Symbolic Functional and Timing Verification of Transistor-Level Circuits

Clayton B. McDonald and Randal E. Bryant *
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We introduce a new method of verifying the timing of custom CMOS circuits. Due to the exponential number of patterns required, traditional simulation methods are unable to exhaustively verify a medium-sized modern logic block. Static analysis can handle much larger circuits but is not robust with respect to variations from standard circuit structures. Our approach applies symbolic simulation to analyze a circuit over all input combinations without these limitations. We present a prototype simulator (SirSim) and experimental results. We also discuss using SirSim to verify an industrial design which previously required a special-purpose verification methodology.

1 Introduction

Custom circuits that have been hand-optimized at the transistor level appear primarily in microprocessors and other applications where performance is absolutely crucial and volumes are high enough to justify the additional design effort. They also appear in many other applications where small amounts of hand-optimization can go a long way toward meeting performance, power, or area goals. Because these circuits are often found in the critical timing paths of designs, the verification of their timing behavior is extremely important.

The timing verification of custom circuits is a difficult task due to their inherent variability, and no automatic methods are known for verifying them except exhaustive simulation. Unfortunately, a moderate-sized modern CMOS logic block can easily have over 100 inputs, requiring $> 2^{100}$ simulation patterns and making exhaustive simulation infeasible. Thus, designers must either generate directed simulation patterns by hand or rely on static timing analysis [9].

However, static timing analyzers are inherently unsuited to the analysis of highly customized circuits because of their reliance on heuristics to understand the timing requirements implied by certain circuit topologies. The analyzer must be able to recognize, for example, all possible latch and flip-flop design styles and perform any setup and hold-time checks required for proper operation. Small variations in a standard latch design, which often occur in custom circuitry, can easily defeat pattern recognition heuristics and lead to an incorrect analysis. Structures such as domino or self-timed logic also require recognition and constraint application and can vary greatly in implementation style. In a full-custom design environment where designers are free to develop innovative new circuits, maintaining reliable sets of heuristics is virtually impossible, resulting in error-prone verification and wasted engineering effort.

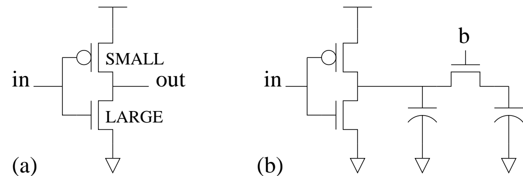


Figure 1: Data-dependent Delay Examples

To address this problem, we are proposing a new methodology called Symbolic Timing Simulation, which can be used to verify the hand-optimized portions of CMOS designs. This approach utilizes a symbolic simulator with an explicit notion of time. Seger and Bryant [13] considered the problem of incorporating gate delays into a symbolic simulator, and Devadas et.al. [7] used a gate-level symbolic simulator to compute the Transition Delay of a combinational circuit. However, neither group attempted to handle continuous data-dependent delays. Section 4.1 discusses the computation of delays using Multi-Terminal Binary Decision Diagrams (MTBDDs) [2], and section 3.2 presents a novel simulation algorithm capable of exploiting this information.

This approach should not be confused with other techniques based on MTBDDs (also known as ADDs). Bahar et. al. [1] proposed a variant of static analysis that represents node arrival times with MTBDDs to automatically eliminate false paths. Our approach, being simulation-based, does not explicitly compute node arrival times, and the MTBDDs are temporary data structures that are freed after each event processing step.

2 Symbolic Timing Simulation

2.1 Description

Symbolic simulation (without timing information) [5] was developed for the functional verification of digital designs. It performs an input-pattern-independent simulation by stimulating the circuit with a vector of Boolean variables instead of constant 0's and 1's. The values of the circuit nodes then become Boolean functions of the stimulus variables.

Once we introduce the concept of time into a symbolic simulator, we immediately discover the problem of data-dependent

*This research was supported by the SRC (contract DC-068), Intel Corporation, and Motorola.

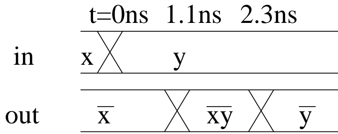


Figure 2: Skewed Inverter Timing Diagram

delays. When an input to a logic gate changes from one Boolean function to another, the delay through the gate may be dependent on the particular values the input is switching between. A simple example is the CMOS inverter in Figure 1(a) having a much larger pulldown device than pullup device. Delay can also be dependent on the values of stable signals in the gate's environment, such as the example in Figure 1(b). Here, the load seen by the inverter (and thus its delay) is dependent on the state of signal *b*.

To address this problem, we first recognize that for any given input pattern, a node transition will occur at some calculable point in time. Thus, the value of that node at any point is well-defined, and it can be represented by a Boolean function of the input variables. This means that the output will progress through a series of node functions. Consider again the skewed inverter example, and what happens when its input changes from function x to function y (Figure 2). If the $in \uparrow out \downarrow$ delay is 1.1ns, and the $in \downarrow out \uparrow$ delay is 2.3 ns, we obtain the series of 3 node functions shown. Initially, the output function is $\overline{x}$, and eventually it settles to $\overline{y}$. Since a falling transition will occur at the first timepoint and a rising transition will not occur until the second timepoint, the only way that *out* will be high in between is if both x and y were 0 and the output actually remained high continuously. This behavior is captured in the function $\overline{xy}$. In general, the output node function will progress from being dependent only on the old input variables to being dependent on the new.

2.2 Methodology

To verify a block containing arbitrary circuit structures, we simply perform a symbolic timing simulation while monitoring the Boolean functions on the output nodes. A specification of the correct output function must be supplied by the user or extracted from the RTL description of the block. For datapath circuits, which often contain highly customized circuits, the output function can usually be expressed very easily. If the output nodes settle to the proper functions, while being simulated under a realistic delay model, then the timing (and functional) correctness of the circuit under all input patterns is implied.

If the initial values of particular latch-nodes are required to express the expected output function, then the user must initialize them to symbolic values. Although these storage nodes must be identified, the simulator does not need to understand the detailed timing requirements of the latches in order to verify them.

2.3 Advantages

Timing constraints on circuits exist to ensure that signal transitions occur in the order required for proper operation. Some constraints are imposed by the circuit's environment, and some are due to structures internal to the circuit. These structures can be latches, dynamic gates, self-timed loops, etc. Static timing analysis relies on pattern matching routines to identify these structures from the circuit netlist and apply timing constraints based on a set of precompiled rules.

In a full-custom design environment, designers often creatively hand-optimize circuitry to take advantage of local don't-care cases or fix critical timing paths. These hand-optimized circuits rarely match the patterns built into a static timing analyzer, causing it to apply incorrect constraints. To perform a timing analysis in this situation, designers are left with two equally unattractive options: develop a substantial simulation suite or train the analyzer to "understand" the circuit. Both options are labor intensive, and the simulation option may simply be infeasible if the circuit is large enough. The result in either case is a time-consuming and error-prone analysis.

Given the same delay computation model, output arrival times computed by a symbolic simulator will be more accurate than those computed by a timing analyzer. The simulator is not susceptible to false paths, which will be eliminated by a dynamic sensitization criteria. McGeer demonstrates that the dynamic criteria cannot underestimate the true circuit delay, while the static criteria can [10]. While the dynamic criteria does not satisfy the monotone speedup property, we argue that it matches reality much more closely and will give a higher quality estimate of the true delay. Furthermore, McGeer shows in [11] that the dynamic criteria does satisfy the monotone speedup property for dynamic(precharge unate) circuits, an application for which this methodology is particularly well-suited.

In addition, a symbolic timing simulator need not make worst-case assumptions about the state of the surrounding circuitry when computing delays. A static analyzer must assume worst-case loading, simultaneous-switching, and capacitive-coupling during delay calculation to ensure a conservative analysis. Because a symbolic simulator knows the state of every node in the circuit, it can avoid this pessimism if it's delay model correctly accounts for these effects.

2.4 Complexity

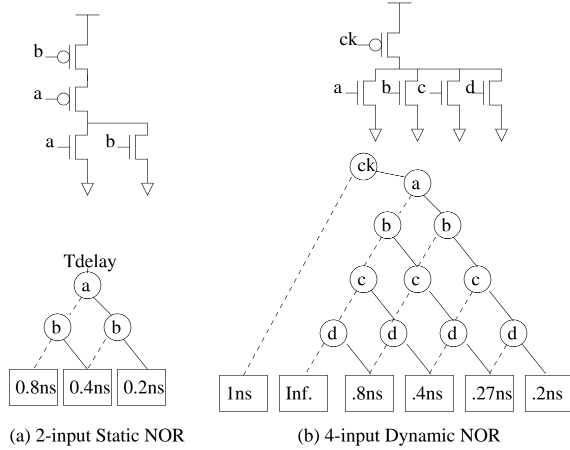
Since we are performing a complete analysis over all input combinations, the worst-case complexity of symbolic timing simulation is necessarily exponential in the size of the circuit. However, the actual complexity is highly dependent on the efficiency with which the circuit's node functions can be represented. Implementations of symbolic simulators using BDDs to compute and represent node functions have been shown to be very efficient for a wide range of interesting circuits.

While the computational complexity of this technique is clearly greater than that of static timing analysis, we show in Section 4.3 that it is feasible for moderately sized circuits. Symbolic timing simulation will provide designers with an alternative when static analysis fails. Currently, no such alternative exists.

3 Implementation

3.1 Data structures

Our approach utilizes symbolic simulation, where a circuit node's value is a Boolean function of the input variables. We employ the same node value encoding scheme as COSMOS[4, 5]. Two Bi-



(a) 2-input Static NOR (b) 4-input Dynamic NOR

Figure 3: Example Delay MTBDDs

```
[1]  SymbolicSchedule( Node, DCVal, Tdelay )
[2]     while( Tdelay != Constant( inf. ) )
[3]        dmin = MtbddMinValue( Tdelay )
[4]        Event.mask = MtbddEqual(Tdelay,dmin)
[5]        Event.value = DCVal
[6]        Event.time = curtime + dmin
[7]        EnqueueEvent(Node,Event)
[8]        Tdelay = ITE(Event.mask,inf.,Tdelay)
[9]
[10] SymbolicSimulate()
[11]    while( <Node,Event> = GetNext() )
[12]       curtime = Event.time
[13]       Node.value = ITE( Event.mask,
[14]              Event.value, Node.value )
[15]       For each affected node N
[16]          DCVal = ComputeDC(N)
[17]          Tdelay = ComputeDelay(N)
[18]          SymbolicSchedule( N,
[19]                        DCVal, Tdelay)
```

Figure 4: Scheduling algorithm

nary Decision Diagrams (BDDs)[3], node.$\mathbf{h}$[1] and node.$\mathbf{l}$, represent the current state of each node. These BDDs can be thought of as representing the on-functions of the pullup and pulldown chains attached to the node. If, for a given input pattern, both functions are true, the node is in an indeterminate state *X*.

At the time that a node value transition occurs, we calculate a delay function in the form of an MTBDD. Each terminal of the delay MTBDD $\mathbf{T_{delay}}$ represents a point in time at which a portion of the transition can occur. As an example, Figure 3(a) shows the delay MTBDD that might result from a static 2-input NOR gate formed from equally sized transistors. To interpret this MTBDD, we follow the solid arc when the associated variable is true, and the dashed arc when it is false. Note that the pulldown delay is smaller in the case where both *a* and *b* are true than in the case where only one is true. Also note that the pullup delay (*a* and *b* false) is significantly larger than the pulldown delay.

In the worst case, $\mathbf{T_{delay}}$ can become exponentially large relative to the size of the circuit. However, our delay calculations are performed on single stages of logic, which tend to be quite small. In addition, larger logic stages are often very regular, allowing for efficient MTBDD representations. Consider the 4-input dynamic NOR gate in Figure 3(b), and its delay MTBDD. One terminal is required for each number of pulldown FETs that can be on at the same time, resulting in a $\mathbf{T_{delay}}$ with 17 total nodes. This type of circuit will produce a $\mathbf{T_{delay}}$ that is quadratic in the circuit size.

The advantage in using MTBDDs is that we can calculate delays by using a simple polynomial-time algorithm for applying any arbitrary binary operator (+,*,/,...) to two input MTBDDs. This function,*MtbddApply*, is similar to the well-known BDD Apply algorithm[3].

3.2 Symbolic Simulation

Given a method to compute the MTBDD $\mathbf{T_{delay}}$, we can perform a symbolic timing simulation using the algorithm shown in Figure 4. We believe this to be the first symbolic scheduling algorithm capable of dealing with both continuous time and data-dependent delays. The key features are the use of MTBDDs to represent data-dependent delay values, and the inclusion of masks specifying the logical conditions under which events occur.

[1]BDDs and MTBDDs appear in bold-face

SymbolicSchedule() is the event scheduler. It takes as arguments $Node$, $\mathbf{DCVal}$ and $\mathbf{T_{delay}}$. $\mathbf{DCVal}$ is a BDD representing the function to which $Node$ will settle. Lines 2-3 repeatedly select the smallest remaining terminal value, d_{min}, in $\mathbf{T_{delay}}$. Line 4 computes the event mask by selecting the subset of events which will occur at time $curtime + d_{min}$. This is done through the function *MtbddEqual(*$\mathbf{M}$*,v)* which returns a BDD where all terminals of $\mathbf{M}$ that are equal to v are replaced by '1', and all others by '0'. Line 6 computes the time at which the new event will occur, and line 7 inserts the new event into the queue. Line 8 modifies $\mathbf{T_{delay}}$ so that d_{min} will get the next smallest terminal on the subsequent iteration.

SymbolicSimulate forms the main body of the simulator. Line 11 repeatedly selects the next event to be processed, and line 12 advances the current time. Lines 13-14 compute the new value of $Node$ by selecting the event value for all cases where the event mask is true, and $Node$'s current value otherwise. Line 15 determines all nodes which will be affected by the event being processed. The functions *ComputeDC* and *ComputeDelay* are called to determine each node's new DC value and the new delay MTBDD. Line 18 then calls *SymbolicSchedule* to enqueue the resultant events.

4 SirSim

4.1 Delay computation

To test these algorithms, we implemented a symbolic version of IRSIM called SirSim. The delay calculation mechanism, first introduced in the simulator nRSIM[6], models transistors as switched resistors, lumps all capacitances to ground, and uses the first RC time constant as a delay estimate (Elmore delay). It handles both resistively- and capacitively-driven nodes, and has a conservative model of nodes in the unknown "X" state.

To perform the nRSIM calculations symbolically, we used the BDD and MTBDD routines from CUDD version 2.2. Further information on these routines can be found in [2]. We represent the symbolic resistance of a transistor by using an MTBDD with two terminals: ∞ and the finite on-resistance. Since we can perform multiplication, addition and division on MTBDDs, we can evaluate series and parallel combinations of these resistors. Sym-

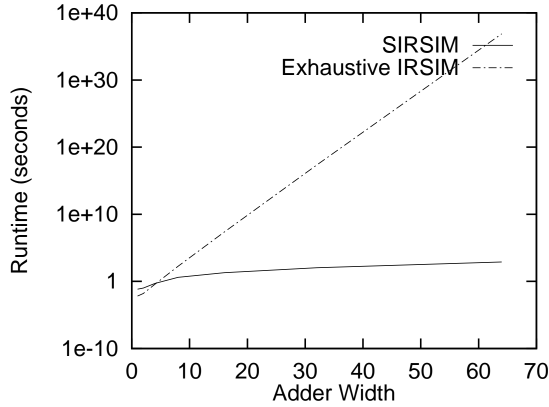


Figure 5: Runtime vs. Adder Width

bolic capacitances are easily computed using one MTBDD summing capacitors charged high, and another summing capacitors charged low. Delay values computed in this manner are identical to those computed by IRSIM.

While the Elmore delay is not particularly accurate, it is straight-forward and captures the data-dependent nature of circuit delays. Since MTBDDs allow arbitrary mathematical operations, any delay calculator could theoretically be implemented symbolically, although a significantly more complicated algorithm would be quite inefficient. Instead, one could identify delay cases using the symbolic Elmore delay, and then compute actual delay values with a standard circuit simulator.

4.2 Experimental Results

4.2.1 Adders

For our first set of substantial test cases, we ran SirSim on varying widths of Manchester carry-chain adders. We expected these circuits to exhibit worst-case behavior for our scheduling algorithm in two ways. First, event timings are highly dependent on the choice of input values, resulting in smaller sets of events that can be scheduled together as symbolic transitions. Second, the depth of the carry-chain logic is proportional to the width of the adder, which we expected to generate an exponential number of events relative to the circuit size. In most other cases, one would expect the depth of the logic cone to be O(log n), creating a polynomial number of events.

However, the runtimes were surprisingly good (Table 1), and in fact only grew as the cube of the adder width. For the 32-bit adder, SirSim performed a complete analysis in less than 2 minutes on a 300MHz UltraSparc system, representing a speedup over exhaustive conventional simulation of 10^{17}. For the 64-bit adder, SirSim required less than 15 minutes, and achieved a speedup of 10^{33}.

We believe the following analysis justifies the cubic behavior. The runtime will be composed primarily of two factors, the number of events processed E and the average cost of processing one event C. Since most of the processing cost is MTBDD traversal, C will be proportional to their average size. For an adder, we know that the output BDDs are linear in the width of the adder, so we can expect this to be true of the computational MTBDDs as well, and thus of C. To estimate E, we look at the i-th adder bit-slice. Each slice will locally generate a constant number of

Table 1: SirSim Runtimes

Name	FETs	Inputs	MB	sec.
adder4	164	10	0.6	0.5
adder8	328	18	4.1	3.2
adder16	656	34	22.3	19.5
adder32	1312	66	102.4	107.1
adder64	2624	130	280.5	783.9
byp_adder8	388	18	4.0	3.2
byp_adder16	776	34	32.5	25.5
byp_adder32	1590	66	249.5	213.3
s298	582	17	0.87	0.9
s349	654	24	0.70	0.7
s382	682	24	3.09	2.7
s444	758	24	6.96	6.2
s820	1786	23	2.48	3.9
s1423	2996	91	0.69	1.3
s1494	3902	14	0.81	1.5
s5378	8902	214	64.7	69.4
sr_incr64	4218	129	39.8	40.5

events l on the carry output $carry_i$. In the Manchester carry chain design, there is only one delay path possible from $carry_i$ to $carry_{i+1}$. Thus if we assume that k_i events were scheduled on $carry_i$, then there should be $k_i + l$ events on $carry_{i+1}$. Since $carry_0$ is a constant and has no events, $carry_1$ will have only the locally generated l events, and $carry_n$ will have nl events. Thus the total event count $E = l \sum_{i=1}^{n} in = O(n^2)$,and the total runtime $T = EC = O(n^3)$.

We also constructed several widths of a carry bypass design. The runtimes (shown in Table 1) were slightly higher than those for the ripple carry design due to the additional circuitry. This test case is particularly interesting because carry bypass adders are notoriously difficult for static timing analysis due to the huge number of false paths.

To understand the speedup attained by performing the simulation symbolically, we compared the total number of symbolic events per timestep with the total number of real events for the 8-bit adder. We defined the real event count as the sum of all events that would occur in a given timestep in an exhaustive conventional simulation. This analysis revealed an average symbolic compression (ratio of real to symbolic events) of > 9600 for $n = 8$, which increases exponentially with the adder width.

4.2.2 Combinational Circuits

To determine how SirSim performs on combinational networks, we ran several of the ISCAS89 benchmarks. To obtain transistor-level networks, we replaced each gate with an equivalent nominally-sized static CMOS subcircuit. We also removed the DFFs and turned their inputs into primary outputs, and the outputs into primary inputs. As can be seen from Table 1, the runtimes varied substantially and were not highly correlated with the size of the circuit. The efficiency of this technique is heavily dependent on the BDD/MTBDD variable order selected, and on the compactness of the BDDs for the circuit's node functions. However, this data suggests that symbolic timing simulation is computationally feasible on reasonably large circuits.

4.2.3 Industry Example

We also implemented an industrial self-resetting 64-bit incrementer[8]. This circuit makes use of self-timed locally-

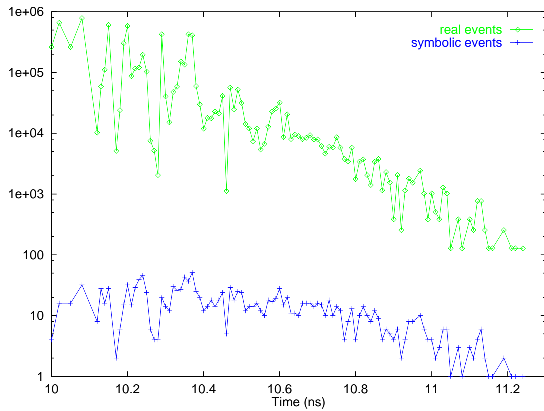


Figure 6: Symbolic and Real Events

generated reset signals to accept a pulsed input, compute the incremented value, signal a pulsed output, and reset itself to prepare for the next input. It uses no global clocks, and all operations are triggered by the pulsing of the input data lines.

We used SirSim throughout the implementation of the incrementer to verify both functionality and timing, and found it to be a very natural way to identify errors. By simulating a pulsed symbolic input vector and placing checks on the output lines, we located and debugged problems in connectivity, drive strengths, reset delays, etc.

In contrast, the original designers made use of a complicated ad hoc timing verification methodology, which they outlined in [12]. Their methodology involved adding pulse-propagation and overlapping pulse-width checks to an in-house static timing analyzer. Since SirSim implements an inertial delay model, it has virtually identical verification power to these special-purpose checks. This example clearly demonstrates SirSim's ability to handle even highly customized circuit design styles. Furthermore, SirSim's runtime for this 4200-transistor design was ≈ 40 seconds(*sr_incr64* in Table 1), which was less time than required to generate netlists from our schematics.

4.2.4 Other Results

Of perhaps greater theoretical interest are the traces shown in Figure 6. In order to determine the limit of speedup attainable by performing a symbolic simulation, we compared the total number of symbolic events per timestep with the total number of real events for the 8-bit adder. We defined the real event count as the sum of all events that would occur in a given timestep in an exhaustive conventional simulation comprised of 2^{16} IRSIM runs, and computed it by examining the don't-care sets of the symbolic event trace. Despite the highly data-dependent timing behavior of this circuit, SirSim was quite successful in symbolically encoding large groups of real events. The resulting average symbolic compression (ratio of real events to symbolic events) is $\approx 10^3$ for the 8-bit case and grows exponentially with increasing width.

5 Conclusions

We have introduced a new way to verify the timing and functionality of custom digital circuits that has significant advantages in robustness and accuracy over static timing analysis. We have also shown algorithms that can be used to realize this technique and presented the simulator SirSim. Despite higher complexity, our results demonstrate that symbolic timing simulation is feasible for many reasonably sized circuits. We believe this methodology will provide custom circuit designers with an important new capability.

References

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, and G. D. Hachtel. Timing analysis of combinational circuits using add's. *EDAC-94: The European Design and Test Conference*, pages 625–629, Feb 1994.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *ACM/IEEE International Conference on Computer Aided Design*, pages 38–43, October 1992.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):79–85, August 1986.

[4] R. E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, CAD-6(4):618–633, July 1987.

[5] R. E. Bryant. Boolean analysis of mos circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, CAD-6(4):634–639, July 1987.

[6] C. Y. Chu. *Improved Models for Switch-Level Simulation*. PhD thesis, Stanford University, October 1988.

[7] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Certifi ed timing verifi cation and the transition delay of a logic circuit. *Proceedings of the Design Automation Conference*, 1992.

[8] R. A. Haring, M. S. Milshtein, T. I. Chappell, S. H. Dhong, and B. A. Chappell. Self resetting logic register and incrementer. *Symposium on VLSI Circuits*, pages 18–19, 1996.

[9] R. B. Hitchcock, G. L. Smith, and D. D. Cheng. Timing analysis of computer hardware. *IBM Journal of Research and Development*, 26(1):100–105, Jan 1982.

[10] P. C. McGeer and R. K. Brayton. Effi cient algorithms for computing the longest viable path in a combinational network. *Proceedings of the Design Automation Conference*, 1989.

[11] P. C. McGeer and R. K. Brayton. Timing analysis in precharge/unate networks. *Proceedings of the Design Automation Conference*, 1990.

[12] V. Narayanan, B. A. Chappell, and B. M. Fleischer. Static timing analysis for self resetting circuits. *ACM/IEEE International Conference on Computer Aided Design*, pages 119–126, 1996.

[13] C. J. Seger and R. E. Bryant. *Modeling of Circuit Delays in Symbolic Simulation*, volume 2 of *Formal VLSI Correctness Verification*. Elsevier Science Publishers, 1990.