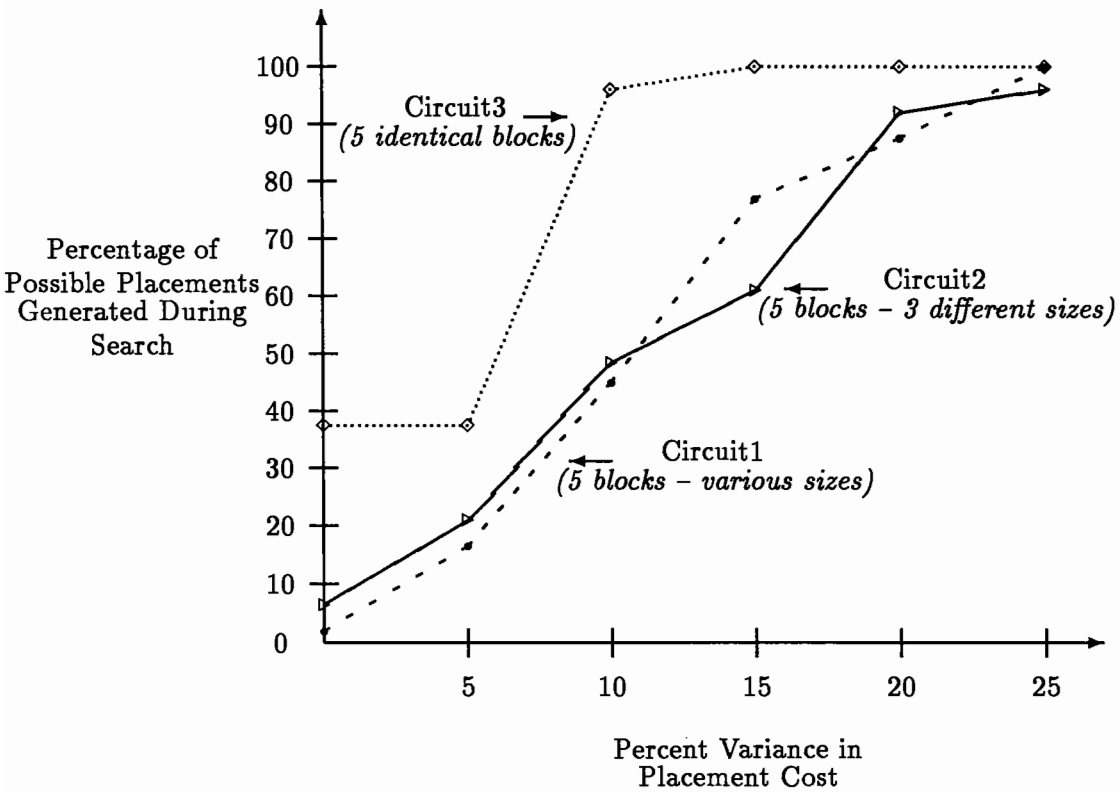


Figure 6.9: Cost Interval Width vs. Pruning Efficiency

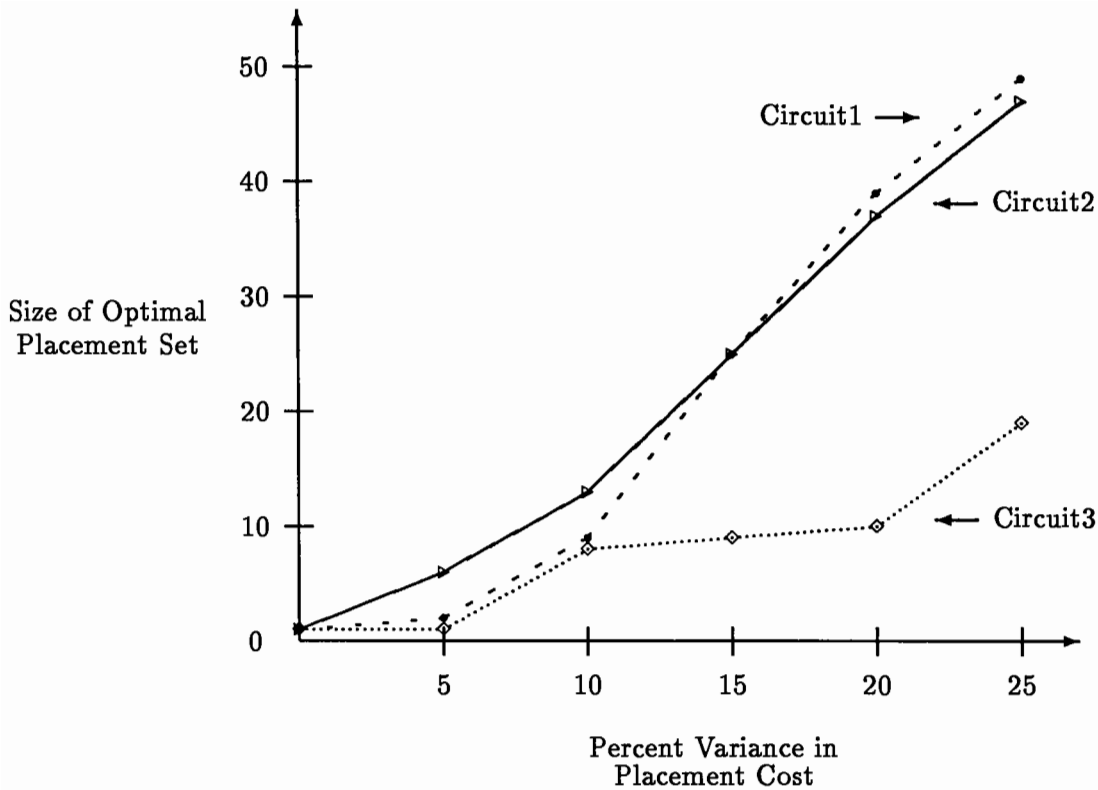


Figure 6.10: Cost Interval Width vs. Optimal Placement Set Size

cost intervals by the given percentage and recorded the number of placements in the final solution set. As shown in the graph, the solution sets for small cost interval widths ($\pm 5\%$) contain few configurations. For larger cost intervals ($\pm 25\%$), the solution sets become too large to be useful. Not surprisingly, the less we know about the circuit, the less we can conclude about the optimal layout.

6.5 Summary

In this chapter, we discussed the problem of uncertain cost functions in placement algorithms. Using several common objective functions, we demonstrated the prevalence of uncertain parameters in placement cost metrics. In spite of this, current placement algorithms use only single-valued estimates in their objective functions. We argued that using the full range of possible values when computing placement costs yields

a more accurate prediction of placement quality, reveals more about the structure of the optimal configurations, and facilitates iterative improvement. For these reasons, we proposed an interval-based approach for modeling uncertainty in placement algorithms. Given the interval range for each uncertain parameter in an objective function, the interval placement algorithm computes and returns bounds on minimum placement cost and a set of placements, each of which is optimal for some combination of input values.

We illustrated this approach by implementing an interval branch and bound placement algorithm, Intplace. We showed how interval operators could be used to compute the interval cost of a placement and to determine the minimum-cost configuration. Although a branch and bound algorithm was chosen, the general principles employed also apply to other placement methods.

To test the performance of our interval approach, we implemented interval and non-interval versions of the placement algorithm. Analysis suggests that the interval program may be slower than the non-interval one by a factor of m, the number of circuit components, in the worst-case. Experiments on several circuits, however, showed no appreciable difference in the operating speeds of the two implementations. We also observed that the width of the intervals affects the efficiency of the search, since wider placement cost bounds reduce the number of placements that are pruned from the search tree. Wider intervals also increase the size of the solution set, as fewer placements can be pruned from the optimal set.

Chapter 7

Conclusions

As the examples in the previous chapters demonstrate, numerical uncertainty is prevalent in VLSI design. Delayed design decisions produce uncertain transistor strengths in switch-level simulation, exploring layout alternatives results in uncertain cell sizes in hierarchical placement, and manufacturing disturbances lead to uncertain electrical characteristics in RC timing analysis. For these and other reasons, the values of many variables used in VLSI analysis cannot be known exactly. In spite of this, most current CAD tools ignore parameter uncertainty and employ only "expected" values in their calculations. Systems that model uncertainty offer several advantages over these conventional approaches. Among their benefits, they provide a more complete characterization of circuit behavior and facilitate iterative improvement. We therefore proposed a new methodology based on interval arithmetic for modeling numerical uncertainty in VLSI design. To illustrate our approach, we applied our technique to several applications, including switch-level simulation, timing analysis, and placement. These applications reveal the strengths and weaknesses of our interval paradigm.

7.1 Advantages of the Method

One strength of the interval method is its simplicity. To create an interval algorithm for each application, we began by deriving the interval extensions of the discrete operators used in the conventional implementation. The basic arithmetic operations (i.e., addition, subtraction, multiplication, and division) were taken from a field of mathematics known as interval arithmetic. Other operations were derived as needed using Equation 1.1. Once derived, the interval operators were incorporated within the algorithm

to create an interval version of it. In two instances, the longest path and placement algorithms, further changes were required. When the longest path algorithm is used for timing verification, it is not sufficient to return bounds on longest path length and a *single* representative longest path. To efficiently reduce circuit delay, we need the *set* of all potential longest paths. Likewise, the interval placement algorithm must return the set of potentially optimal configurations, in addition to computing bounds on optimal placement quality. Storing these extra paths and configurations required further modifications to the algorithms. Even so, the required additions were simple and intuitive.

The second advantage of the interval approach is its generality. As demonstrated, these techniques can be easily incorporated within many kinds of VLSI CAD tools. They can also be used to model many types of uncertainty, including some that cannot be statistically modeled. Uncertain logic levels and variation due to exploration of design alternatives are two such examples. When an uncertain value is given as a mean value and standard deviation, the 1σ, 2σ, or 3σ endpoints can be used as interval bounds.

Another advantage is speed. For many of our applications, the running time of the interval algorithm is greater than that of its non-interval counterpart by only a small constant. Two exceptions to this rule are the longest path and placement algorithms which return path or layout information, in addition to computing bounds. The longest path algorithm has a worst-case complexity $\log n$ times greater than that of the non-interval version, where n is the number of vertices in the graph. Similarly, the interval placement algorithm is slower by a factor of $2m$ in the worst-case, where m is the number of circuit components. In all applications, experiments comparing the conventional and interval implementations show little difference in operating speed. For RC timing analysis, we also created a Monte Carlo algorithm for comparison. Our tests demonstrate that the interval algorithm runs several orders of magnitude faster than the Monte Carlo version for the same quality results.

7.2 Limitations of the Method

While it has many strengths, the interval paradigm is not without its limitations. As the RC timing example shows, this technique may generate overly conservative bounds for some applications. This is due to an inherent weakness in the algebra: a lack

of additive and multiplicative inverses. For those applications which do not require inverses, the interval algorithms return tight and accurate bounds on the solution. This is true for placement, switch-level simulation, and the longest path computations. For functions that do contain inverses, tight bounds are not guaranteed.

For these cases, we presented two interval-based alternatives for computing better bounds. These methods achieve accuracy at the cost of some extra computational effort. Even so, both methods are significantly faster than Monte Carlo techniques. Like the original interval solution, these approaches are general and can be used on any real-valued arithmetic function $F(x_1, ..., x_n)$.

The first alternative is the centred method presented in Chapter 4. Using this technique, we evaluate the given function F on one particular assignment of values, $c_1, ..., c_n$, (usually the midpoints of all intervals) and create an interval representation of F that produces symmetric bounds centered at $F(c_1, ..., c_n)$ containing the theoretical solution. While there are many types of centred forms, we chose Krawczyk's centred form for its flexibility. This algorithm has a time complexity of $O(sn)$, where s represents the number of arithmetic operations in the function and n represents the number of independent variables.

In Chapter 4, we also presented a bounding method based on interval differential calculus. With this approach, we use derivatives to check the monotonicity of function F with respect to the ranges of each variable. If the function is monotonic in all variables, we can exactly compute bounds on F. If the function is non-monotonic, we subdivide the variable ranges and repeat the analysis on the resulting subintervals. This algorithm has a worst-case time complexity of $O(2^{b+1}sn)$, where b is the number of interval bisections allowed, s is the number of arithmetic operations in F, and n is the number of independent variables. Although it is the slowest of the interval methods presented, it is the only one that provides a measure of the goodness of its bounds.

A second limitation of bounding approaches is that they return performance limits without any indication of the probability of their occurrence. If the probability of these worst-case conditions is small, then these results are again overly conservative. Such results may prompt the design engineer to needlessly overdesign the chip in an effort to prevent events that are unlikely to occur. To avoid this, 1σ or 2σ endpoints may be used in the computations, if 3σ bounds are too pessimistic.

7.3 Future Work

In this thesis, we presented several examples to illustrate the usefulness of our interval approach. These examples are but a sampling of the potential CAD applications for this methodology. Among the many other possible applications are circuit simulation and timing-driven placement.

The necessity of considering the effects of manufacturing disturbances in circuit simulation has long been recognized [Nass84]. As previously mentioned, expensive Monte Carlo techniques are frequently applied in an effort to model the effects of fabrication variations on circuit behavior. Many papers have been written presenting statistical methods for finding correlations between parameters for use in both statistical and worst-case analysis [Dive84]. Because of the computational expense involved in circuit simulation, Monte Carlo techniques are particularly time-consuming. If developed, an accurate interval circuit simulator could produce bounds in much less time.

Another possible application is timing-driven placement. Here, timing considerations are used when computing the layout of a circuit. Configurations containing long wires with large delays can severely degrade the performance of the chip. Timing-driven placement algorithms use information about the critical paths through a circuit to generate a layout that minimizes the wire delays along this path. As we have already shown, the component and wire delays employed in this computation are subject to variation, so interval longest path algorithms can be applied here to locate critical paths through circuits with uncertain component and wire delays. Once computed, these critical paths can be used in several ways to direct the placement. One solution [Jack86] gives higher weights to nets along the critical path and employs a min-cut partitioning scheme to assign circuit components to specific regions on the layout. Since the delays used to determine the weights are uncertain, we would need to create an algorithm for min-cut partitioning on a graph with interval-weighted edges.

In addition to timing-dependent methods, there are many other objective functions for placement that contain uncertain values. In Chapter 6, we cited several examples. While we implemented one possible interval objective function, there are many others to consider.

In addition to exploring other applications for our interval methods, another area for future research is statistical algebras. The goal here is to derive a simple set of

rules for performing arithmetic operations on (presumably Gaussian) statistical distributions. If such algebras were to be developed, they could be incorporated into existing algorithms in the same manner as intervals. The algorithms employing these algebras would then return the probability distribution of the results. This methodology would complement our interval approaches, providing a fast method for generating probability distributions, when worst- and best-case bounds alone are insufficient. With the exception of Hitchcock's rule for adding statistical distributions [Hitc82], such algebraic operators have yet to be developed.

7.4 Closing Remarks

In conclusion, there are some situations for which statistical measures are more appropriate, but bounding methods are simpler and faster. We therefore advocate using our interval methods first to produce preliminary estimates of behavior. Then, if further information is required, the computationally-expensive statistical methods, such as Monte Carlo simulation, should be employed. In situations where statistical methods are not applicable, our interval approach offers a simple, fast, and provably correct solution.

Appendix A

Properties of the Operators

In this appendix, we prove that the associative, commutative, identity, and closure properties hold for the interval minimum (MIN) and maximum (MAX) operators. Let I^+ be the set of intervals over the augmented set of real numbers, $\Re \cup \{\infty\} \cup \{-\infty\}$. Details are given for the MIN operator only. The proofs for the MAX operator are analogous.

Proof.

1. MIN is associative

   For all $[a,b],[c,d],[e,f] \in I^+$, we must show that

   $$MIN(MIN([a,b],[c,d]),[e,f]) = MIN([a,b],MIN([c,d],[e,f])).$$

   By the definition of MIN, we know that

   $$\begin{aligned} MIN(MIN([a,b],[c,d]),[e,f]) &= [min(min(a,c),e), min(min(b,d),f)] \\ MIN([a,b],MIN([c,d],[e,f])) &= [min(a,min(c,e)), min(b,min(d,f))]. \end{aligned}$$

   Because min is associative,

   $$\begin{aligned} min(min(a,c),e) &= min(a,min(c,e)) \; and \\ min(min(b,d),f) &= min(b,min(d,f)). \end{aligned}$$

   Hence, MIN is associative. □

2. MIN is commutative

   For all $[a,b],[c,d] \in I^+$, we must show $MIN([a,b],[c,d]) = MIN([c,d],[a,b])$.

Expanding the expression using the definition of MIN yields

$$\begin{aligned} MIN([a,b],[c,d]) &= [min(a,c), min(b,d)] \text{ and} \\ MIN([c,d],[a,b]) &= [min(c,a), min(d,b)]. \end{aligned}$$

Since min is commutative, these expressions are equivalent. □

3. MIN has identity element $[\infty, \infty]$
   For all $[a,b] \in I^+$, we must show that $MIN([a,b],[\infty,\infty]) = [a,b]$. This follows directly from the definition of MIN and that ∞ is the identity of min. □

4. I^+ is closed with respect to MIN
   For all $[a,b],[c,d] \in I^+$, we must show that $MIN([a,b],[c,d]) \in I^+$. Applying the definition of MIN, $MIN([a,b],[c,d]) = [min(a,c), min(b,d)]$. Since augmented $\Re$ is closed with respect to min, then $min(a,c)$ and $min(b,d) \in \Re \cup \{\infty\} \cup \{-\infty\}$. Now all that remains is to show that $min(a,c) \leq min(b,d)$. Clearly this is true, since by definition $[a,b]$ implies $a \leq b$ and $[c,d]$ implies $c \leq d$; therefore, $[min(a,c), min(b,d)] \in I^+$. □

Appendix B

Common Interval Operations

The following is the C language code for the common interval operations employed in this thesis. They are interval addition (ADD), subtraction (SUB), multiplication (MULT), division (DIV), minimum (MIN), and maximum (MAX). All assume the following definition of an *interval*:

```
typedef struct
  {
  float min;          /* the lower bound of the range */
  float max;          /* the upper bound of the range */
  }  INTERVAL;
```

B.1 Addition

Given two intervals over the real numbers, this routine performs interval addition on these ranges and returns their sum.

```
INTERVAL ADD(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min + range2.min;
  result.max = range1.max + range2.max;
  return(result);
  }
```

B.2 Subtraction

Given two intervals over the real numbers, this routine performs interval subtraction on these ranges and returns their difference.

```
INTERVAL SUB(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min - range2.max;
  result.max = range1.max - range2.min;
  return(result);
  }
```

B.3 Multiplication

Given two intervals over the positive real numbers, this routine performs interval multiplication on these ranges and returns their product.

```
INTERVAL MULT(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min * range2.min;
  result.max = range1.max * range2.max;
  return(result);
  }
```

B.4 Division

Given two intervals over the positive real numbers, this routine performs interval division on these ranges and returns their quotient.

```
INTERVAL DIV(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = range1.min / range2.max;
  result.max = range1.max / range2.min;
  return(result);
  }
```

B.5 Minimum

Given two intervals over the real numbers, this routine computes and returns their interval minimum.

```
INTERVAL MIN(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = (range1.min < range2.min) ? range1.min : range2.min;
  result.max = (range1.max < range2.max) ? range1.max : range2.max;
  return(result);
  }
```

B.6 Maximum

Given two intervals over the real numbers, this routine computes and returns their interval maximum.

```
INTERVAL MAX(range1, range2)
  INTERVAL range1;
  INTERVAL range2;
  {
  INTERVAL result;

  result.min = (range1.min > range2.min) ? range1.min : range2.min;
  result.max = (range1.max > range2.max) ? range1.max : range2.max;
  return(result);
  }
```

Appendix C

Interval Differential Operations

In this appendix, we present the C language code for the interval differential operations defined in Chapter 4. These operations use the interval algebraic operators ADD, SUB, MULT, and DIV to perform their calculations. They also employ the following definitions of an *interval* and an *interval operand*:

```
typedef struct
  {
  float min;          /* the lower bound of the range */
  float max;          /* the upper bound of the range */
  }  INTERVAL;

typedef struct
  {
  INTERVAL value;     /* the value of the operand */
  INTERVAL *deriv;    /* the vector of partial derivatives */
  }  IOPERAND;
```

C.1 Addition

Given two interval operands, this function computes and returns the interval sum of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DADD (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL ADD();

                                          /* compute the interval sum */
  result.value = ADD(o1.value, o2.value);
                                           /* create derivative array */
  create_deriv_array(&(result.deriv));
                                        /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = ADD(o1.deriv[i], o2.deriv[i]);
                                                  /* return the result */
  return(result);
  }
```

C.2 Subtraction

Given two interval operands, this function computes and returns the interval difference of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DSUB (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL SUB();

                                    /* compute the interval difference */
  result.value = SUB(o1.value, o2.value);
                                              /* create derivative array */
  create_deriv_array(&(result.deriv));
                                          /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = SUB(o1.deriv[i], o2.deriv[i]);
                                                    /* return the result */
  return(result);
  }
```

C.3 Multiplication

Given two interval operands, this function computes and returns the interval product of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DMULT (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL ADD(), MULT();

                                        /* compute the interval product */
  result.value = MULT(o1.value, o2.value);
                                             /* create derivative array */
  create_deriv_array(&(result.deriv));
                                         /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = ADD(MULT(o1.value, o2.deriv[i]),
                          MULT(o2.value, o1.deriv[i]));
                                                /* return the result */
  return(result);
  }
```

C.4 Division

Given two interval operands, this function computes and returns the interval quotient of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DDIV (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL SUB(), MULT(), DIV();

                                        /* compute the interval quotient */
  result.value = DIV(o1.value, o2.value);
                                              /* create derivative array */
  create_deriv_array(&(result.deriv));
                                          /* compute partial derivatives */
  for (i = 0; i < numvariables; ++i)
    result.deriv[i] = DIV(SUB(o1.deriv[i],
                              MULT(result.value, o2.deriv[i])),
                          o2.value);
                                                  /* return the result */
  return(result);
  }
```

Bibliography

[Aho74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[Aho83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[Breu77] Melvin A. Breuer. A class of min-cut placement algorithms. In *14th IEEE Design Automation Conference*, pages 284–290, 1977.

[Brya80] Randal E. Bryant. An algorithm for MOS logic simulation. *Lambda*, 1(3):46–53, 1980.

[Brya84] Randal E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, pages 160–177, February 1984.

[Byrd85] Richard H. Byrd, Gary D. Hachtel, Micheal R. Lightner, and Michel H. Heydemann. Switch-level simulation: Models, theory, and algorithms. In A. Sangiovanni-Vincentelli, editor, *Computer Aided-Design of VLSI Circuits and Systems*, pages 93–148. JAI Press, Greenwich, CT, 1985.

[Chow88] Paul Chow and Mark Horowitz. The design and testing of MIPS-X. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 95–114. MIT Press, 1988.

[Cies84] Maciej J. Ciesielski and Edwin Kinnen. Digraph relaxation for CAD layouts of cell based integrated circuits. In *1983-84 Computer Science and Computer Engineering Research Review.* University of Rochester, 1983-84.

[Dai89] Wayne Wei-Ming Dai, Bernhard Eschermann, Ernest S. Kuh, and Massoud Pedram. Hierarchical placement and floorplanning in BEAR. *IEEE Transactions on Computer-Aided Design*, 8(12):1335–1349, December 1989.

[Dive84] Dileep A. Divekar. DC statistical circuit analysis for bipolar IC's using parameter correlations – an experimental example. *IEEE Transactions on Computer-Aided Design*, 3(1):101–103, January 1984.

[Fidu82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th IEEE Design Automation Conference*, pages 175–181, 1982.

[Gecs87] Jan Gecsei and Eduard Cerny. Self-adjusting networks for VLSI simulation. *IEEE Transactions on Computers*, 36(9):1114–1120, September 1987.

[Hajj87] Ibraham Hajj and Daniel Saab. Switch-level logic simulation of digital bipolar circuits. *IEEE Transactions on Computer-Aided Design*, 6(2):251–258, March 1987.

[Hard88] Bill Harding. New analysis techniques pave the way for analog simulation. *Computer Design*, pages 37–41, September 1988.

[Haye86] John P. Hayes. Uncertainty, energy, and multiple-valued logics. *IEEE Transactions on Computers*, 35(2):107–114, February 1986.

[Hitc82] Robert B. Hitchcock Sr. Timing verification and the timing analysis program. In *19th IEEE Design Automation Conference*, pages 594–603, 1982.

[Hugh89] Richard Hughey and Daniel P. Lopresti. The B-SYS programmable systolic array. Technical Report CS-89-32, Department of Computer Science, Brown University, 1989.

[Jack86] Micheal Jackson and Ernest Kuh. Performance-driven placement of cell based IC's. In *26th IEEE Design Automation Conference*, pages 370–375, 1986.

[Joup83] Norman P. Jouppi. Timing analysis for nMOS VLSI. In *20th IEEE Design Automation Conference*, pages 411–418, 1983.

[Kern70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[Kirk83] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Kuli81] Ulrich Kulisch and Willard Miranker. *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[McWi80] Thomas M. McWilliams. Verification of timing constraints on large digital systems. In *17th IEEE Design Automation Conference*, pages 139–147, 1980.

[Moor79] Ramon E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.

[MS89] Malgorzata Marek-Sadowska and Shen P. Lin. Timing driven placement. In *IEEE International Conference on Computer-Aided Design*, pages 94–97, November 1989.

[Nass84] Sani R. Nassif, Andrzej J. Strojwas, and Stephen W. Director. Fabrics II: A statistically based IC fabrication process simulator. *IEEE Transactions on Computer-Aided Design*, 3(1):40–46, January 1984.

[Oust83] John K. Ousterhout. Crystal: A timing analyzer for nMOS VLSI circuits. In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 57–70. Computer Science Press, 1983.

[Oust84] John K. Ousterhout. Switch-level delay models for digital MOS VLSI. In *21st IEEE Design Automation Conference*, 1984.

[Oust86] John K. Ousterhout. Using crystal for timing analysis. Technical Report UCB/CSD 86/272, University of California at Berkeley, 1986.

[Prea79] B.T. Preas and W.M. van Cleemput. Placement algorithms for arbitrarily shaped blocks. In *16th IEEE Design Automation Conference*, pages 474–480, 1979.

[Prea86] Bryan T. Preas and Patrick G. Karger. Automatic placement: A review of current techniques. In *23rd IEEE Design Automation Conference*, pages 622–629, 1986.

[Rall86] L.B. Rall. Improved interval bounds for ranges of functions. In K. Nickel, editor, *Interval Mathematics 1985*, pages 143–155. Springer-Verlag, Berlin, 1986.

[Rama83a] Vijaya Ramachandran. An improved switch-level simulator for MOS circuits. In *20th IEEE Design Automation Conference*, pages 293–299, 1983.

[Rama83b] Vijaya Ramachandran. *Studies in VLSI Layout and Simulation.* PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, August 1983.

[Rats84] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions.* Ellis Horwood Limited, Chichester, 1984.

[Rubi83] Jorge Rubinstein, Paul Penfield Jr., and Mark A. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on Computer-Aided Design*, 2(3):202–211, July 1983.

[Sobo75] I.M. Sobol. *The Monte Carlo Method.* Mir Publishers, Moscow, 1975.

[Sund87] Rolf Sundblad and Christer Svensson. Fully dynamic simulation of CMOS circuits. *IEEE Transactions on Computer-Aided Design*, 6(2):282–289, March 1987.

[Tarj83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[Tric86] Micheal T. Trick and Stephen W. Director. LASSIE: Structure to layout for behavioral synthesis tools. In *26th IEEE Design Automation Conference*, pages 104–109, 1986.

[Wall86] David E. Wallace and Carlo H. Sequin. Plug-in timing models for an abstract timing verifier. In *23rd IEEE Design Automation Conference*, pages 683–689, 1986.

[Webs84] *Webster's II New Riverside Dictionary.* Berkeley Publishing Group, New York, 1984.

[Zuko86] Charles A. Zukowski. *The Bounding Approach to VLSI Circuit Simulation.* Kluwer Academic Publishers, Boston, 1986.