CMOS Circuit Verification with Symbolic Switch-Level Timing Simulation

Clayton B. McDonald (clayton@ece.cmu.edu)
Randal E. Bryant (randy.bryant@cs.cmu.edu)
Electrical and Computer Engineering Department
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213

Abstract

Symbolic switch-level simulation has been extensively applied to the functional verification of CMOS circuitry. We have extended this technique to account for real-valued, data-dependent delay values, and have developed a novel mechanism for symbolically computing data-dependent Elmore delays. We present our symbolic simulation and delay calculation algorithms, and discuss their application to the timing and functional verification of full-custom transistor-level CMOS circuitry.

I. Introduction

Symbolic simulation is a form of data-parallel simulation in which Boolean functions are used to encode a set of input data patterns. In conventional simulation the user applies a pattern of constant 0's and 1's to each of the circuit inputs, steps the simulator, and verifies that the outputs and state elements have settled to the desired values. With a symbolic simulator, the user may substitute Boolean variables for any of the input values to signify that the input may be either a 0 or a 1. If the user applies n Boolean variables, the symbolic simulator will perform the equivalent of 2^n conventional simulations. The outputs and state elements of the circuit will evaluate to Boolean functions of the input variables, which can be verified against the desired behavior.

Previous work on symbolic simulation has largely focused on unit-delay models, in which all node transitions require a uniform amount of time. This is sufficient for the majority of functional verification problems, but is clearly inadequate for verifying circuit timing. In some cases, as we show in Section III-A, more sophisticated timing models are often required simply to model functionality.

Event-driven symbolic simulation has previously been extended to handle some degree of timing information. Devadas et.al. [6] constructed a gate-level symbolic simulator that utilized pre-assigned gate delays in order to study the transition delay of combinational circuits. However their gate-delay model is unable to simulate separate rising and falling delays, a crucial capability for extension to the transistor level. The skewed inverter in Figure 1(a) is one example which exhibits this behavior. Since the pulldown nFET is stronger than the pullup pFET, the output will fall more quickly than it will rise.

Seger and Bryant [16] proposed another extension to event-driven symbolic simulation to model rising vs. falling delays on specific nodes by inserting explicit delay elements and additional logic gates. One drawback of their model is its assumption of quantized delays. Furthermore, it is limited to the assignment of a single rising and falling delay value for any given node, which fails to capture several other data-dependent delay cases. Consider the data-dependent loading on the inverter output in Figure 1(b). Here, the output capacitance (and thus the inverter's delay) is dependent on the state of signal b. Another case is the asymetric NOR-gate in Figure 1(c), where the falling delay is dependent on which input fired.

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

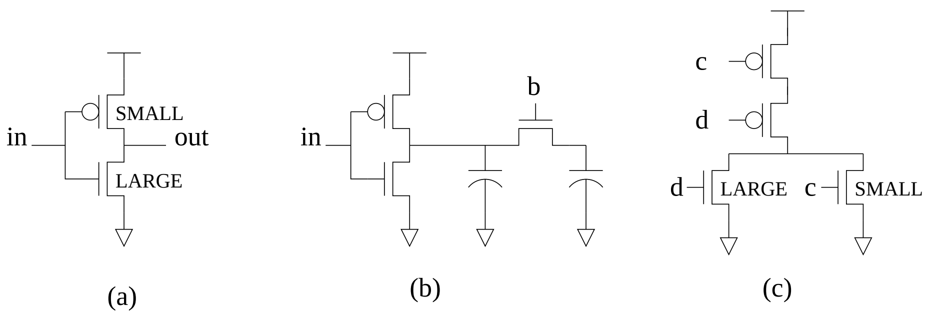


Fig. 1. Data-dependent Delay Examples

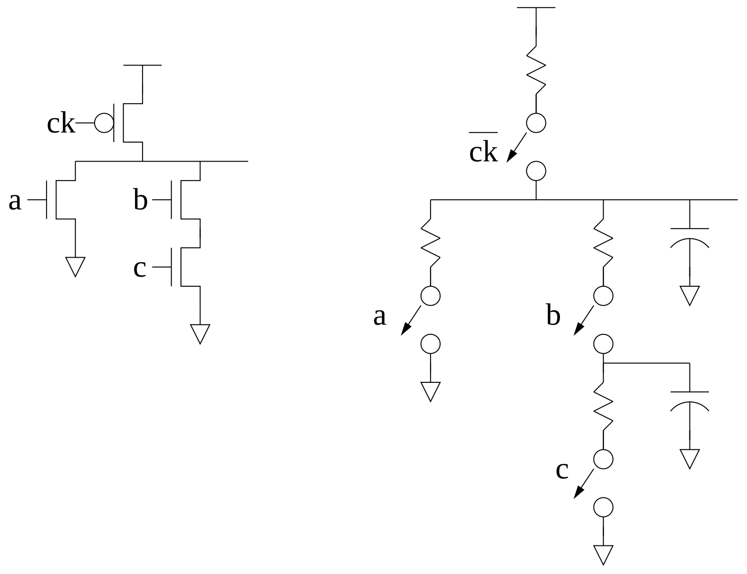


Fig. 2. IRSIM Circuit Model

To capture the full generality of the data-dependent delay model, we have developed a new methodology called *symbolic timing simulation* (STS). This methodology has been implemented in the simulator SirSim, a symbolic extension of the well-known transistor-level timing simulator IRSIM[15]. SirSim is event-driven and utilizes several novel event-management techniques which allow for arbitrary real-valued delays. To capture the effects demonstrated in Figure 1, we have also developed a procedure for computing data-dependent delays in transistor networks at run-time. The details of these algorithms are presented in Section II.

STS has applications in both functional and timing verification of VLSI circuits. We discuss these applications and the advantages of STS in Section III, and present experimental results from applying SirSim to a number of substantial test-cases in Section IV. Section V gives our conclusions and suggests future work.

II. Implementation

As a testbed for symbolic timing simulation, we have implemented a symbolic version of the timing simulator IRSIM[15]. IRSIM is itself derived from two earlier simulators, RSIM and nRSIM. RSIM[18] introduced the concept of event-driven switch-level timing simulation based on Elmore delays[7], [14], which are delay estimates computed as RC products. It models transistors as switched linear resistors and all capacitors are connected to ground. Figure 2 shows a simple circuit and its representation under the RSIM model.

RSIM contained a simple and somewhat pessimistic model of nodes with unknown (X) values. nRSIM[5] improved on this model of X values and introduced several other enhancements, such as an improved model of charge-sharing effects and the simulation of transient voltage spikes. Lastly, IRSIM implemented an incremental simulation model, where circuit updates could be analyzed with only partial re-simulation. SirSim (Symbolic IRSIM) implements the full nRSIM model with the exception of voltage spikes, but does not include incremental simulation. Thus it is primarily indebted to nRSIM, despite being based on the the IRSIM source code.

A. Event Handling

IRSIM is an event-driven timing simulator, and utilizes relatively standard event-management techniques with several interesting enhancements. All circuit nodes are capable of maintaining their voltage state, which can be either 0,1 or X. "Events" are defined as changes in the state of a circuit node at a particular time. Thus an event has the following form:

$$\begin{aligned} Event &= \langle Node, Time, Value\rangle \\ Node &: \text{Circuit node to be updated} \\ Time &: \text{Absolute time of event} \\ Value &: \text{New node value} \in \{0,1,X\} \end{aligned}$$

```
$CleanEvents$( node $n$, real $time$ )
    $\forall e \in EventQueue$ such that $(e.Node = n) \wedge (e.Time > time)$
        Delete $e$ from $EventQueue$

$Simulate()$
    while( $\langle node, value, time\rangle \leftarrow GetNext()$ )
        $curtime \leftarrow time$
        $node.value \leftarrow value$
        $N \leftarrow AffectedNodes(node)$
        $\forall n \in N$
            $newvalue \leftarrow ComputeDC(n)$
            if( $newvalue \neq n.value$ )
                $T_{delay} \leftarrow ComputeDelay(n)$
                $EnqueueEvent( \langle n, newvalue, curtime + T_{delay}\rangle )$
                $CleanEvents( n, curtime + T_{delay} )$
            else
                $CleanEvents( n, curtime )$
```

Fig. 3. Conventional Scheduling Algorithm

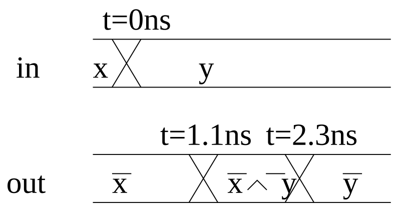


Fig. 4. Skewed Inverter Timing Diagram

Although events are handled as if instantaneous, IRSIM associates a transition time with each event that is used in the computation of resultant transition delays. While we will not discuss how this information was incorporated into SirSim, it is straightforward to do so using the techniques presented in this paper.

Pending events are stored in the event queue, where they are sorted by increasing time. The main simulation loop repeatedly selects the earliest event in the event queue and updates the node state. It then identifies affected downstream nodes, recomputes their steady-state node values, determines the delays to each node, and schedules the appropriate events.

IRSIM extends this procedure with an inertial delay model, which filters out input glitches having durations less than the stage delay. This is accomplished by removing all pending events on a stage output when it is determined that the current state matches the steady-state value.

IRSIM's event-handling procedure is shown in Figure 3. This algorithm has been re-organized slightly to facilitate comparison with the symbolic version, and several important features have been dropped. In particular, we will only discuss binary simulation here, while the generalization to ternary simulation (node values $\in \{0, 1, X\}$) will be covered in Section II-G. In addition, we do not show the handling of charge-sharing effects and several efficiency enhancements.

Simulate contains the main simulation loop. It calls *GetNext* to obtain the earliest pending event, and updates *curtime* and the node state. It then calls *AffectedNodes* to determine which downstream nodes need to be visited. For each downstream node, it determines the new steady-state value using *ComputeDC*, and checks if the node value has changed. If it has changed, it computes the logic stage delay with *ComputeDelay* and schedules a new event on the node.

CleanEvents scans through the event queue, deleting all events on the specified node that are scheduled to occur after a certain time. The first use of *CleanEvents*, immediately after the new event is enqueued, overrides previously computed node values that would take effect after the newly-inserted event. This is necessary because the newly-inserted event represents the latest information about the node's steady-state value and shouldn't be overwritten by long-delay events generated previously using old information. The second call to *CleanEvents*, performed when no change is detected on the node, is used to implement the inertial delay model.

B. Symbolic Event Handling

Utilization of IRSIM's event-handling methodology in the symbolic domain requires some additional enhancements. First of all, each circuit-node's state is no longer a simple scalar value in $\{0,1\}$[1], but a Boolean function of the variables applied to the circuit inputs. We have chosen to represent node state with Reduced Ordered Binary Decision Diagrams (ROBDDs, or simply BDDs)[3].

The primary difficulty is the proper scheduling of data-dependent delays. Consider again the skewed inverter example from Figure 1. If the input switches from symbolic variable x to symbolic variable y, either or both of which could represent 0 or 1, it is not obvious when the resultant event should be scheduled on the output. However, for any given input pattern, the output will transition at some well-defined point in time. Thus, the value of that node at any time can be represented by a Boolean function of the input variables, and the full symbolic transition is actually a progression through a series of node functions.

Using this idea, we will construct a valid sequence of functions for the output node of the skewed inverter. If we determine that a falling transition on the output occurs after 1.1ns and a rising transition occurs after 2.3ns, we obtain the series of 3 node functions shown in Figure 4. 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 $\overline{x} \wedge \overline{y}$. In general, the output node function will progress from being dependent only on the old input variables to being dependent on the new, and in between it will be dependent on a mix of the two.

B.1 Event Masks

The key to handling data-dependent delays is to view symbolic simulation as simultaneously simulating all possible input patterns. Under different input patterns, a particular transition might occur at different points in time, or perhaps not occur at all. For the inverter example above, the falling transition always occurs at the same time, but only when the old value x is 0 and the new value y is 1. The rising transition only occurs when x is 1 and y is 0, and no transition occurs if $x = y$.

In the conventional event-handling algorithm described above, node state was updated by assigning an event's *value* to the node. For the symbolic case, we wish to update the node state selectively, so that it is not disturbed for input patterns under which no event should occur at that time. This is accomplished using *event masks*, which are Boolean functions that encode the conditions under which a transition occurs. The mask is added to each event record, such that an event is now defined as :

$$Event = \langle Node, Time, \mathbf{Value}, \mathbf{Mask} \rangle$$

Note that we have used boldface for the Value and Mask fields to highlight the fact that they are Boolean functions rather than scalar values. This convention will be used for the remainder of this paper.

Rather than simply copying the event's $Value$ field into the output node's state, we select the event value only for those cases where the event mask is true:

$$Node.Value \leftarrow (Event.\mathbf{Mask} \wedge Event.\mathbf{Value}) \vee (\overline{Event.\mathbf{Mask}} \wedge Node.\mathbf{Value})$$

Since our implementation utilizes BDDs, the above computation can be more efficiently computed using the equivalent ITE (if-then-else) operation, which forms the core of most BDD packages :

$$Node.Value \leftarrow ITE(Event.\mathbf{Mask}, Event.\mathbf{Value}, Node.\mathbf{Value})$$

For the skewed inverter example, we would schedule events at both 1.1ns and 2.3ns having masks $(\overline{x} \wedge y)$ and $(x \wedge \overline{y})$ respectively, both with the steady-state value $\overline{y}$:

$$\begin{aligned}
@t = 1.1ns \quad out.\mathbf{Value} &= ((\overline{x} \wedge y) \wedge \overline{y}) \vee (\overline{(\overline{x} \wedge y)} \wedge \overline{x}) \\
&= \overline{y} \wedge \overline{x} \\
@t = 2.3ns \quad out.\mathbf{Value} &= ((x \wedge \overline{y}) \wedge \overline{y}) \vee (\overline{(x \wedge \overline{y})} \wedge (\overline{y} \wedge \overline{x})) \\
&= \overline{y}
\end{aligned}$$

[1]X values will be incorporated in Section II-G

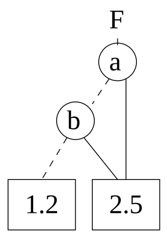


Fig. 5. Example MTBDD

B.2 Delay MTBDDs

Before we can describe our implementation of event masks in SirSim's event-handling algorithm, we must describe the data structure used to encode data-dependent delay values. Since the potential delays associated with a transition are defined for mutually disjoint sets of input assignments, they can be represented as mappings from Boolean values to real numbers:

$$F : \mathcal{B}^{\mathrm{n}} \mapsto \mathcal{R}$$

One method of representing such mapping functions is known as a Multi-Terminal Binary Decision Diagram (MTBDD)[1]. MTBDDs are generalizations of BDDs that allow an arbitrary number of real-valued terminals. For example, the MTBDD in Figure 5 represents the function F having two inputs a and b. To determine the return value for any given input assignment, we work downwards from the root, following the solid arc from nodes assigned 1 and the dashed arcs from nodes assigned 0. We can see that $F \leftarrow 2.5$ when either a or b is 1, and $F \leftarrow 1.2$ otherwise.

To describe the MTBDD operators needed by our algorithm, we introduce the following notation. Let us define $A = \{a_0, a_1, \ldots, a_{2^n-1}\}$ as the set of 2^n possible assignments to the n variables in the support of MTBDD $\mathbf{M}$, and define $\mathbf{M_{a_i}}$ as the terminal value returned by $\mathbf{M}$ under the assignment a_i. We will consider BDDs to be a special case of MTBDDs where $\mathbf{M_{a_i}}$ is limited to the set {0,1}.

The operator *MtbddITE*$(\mathbf{I}, \mathbf{T}, \mathbf{E})$ is similar to the BDD ITE operator, selecting $\mathbf{T}$ for assignments which satisfy $\mathbf{I}$ and selecting $\mathbf{E}$ otherwise. Note that $\mathbf{I}$ must be a BDD, while $\mathbf{T}$ and $\mathbf{E}$ are MTBDDs. *MtbddITE* returns an MTBDD $\mathbf{R^{ITE}}$, such that for all i:

$$\left[\left((\mathbf{I_{a_i}} = 1) \rightarrow (\mathbf{R^{ITE}_{a_i}} = \mathbf{T_{a_i}})\right) \wedge \left((\mathbf{I_{a_i}} = 0) \rightarrow (\mathbf{R^{ITE}_{a_i}} = \mathbf{E_{a_i}})\right)\right]$$

The functions *MtbddEqual*$(\mathbf{M}, v)$ and *MtbddThreshold*$(\mathbf{M}, v)$ are used to map from MTBDDs to BDDs. *MtbddEqual* replaces all terminals in MTBDD $\mathbf{M}$ that are equal to v with 1, and replaces all others with 0. *MtbddThreshold* replaces terminals in $\mathbf{M}$ that are greater than v with 1, and the remainder with 0. They return the BDDs $\mathbf{R^{Equal}}$ and $\mathbf{R^{Threshold}}$ respectively, such that for all i:

$$\left[\mathbf{R^{Equal}_{a_i}} = 1 \iff (\mathbf{M_{a_i}} = v)\right]$$

$$\left[\mathbf{R^{Threshold}_{a_i}} = 1 \iff (\mathbf{M_{a_i}} > v)\right]$$

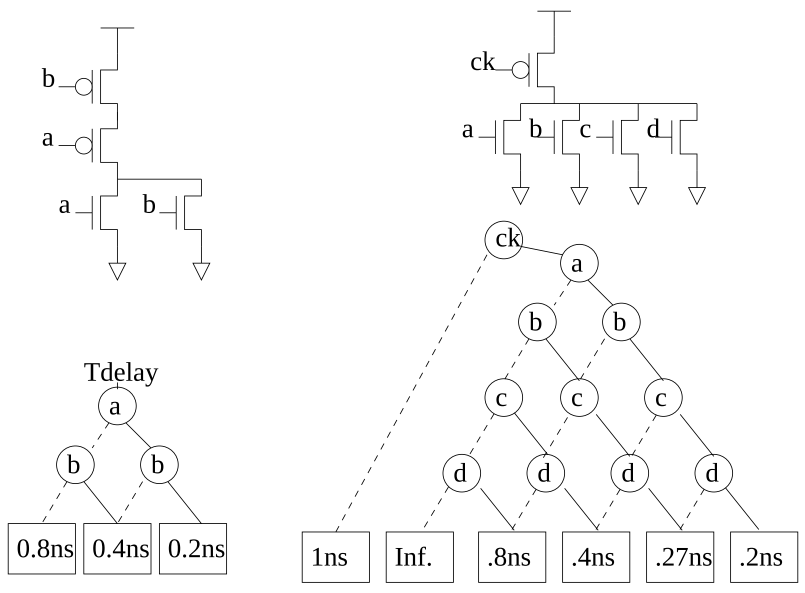
The last operator needed for this algorithm is *MtbddMinValue*$(\mathbf{M})$, which returns the smallest terminal value d in MTBDD $\mathbf{M}$:

$$d = Min_i(\mathbf{M_{a_i}})$$

At times, we will also need to convert scalar constants into trivial MTBDDs containing only a single terminal node. For an arbitrary scalar α, the trivial MTBDD will be denoted $[\alpha]$.

To illustrate the use of MTBDDs for representing data-dependent delays, Figure 6(a) shows the MTBDD that might result from a static 2-input NOR gate formed from equally sized transistors. 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, the delay MTBDD $\mathbf{T_{delay}}$ can become exponentially large relative to the number of inputs to the circuit. However, our delay calculations are performed on single stages of logic, and the subcircuits in consideration are typically quite small. Furthermore, larger logic stages tend to be highly regular, allowing for significant sharing in the MTBDD delay representation. For example, consider the 4-input dynamic NOR gate in Figure 6(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. An arbitrary width NOR gate constructed in this manner will produce a $\mathbf{T_{delay}}$ that is quadratic in the circuit size, rather than exponential.



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

Fig. 6. Example Delay MTBDDs

```
$SymbolicCleanEvents($ node $n$, real $time$, BDD $\mathbf{mask}$ $)$
    $\forall e \in EventQueue$ such that $(e.Node = node) \wedge (e.Time > time)$
        $e.\mathbf{Mask} \leftarrow e.\mathbf{Mask} \wedge \overline{\mathbf{mask}}$
        if( $e.\mathbf{Mask} = \mathrm{FALSE}$ )
            Delete $e$ from $EventQueue$

$SymbolicSchedule($ node $n$, BDD $\mathbf{value}$, MTBDD $\mathbf{T_{delay}}$ $)$
    while( $\mathbf{T_{delay}} \neq [\infty]$ )
        $dmin \leftarrow MtbddMinValue(\mathbf{T_{delay}})$
        $\mathbf{mask} \leftarrow MtbddEqual(\mathbf{T_{delay}}, dmin)$
        $time \leftarrow curtime + dmin$
        $SymbolicCleanEvents(\ n, time, \mathbf{mask}\ )$
        $EnqueueEvent(\ \langle node, time, \mathbf{value}, \mathbf{mask} \rangle\ )$
        $\mathbf{T_{delay}} \leftarrow MtbddITE(\mathbf{mask}, [\infty], \mathbf{T_{delay}})$

$SymbolicSimulate()$
    while( $\langle Node, Time, \mathbf{Value}, \mathbf{Mask} \rangle \leftarrow GetNext()$ )
        $curtime \leftarrow Time$
        $Node.\mathbf{value} \leftarrow ITE(\mathbf{Mask}, \mathbf{Value}, Node.\mathbf{value})$
        $N \leftarrow SymbolicAffectedNodes(Node)$
        $\forall n \in N$
            $\mathbf{newvalue} \leftarrow SymbolicComputeDC(n)$
            $\mathbf{change} \leftarrow \mathbf{newvalue} \oplus n.\mathbf{value}$
            if( $\mathbf{change} \neq \mathrm{FALSE}$ )
                $\mathbf{T_{delay}} \leftarrow SymbolicComputeDelay(n)$
                $\mathbf{T_{delay}} \leftarrow MtbddITE(\mathbf{change}, \mathbf{T_{delay}}, [\infty])$
                $SymbolicSchedule(n, \mathbf{newvalue}, \mathbf{T_{delay}})$
            $SymbolicCleanEvents(\ n, curtime, \overline{\mathbf{change}}\ )$
```

Fig. 7. Symbolic Scheduling Algorithm

B.3 Symbolic Event-Handling Procedure

We now have the basic machinery needed to implement the symbolic event handling procedure for SirSim (Figure 7), based on the conventional IRSIM algorithm presented above. Throughout this discussion, we will continue to denote symbolic values (BDDs and MTBDDs) with boldface (**F**), while scalar values will appear in normal type. In SirSim, all BDD and MTBDD primitive operations are performed using the University of Colorado Decision Diagram Package (CUDD) version 2.2.0[1], [17].

SymbolicSimulate forms the main body of the simulator, and it differs from *Simulate* in several places. First of all, scalar node values have been replaced with symbolic node values, represented as BDDs, and masks have been added to each event record. Also, all calls to subroutines have been replaced by symbolic versions.

As discussed in Section II-B.1, the node state update is performed selectively using the *ITE* operator and the event mask. Once the node state has been updated, *SymbolicSimulate* identifies those nodes requiring recomputation, and iterates on them as in the conventional algorithm. *SymbolicComputeDC* returns a BDD representing the symbolic steady-state value for the node. In the symbolic case, the same node may change state under one input assignment but remain stable under another. Therefore, we must compute the function **change** as the XOR of the old and new state, and then use this function to selectively perform both new event scheduling and event-cancellation.

New event scheduling begins by computing an MTBDD representing the data-dependent logic-stage delay. Using *MtbddITE*, $\mathbf{T_{delay}}$ is then modified by setting it to ∞ for all input assignments where no state change occurs.

SymbolicSchedule is new, and is responsible for creating new events for each of the possible delay-cases represented in the delay MTBDD. Its main loop repeatedly selects the smallest remaining terminal value, *dmin*, in $\mathbf{T_{delay}}$. It then selects the subset of events which will occur at time $curtime + dmin$ using *MtbddEqual*. This result becomes the mask for the new event, which is assembled and inserted into the event queue. The last line of the *SymbolicSchedule* loop modifies $\mathbf{T_{delay}}$ so that *dmin* will get the next smallest terminal on the subsequent iteration.

Event cancellation to implement the inertial delay model is performed by the call to *SymbolicCleanEvents* inside *SymbolicSimulate*. Like *CleanEvents*, its purpose is to remove any events on the specified node that occur after a certain time. However, event removal is now qualified by a masking function, such that only those events that occur under certain input assignments will be removed. This translates directly into reducing the event mask by ANDing it with the inverse of the passed-in masking function. If the event mask becomes *FALSE* (0 under all input assignments), the event is deleted from the queue.

C. Determining the Affected Nodes

The first thing that occurs after updating a node's value is to compute the set of downstream nodes that will need to be recomputed. While this is perhaps the least complicated portion of the IRSIM algorithm, it required some of the most subtle modifications to enable symbolic operation.

Under a switch-level model with grounded capacitors, a node's state can only be affected by other nodes that are reachable through transistor channels, and by the gate-nodes of those transistors. Groups of nodes connected by source-drain (channel) connections are called *channel-connected regions*(CCRs). To enumerate the nodes affected by a node transition, it would be sufficient to simply list all nodes of all CCRs connected to the switching node. However, this overlooks the fact that some nodes in each CCR may only be reachable through currently turned-off transistors. For small CCRs such as static logic gates, simple enumeration only results in a small number of unnecessary node re-evaluations. However for very large CCRs such as barrel-shifters or SRAMs, the number of unnecessary re-evaluations could become prohibitive. Since these large CCRs typically have mutually-exclusive control lines that effectively partition the CCR into smaller regions, we might greatly reduce the number of nodes to be processed by identifying them at runtime. This is the approach taken by both IRSIM and SirSim.

IRSIM's affected-node computation is complicated by an additional responsibility. Since the Elmore delay (discussed below) is defined only for tree structures, we must heuristically break any loops formed by conducting transistors. This loop breaking is accomplished by setting a *broken* flag on transistors which close conducting loops.

In IRSIM, affected-node identification and loop-breaking are done using a breadth first search, but a depth-first version converts more easily to the symbolic case. Thus, Figure 8 presents a depth-first version of IRSIM's affected-node procedure. The search is started at the sources and drains of all transistors whose gates are connected to the switching node. *AffectedRecur* performs a recursive depth-first search through source/drain connections. All nodes discovered are added to the list of affected nodes, and transistors are marked as *broken* if they lead to a previously reached node. This algorithm dynamically identifies the nodes that make up the channel-connected regions affected by the transitioning node.

In SirSim, the algorithm is conceptually similar though it is complicated by the fact that transistor-gates can have symbolic state values. Thus, each transistor may be "transparent" only under certain input assignments. Furthermore, we can actually reach a node several times under mutually disjoint sets of input assignments without being forced to break loops. In fact, the transistor **broken** flag itself must also be symbolic, since there will be input assignments for which the transistor closes a loop, and others for which it does not.

```
$AffectedNodes$( node $n$ )
    $A \leftarrow \emptyset$
    $\forall t$ such that $t.gate = n$
        $A \leftarrow A \cup AffectedRecur(t.source, NULL)$
        $A \leftarrow A \cup AffectedRecur(t.drain, NULL)$
    return $A$

$AffectedRecur$( node $n$, transistor $via$)
    if( $n.reached$ )
        $via.broken \leftarrow TRUE$
        return $\emptyset$
    $n.reached \leftarrow TRUE$
    $A \leftarrow \{n\}$
    $\forall t \neq via$ such that $t.source = n$ or $t.drain = n$
        if( $n = t.source$ )
            $other \leftarrow t.drain$
        else
            $other \leftarrow t.source$
        if( $transparent$(t) )
            $A \leftarrow A \cup$ $AffectedRecur$( $other, t$)
    return $A$

$transparent$( transistor t )
    if($t.type = NFET$)
        return ($t.gate.value = 1 \wedge t.broken = 0$)
    if( $t.type = PFET$)
        return ($t.gate.value = 0 \wedge t.broken = 0$)

```

Fig. 8. Computing Affected Nodes in IRSIM

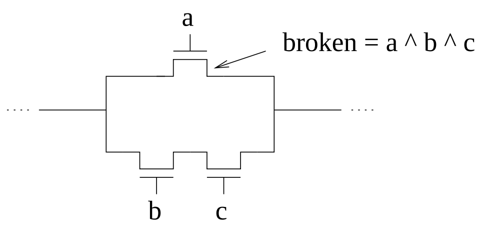


Fig. 9. Data-Dependent Transistor Loop

For example, the simple circuit in Figure 9 contains a data-dependent loop. Since the gates of all three transistors have symbolic values, they will only form a closed loop when a,b, and c are all 1. This means we must set the **broken** flag for at least one of these transistors to the function $(a \wedge b \wedge c)$. It doesn't matter which transistor we set the flag for, and the choice will be determined by the order in which the transistors happen to be identified. With the **broken** flag set as shown, the transistor controlled by node a will be considered non-conducting under all input assignments that satisfy the function $(a \wedge b \wedge c)$.

Figure 10 shows SirSim's algorithm for identifying affected nodes. At each node, we maintain a BDD, Node.**reached**, to keep track of input assignments under which the node has already been reached. Also, at each level of recursion, we keep track of the input assignments for which the search is still active. We return when this **active** BDD becomes FALSE or no unexplored transistors remain. When the **active** function intersects the Node.**reached** function, the result (**loop**) gives the input assignments under which this node was reached multiple times. The value of **loop** is used to compute a new **broken** flag, and then to deactivate further search under those input conditions. If the **active** function becomes FALSE, then the recursion terminates. The remainder translates directly from the conventional *AffectedNodes*.

```
SymbolicAffectedNodes( node $n$ )
    $A \leftarrow \emptyset$
    $\forall t$ such that $t.gate = n$
        $A \leftarrow A \cup SymbolicAffectedRecur(t.source, \emptyset, TRUE)$
        $A \leftarrow A \cup SymbolicAffectedRecur(t.drain, \emptyset, TRUE)$
    return $A$

SymbolicAffectedRecur( node $n$, transistor $via$, BDD **active** )
    $\mathbf{loop} \leftarrow \mathbf{active} \wedge n.\mathbf{reached}$
    if( $\mathbf{loop} \neq FALSE$ )
        $via.\mathbf{broken} \leftarrow via.\mathbf{broken} \vee \mathbf{loop}$
        $\mathbf{reached} \leftarrow \mathbf{reached} \wedge \overline{\mathbf{loop}}$
        if( $\mathbf{active} = FALSE$) return $\emptyset$
    $n.\mathbf{reached} \leftarrow n.\mathbf{reached} \vee \mathbf{active}$
    $A \leftarrow n$
    $\forall t \neq via$ such that $t.source = n$ or $t.drain = n$
        if( $n = t.source$ )
            $other \leftarrow t.drain$
        else
            $other \leftarrow t.source$
        $\mathbf{nextactive} \leftarrow \mathbf{active} \wedge SymbolicTransparent(t)$
        if( $\mathbf{nextactive} \neq FALSE$ )
            $A \leftarrow A \cup$ SymbolicAffectedRecur( $other, t, \mathbf{nextactive}$)
    return $A$

SymbolicTransparent( transistor t )
    if( t.type = NFET )
        return t.gate.value $\wedge$ $\overline{\text{t.}\mathbf{broken}}$
    if( t.type = PFET )
        return $\overline{\text{t.gate.}\mathbf{value}}$ $\wedge$ $\overline{\text{t.}\mathbf{broken}}$
```

Fig. 10. Computing Affected Nodes in SirSim

D. DC Value Computation

After we have identified which nodes are potentially affected by a transition, we need to compute a new steady-state value for each. In IRSIM this computation is performed by *ComputeDC*, shown in Figure 11.

Since all loops have been removed by *AffectedNodes*, the CCR forms a tree. For each node that must be recomputed, we explore outward along the tree in a depth-first manner until we reach Vdd, GND, or a non-conducting transistor. We then work backwards, performing series and parallel combinations of branches.

Each branch of the tree is represented by the tuple $\langle R_h, R_l, C_h, C_l, D\rangle$:

R_h	:	Equivalent resistance to Vdd through the branch
R_l	:	Equivalent resistance to GND through the branch
C_h	:	Total capacitance charged high in the branch
C_l	:	Total discharged capacitance in the branch
D	:	Driven flag: true if there is conducting path to Vdd or GND

For a full discussion of this tuple representation, and the correctness of the computations in functions *series* and *parallel*, we refer the reader to Chu's thesis[5].

When the recursive tree exploration terminates, it returns a tuple representing the parallel composition of all branches leading from the starting node. Using this tuple, *ComputeDC* computes a normalized steady-state node voltage and then determines whether it represents a logical 0 or 1. If the node is *driven* (connected to Vdd or GND through conducting transistors), then its DC voltage is given by a voltage-divider equation. If it is floating, then its voltage is given by a charge-sharing equation. If this voltage is greater than Vdd/2, it is considered a 1, otherwise a 0.

```
$ComputeDC$( node $n$ )
    $\langle R_h, R_l, C_h, C_l, D\rangle \leftarrow GetDC(n, \emptyset)$
    if( $D = true$ )
        $V \leftarrow R_l/(R_h + R_l)$
    else
        $V \leftarrow C_h/(C_h + C_l)$
    if( $V > 0.5$ )
        return 1
    else
        return 0

$GetDC$( node $n$, transistor $via$ )
    if( $n =$ VDD )
        return $\langle 0, \infty, 0, 0, \mathrm{TRUE}\rangle$
    if( $n =$ GND )
        return $\langle \infty, 0, 0, 0, \mathrm{TRUE}\rangle$

    $\langle R_h, R_l, C_h, C_l, D\rangle \leftarrow \langle \infty, \infty, 0, 0, \mathrm{FALSE}\rangle$

    $\forall t \neq via$ such that $((t.source = n) \vee (t.drain = n))$
        if( $n = t.source$ )
            $other \leftarrow t.drain$
        else
            $other \leftarrow t.source$
        if( $transparent$(t) )
            $\langle r_h, r_l, c_h, c_l, d\rangle \leftarrow series(t, GetDC(other, t))$
            $\langle R_h, R_l, C_h, C_l, D\rangle \leftarrow parallel(\langle R_h, R_l, C_h, C_l, D\rangle, \langle r_h, r_l, c_h, c_l, d\rangle)$
    return $\langle R_h, R_l, C_h, C_l, D\rangle$

$series$( transistor $t$, $\langle R_h, R_l, C_h, C_l, D\rangle$)
    $r_t \leftarrow$ t.res
    $r_h \leftarrow R_h + r_t * (1 + R_h/R_l)$
    $r_l \leftarrow R_l + r_t * (1 + R_l/R_h)$
    $c_h \leftarrow C_h$
    $c_l \leftarrow C_l$
    $d \leftarrow D$
    return $\langle r_h, r_l, c_h, c_l, d\rangle$

$parallel(\langle r_{h1}, r_{l1}, c_{h1}, c_{l1}, d_1\rangle, \langle r_{h2}, r_{l2}, c_{h2}, c_{l2}, d_2\rangle)$
    $r_h \leftarrow r_{h1} \parallel r_{h2}$
    $r_l \leftarrow r_{l1} \parallel r_{l2}$
    $c_h \leftarrow c_{h1} + c_{h2}$
    $c_l \leftarrow c_{l1} + c_{l2}$
    $d \leftarrow d_1 \vee d_2$
    return $\langle r_h, r_l, c_h, c_l, d\rangle$
```

Fig. 11. DC-Value Computation in IRSIM

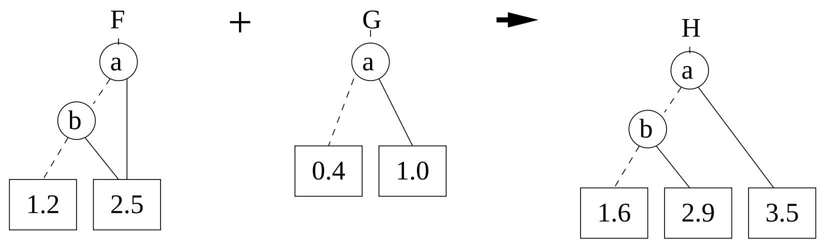


Fig. 12. Example MTBDD Operation

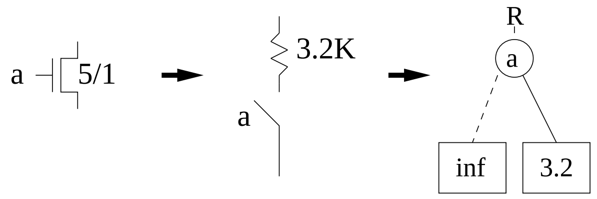


Fig. 13. nFET Resistance MTBDD

D.1 Symbolic Algebra

To discuss the symbolic version of the DC-value algorithm, we need to first introduce the concept of symbolic algebra. In SirSim, symbolic algebra is implemented with MTBDDs, again using the CUDD decision diagram package.

The key to performing symbolic algebra is the function *MtbddApply*$(op, \mathbf{M}, \mathbf{N})$. *MtbddApply* applies an arbitrary algebraic operator (e.g. $+, \times, /, \|$) to the argument MTBDDs $\mathbf{M}$ and $\mathbf{N}$. As before, we define $A = \{a_0, a_1, \ldots, a_{2^n-1}\}$ as the set of assignments to the n variables in the support of $\mathbf{M}$ and $\mathbf{N}$, and define $\mathbf{M_{a_i}}$ as the terminal value reached by $\mathbf{M}$ under the assignment a_i. *MtbddApply*(op,$\mathbf{M}$,$\mathbf{N}$) will return MTBDD the $\mathbf{R}$, such that :

$$\forall_i (\mathbf{R_{a_i}} = \mathbf{M_{a_i}} \text{ op } \mathbf{N_{a_i}})$$

For example, Figure 12 shows the result of using *MtbddApply* to compute $\mathbf{H} \leftarrow \mathbf{F} + \mathbf{G}$. For $(a = 0, b = 0)$, $\mathbf{H_0} = \mathbf{F_0} + \mathbf{G_0} = 1.2 + 0.4 = 1.6$, and similarly for the other input assignments.

In the discussion that follows, it will be convenient to use infix notation rather than explicit calls to *MtbddApply*. Thus, any algebraic expression involving bold-face operands should be interpreted as a call to *MtbddApply* with the appropriate operands:

$$\mathbf{F} + \mathbf{G} \equiv \textit{MtbddApply}(+, \mathbf{F}, \mathbf{G})$$

MtbddApply is virtually identical to the well-known *Apply* operator for ROBDDs[3], and its worst-case complexity is $O(|\mathbf{M}| \times |\mathbf{N}|)$, where $|\mathbf{M}|$ represents the number of nodes in MTBDD $\mathbf{M}$.

D.2 Representing Circuit Elements

In symbolic simulation, parameters such as transistor-conductance, which depend on the state of the circuit, require a symbolic representation.

In the IRSIM circuit model, transistors are represented as switched linear resistors, having a finite linear resistance when conducting and an infinite resistance otherwise. We can represent the symbolic resistance as an MTBDD having two terminals, the on-resistance and ∞. Figure 13 shows the symbolic resistance for an nFET t, computed as $\mathbf{R_t} \leftarrow \textit{MtbddITE}(t.gate.\mathbf{value}, [t.res], [\infty])$. Furthermore, since we can perform arbitrary algebraic manipulations on MTBDDs, we can also compute series and parallel combinations of transistors, as shown in Figure 14.

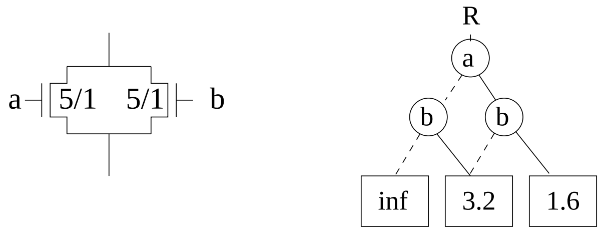


Fig. 14. Symbolic Parallel Resistances

```
$\mathit{SymbolicComputeDC}$( node $n$ )
    $\langle \mathbf{R_h}, \mathbf{R_l}, \mathbf{C_h}, \mathbf{C_l}, \mathbf{D} \rangle \leftarrow \mathit{SymbolicGetDC}(n, \emptyset, [1])$
    $\mathbf{V_r} \leftarrow \mathbf{R_l}/(\mathbf{R_h} + \mathbf{R_l})$
    $\mathbf{V_c} \leftarrow \mathbf{C_h}/(\mathbf{C_h} + \mathbf{C_l})$
    $\mathbf{V} \leftarrow \mathit{MtbddITE}(\mathbf{D}, \mathbf{V_r}, \mathbf{V_c})$
    $n.\mathbf{value} \leftarrow \mathit{MtbddThreshold}(\mathbf{V}, 0.5)$

$\mathit{SymbolicGetDC}$( node $n$, transistor $via$, BDD $\mathbf{active}$ )
    if( n = VDD )
        return $\langle [0], [\infty], [0], [0], [1] \rangle$
    if( n = GND )
        return $\langle [\infty], [0], [0], [0], [1] \rangle$

    $\langle \mathbf{R_h}, \mathbf{R_l}, \mathbf{C_h}, \mathbf{C_l}, \mathbf{D} \rangle \leftarrow \langle [\infty], [\infty], [0], [0], [0] \rangle$

    $\forall t \neq via$ such that $((t.source = n) \vee (t.drain = n))$
        if( $n = t.source$ )
            $other \leftarrow t.drain$
        else
            $other \leftarrow t.source$
        $\mathbf{nextactive} \leftarrow \mathbf{active} \wedge transparent(t)$
        if( $\mathbf{nextactive} \neq false$ )
            $\langle \mathbf{r_h}, \mathbf{r_l}, \mathbf{c_h}, \mathbf{c_l}, \mathbf{d} \rangle \leftarrow \mathit{SymbolicSeries}(t, \mathit{SymbolicGetDC}(other, t, \mathbf{nextactive}))$
            $\langle \mathbf{R_h}, \mathbf{R_l}, \mathbf{C_h}, \mathbf{C_l}, \mathbf{D} \rangle \leftarrow \mathit{SymbolicParallel}(\langle \mathbf{R_h}, \mathbf{R_l}, \mathbf{C_h}, \mathbf{C_l}, \mathbf{D} \rangle, \langle \mathbf{r_h}, \mathbf{r_l}, \mathbf{c_h}, \mathbf{c_l}, \mathbf{d} \rangle)$
    return $\langle \mathbf{R_h}, \mathbf{R_l}, \mathbf{C_h}, \mathbf{C_l}, \mathbf{D} \rangle$

$\mathit{SymbolicSeries}$( transistor $t$, $\langle \mathbf{R_h}, \mathbf{R_l}, \mathbf{C_h}, \mathbf{C_l}, \mathbf{D} \rangle$)
    $\mathbf{r_t} \leftarrow \mathit{MtbddITE}(\mathit{transparent}(t), [t.res], [\infty])$
    $\mathbf{r_h} \leftarrow \mathbf{R_h} + \mathbf{r_t} * ([1] + \mathbf{R_h}/\mathbf{R_l})$
    $\mathbf{r_l} \leftarrow \mathbf{R_l} + \mathbf{r_t} * ([1] + \mathbf{R_l}/\mathbf{R_h})$
    $\mathbf{c_h} \leftarrow \mathit{MtbddITE}(\mathit{transparent}(t), \mathbf{C_h}, [0])$
    $\mathbf{c_l} \leftarrow \mathit{MtbddITE}(\mathit{transparent}(t), \mathbf{C_l}, [0])$
    $\mathbf{d} \leftarrow \mathit{transparent}(t) \wedge \mathbf{D}$
    return $\langle \mathbf{r_h}, \mathbf{r_l}, \mathbf{c_h}, \mathbf{c_l}, \mathbf{d} \rangle$

$\mathit{SymbolicParallel}(\langle \mathbf{r_{h1}}, \mathbf{r_{l1}}, \mathbf{c_{h1}}, \mathbf{c_{l1}}, \mathbf{d_1} \rangle, \langle \mathbf{r_{h2}}, \mathbf{r_{l2}}, \mathbf{c_{h2}}, \mathbf{c_{l2}}, \mathbf{d_2} \rangle)$
    $\mathbf{r_h} \leftarrow \mathbf{r_{h1}} \parallel \mathbf{r_{h2}}$
    $\mathbf{r_l} \leftarrow \mathbf{r_{l1}} \parallel \mathbf{r_{l2}}$
    $\mathbf{c_h} \leftarrow \mathbf{c_{h1}} + \mathbf{c_{h2}}$
    $\mathbf{c_l} \leftarrow \mathbf{c_{l1}} + \mathbf{c_{l2}}$
    $\mathbf{d} \leftarrow \mathbf{d_1} \vee \mathbf{d_2}$
    return $\langle \mathbf{r_h}, \mathbf{r_l}, \mathbf{c_h}, \mathbf{c_l}, \mathbf{d} \rangle$
```

Fig. 15. DC-Value Computation in SirSim

D.3 Symbolic DC-Value Computation

Using symbolic algebra with MTBDDs, it is not difficult to construct a symbolic version of IRSIM's DC-value procedure. In SirSim, this operation is performed by *SymbolicComputeDC*, shown in Figure 15.

The tuple representation for each branch is unchanged except that each real-valued member is now an MTBDD, and the "driven" flag D is now a BDD: $\langle \mathbf{R_h}, \mathbf{R_l}, \mathbf{C_h}, \mathbf{C_l}, \mathbf{D} \rangle$.

As before, *SymbolicComputeDC* calls *SymbolicGetDC* to compute the tuple for the node of interest. However in the symbolic case, a node may be driven under one input assignment but floating under another, resulting in a non-constant **D** BDD. Therefore we must compute the DC voltage under both assumptions ($\mathbf{V_r}$, $\mathbf{V_c}$) and select the proper value for each input assignment using the *MtbddITE* operator. This voltage MTBDD is then converted into a logical BDD by *MtbddThreshold*.

The depth-first search is performed in *SymbolicGetDC*, utilizing the BDD **active** in the same manner as in *SymbolicAffectedNodes*.

```
*ComputeDelay*( node n, BDD DCVal )
    $\langle R, C\rangle \leftarrow GetTau(n, DCVal, \emptyset)$
    $delay \leftarrow R * C$
    return delay

*GetTau*( node n, BOOL DCVal, transistor via )
    if( $n \in \{Vdd, Gnd\}$ )
        $C \leftarrow 0$
        if( $n.value = DCVAL$ )
            $R \leftarrow 0$
        else
            $R \leftarrow \infty$
        return $\langle R, C\rangle$

    $R \leftarrow \infty$
    if( $n.value = DCVAL$ )
        $C \leftarrow 0$
    else
        $C \leftarrow n.cap$

    $\forall t \neq via$ such that $t.source = n$ or $t.drain = n$
        if( $t.source = n$ )
            $other \leftarrow t.drain$
        else
            $other \leftarrow t.source$
        if( $transparent(t)$ )
            $\langle r_o, c_o\rangle \leftarrow GetTau(o, DCVal, t)$
            $r_t \leftarrow$ t.res

            $r_b \leftarrow r_o + r_t$
            $c_b \leftarrow c_o * (r_o/r_b)$

            $R \leftarrow R \parallel r_b$
            $C \leftarrow C + c_b$
    return $\langle R, C\rangle$
```

Fig. 16. Delay Computation in IRSIM

SymbolicSeries is only slightly modified from the conventional version. It first computes a symbolic resistance value for the series transistor and then uses symbolic algebra to compute the same resistance values as before. In the conventional algorithm, *series* was only called for conducting transistors, so the capacitance values could simply be copied into the output tuple. Now however, the transistor may be conducting only for a subset of the input assignments, so we must use *MtbddITE* to remove capacitance values for assignments under which the transistor is not conducting. The only change to *SymbolicParallel* is the use of symbolic rather than conventional algebraic operations.

E. Delay Computation

To compute the delays associated with node value transitions, IRSIM uses the *ComputeDelay* algorithm shown in Figure 16.

To compute the Elmore delay, we require the resistance of the driving path(s), and the amount of capacitance to be charged or discharged. Like *GetDC*, *GetTau* performs a depth-first search through conducting transistors until it reaches Vdd or GND. However, since it has already computed the DC value for the transition, it need only collect resistance values for the driving paths and capacitances that require charging or discharging.

Besides this simplification, the other primary difference between *GetDC* and *GetTau* is the computation of branch capacitances. For Elmore delay calculation, the effective capacitance is obtained by scaling it by the factor r_o/r_b when translating it across a transistor. Again, the reader is referred to [5] for a complete discussion of this computation.

The symbolic version, shown in Figure 17 is derived quite naturally. The $\langle R, C\rangle$ tuple elements are replaced with

```
ComputeDelay( node n, BDD DCVal )
    ⟨R, C⟩ ← GetTau(n, DCVal, ∅, [1])
    T_delay ← R * C
    return T_delay

GetTau( node n, BDD DCVal, transistor via, BDD reached )
    if( n ∈ {Vdd, Gnd} )
        C ← [0]
        R ← MtbddITE(n.value ⊕ DCVal, [∞], [0])
        return ⟨R, C⟩

    R ← [∞]
    C ← MtbddITE(n.value ⊕ DCVal, [n.cap], [0])

    ∀t ≠ via such that t.source = n or t.drain = n
        if( t.source = n )
            other ← t.drain
        else
            other ← t.source
        reachother ← reached ∧ transparent(t)
        if( reachother ≠ false )
            ⟨r_o, c_o⟩ ← GetTau(o, DCVal, t, reachother)
            r_t ← MtbddITE(transparent(t), [t.res], [∞])

            r_b ← r_o + r_t
            c_b ← c_o * (r_o/r_b)

            R ← R || r_b
            C ← C + c_b
    return ⟨R, C⟩
```

Fig. 17. Delay Computation in SirSim

MTBDDs, and all real-valued computations are performed with symbolic algebra. We again require the **active** flag to control the recursion depth.

A number of enhancements to this algorithm are implemented in IRSIM and duplicated symbolically in SirSim. Both implement a separate delay calculation algorithm for charge-sharing transitions, and compute an additional delay term to account for the rise/fall time of the triggering transition.

F. An Example

In order to illustrate the different steps of the algorithm, we will work through an example evaluation of the domino AND gate shown in Figure 18. Assume that the first event in the event queue is $\langle Node = A, Time = 100, \mathbf{Value} = a, \mathbf{Mask} = a\rangle$, and node A's current value is 0. Further assume that the precharge proceeded as normal, that node B has already transitioned to value b, and the CCR has settled to a steady-state such that the internal node values are as shown.

After updating the state of node A, we first call *SymbolicAffectedNodes* to identify the nodes that may need to be re-evaluated. The search will begin at nodes $x1$ and P, and will return the set $(x1, P, x2)$. Note that no loops are detected in this simple example, so the **Broken** flag is set to $false$ for all transistors.

We next compute the new DC value for all nodes identified in the previous step. For the sake of brevity, the resultant tuple is only shown for the CCR output node P. $\mathbf{R_h}$, representing the pullup resistance is identically ∞, while the pulldown resistance $\mathbf{R_l}$ has value 15KOhms for $a \wedge b$, and is infinite otherwise. The capacitance MTBDDs $\mathbf{C_h}$ and $\mathbf{C_l}$ record the amount of capacitance connected to P that are charged high and low respectively. The driven function $\mathbf{D}$ shows that P is resistively driven only for $a \wedge b$.

Given this tuple, we compute the DC voltage of P using resistance and capacitance information separately, and then combine them using $\mathbf{D}$. A voltage divider equation shows that $\mathbf{V_r}$ is identically 0, since $\mathbf{R_h} = \infty$ and $\forall x, x/(\infty + x) = 0$. Applying the charge-sharing equation gives $\mathbf{V_c}$ as shown. We then use *MtbddITE* to separate the two cases, and then threshold the result to obtain the DC logic function $\overline{a \wedge b}$.

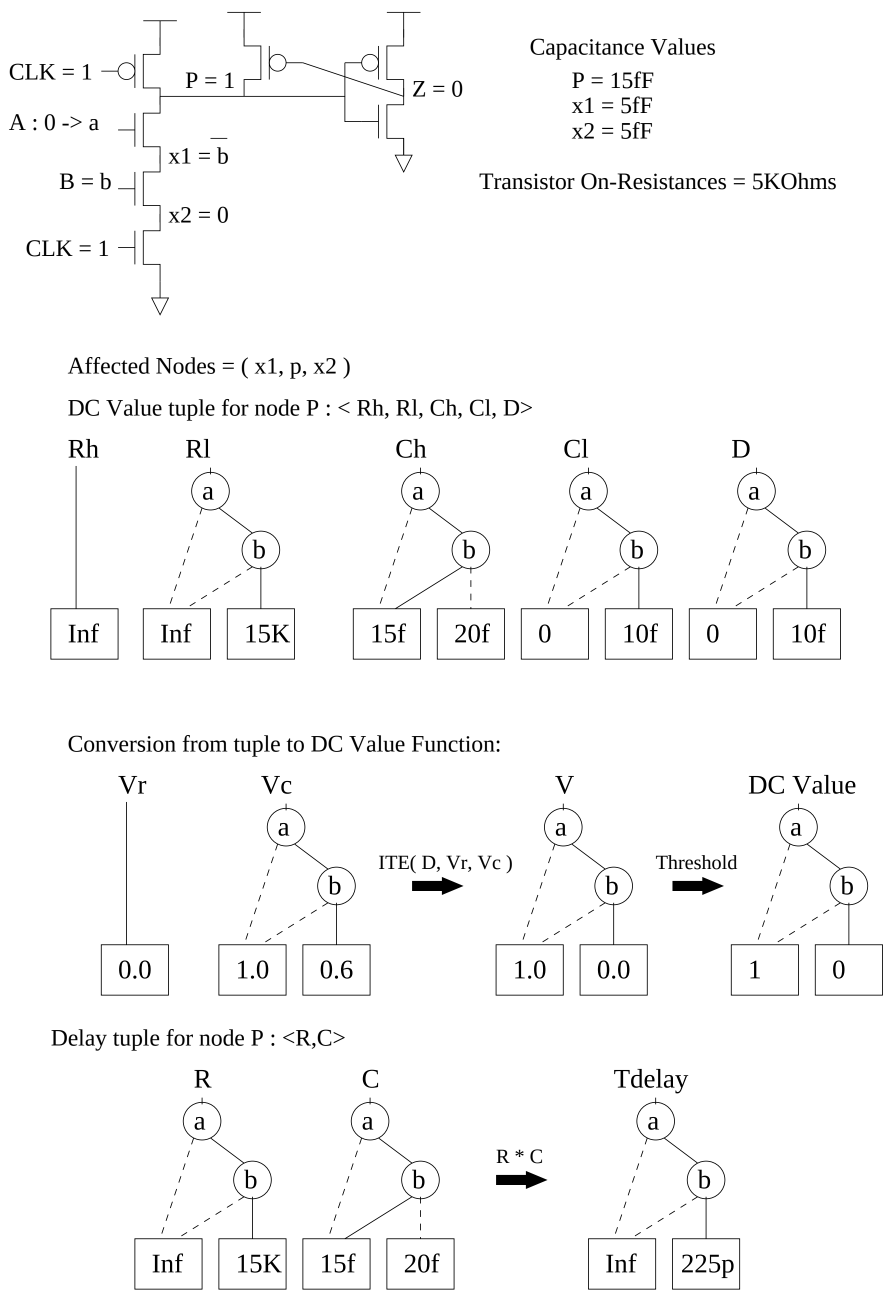


Fig. 18. Domino AND-Gate Example

Now we can compute the delay for the new transition. Note that we have depicted this operation as a product of a lumped R with a lumped C to avoid explicitly stepping through the recursion. In our implementation (and in the algorithms presented earlier), we compute a true Elmore delay, and would obtain the result $(5K * (5fF) + 5K * (5fF + 5fF) + 5K * (5fF + 5fF + 15fF)) = 200ps$.

Lastly, we schedule a resultant event for node P :

$$\langle Node = P, Time = 325, \textbf{Value} = \overline{a \wedge b}, \textbf{Mask} = a \wedge b \rangle$$

G. Ternary Simulation

The preceding discussion has assumed binary (0,1) simulation, when in fact both IRSIM and SirSim utilize ternary (0,1,X) node values. SirSim uses a dual-rail encoding like that used in Cosmos[4] and MOSSYM[2]. Two BDDs, value.**h** and value.**l**, are used to encode the symbolic ternary node value as follows:

$$value.\mathbf{h} = 1, value.\mathbf{l} = 0 \quad : \quad \text{HIGH}$$
$$value.\mathbf{h} = 0, value.\mathbf{l} = 1 \quad : \quad \text{LOW}$$

$$\begin{aligned} \mathit{value}.\mathbf{h} = 1, \mathit{value}.\mathbf{l} = 1 &\quad : \quad \text{UNKNOWN (X)} \\ \mathit{value}.\mathbf{h} = 0, \mathit{value}.\mathbf{l} = 0 &\quad : \quad \text{not defined} \end{aligned}$$

All nodes are initialized to X values at the start of simulation.

G.1 Ternary DC Value Computation

When the node connected to a transistor's gate has an unknown(X) value, its equivalent resistance can vary from the finite conducting value to ∞. This implies a min/max resistance, which can be computed for an nFET as :

$$\begin{aligned} \mathbf{R_{min}} &\leftarrow \mathit{MtbddITE}(t.gate.value.\mathbf{h}, [r_t], [\infty]) \\ \mathbf{R_{max}} &\leftarrow \mathit{MtbddITE}(t.gate.value.\mathbf{l}, [\infty], [r_t]) \end{aligned}$$

Similarly, since a node in the X state contributes an unknown amount of charge, min/max capacitances must also be computed:

$$\begin{aligned} \mathbf{C_{h\_max}} &\leftarrow \mathit{MtbddITE}(node.value.\mathbf{h}, [node.cap], [0]) \\ \mathbf{C_{h\_min}} &\leftarrow \mathit{MtbddITE}(node.value.\mathbf{l}, [0], \mathbf{C_{h\_max}}) \\ \mathbf{C_{l\_max}} &\leftarrow \mathit{MtbddITE}(node.value.\mathbf{l}, [node.cap], [0]) \\ \mathbf{C_{l\_min}} &\leftarrow \mathit{MtbddITE}(node.value.\mathbf{h}, [0], \mathbf{C_{l\_max}}) \end{aligned}$$

Now each term in the tuple representation is actually a pair of min/max MTBDDs. When performing series and parallel combinations of these tuples with ranges, an approximate solution is computed. The details of this approximation comprise a large part of Chu's thesis [5], and are beyond the scope of this article. However, the operations required involve straightforward algebraic manipulation and can be duplicated exactly in symbolic form without substantial difficulty.

To obtain the DC voltage from the tuple with ranges, the voltage divider and charge-sharing computations are slightly modified:

$$\begin{aligned} \mathbf{V_{r\_min}} &\leftarrow \mathbf{R_{l\_min}}/(\mathbf{R_{l\_min}} + \mathbf{R_{h\_max}}) \\ \mathbf{V_{r\_max}} &\leftarrow \mathbf{R_{l\_max}}/(\mathbf{R_{l\_max}} + \mathbf{R_{h\_min}}) \\ \mathbf{V_{c\_min}} &\leftarrow \mathbf{C_{h\_min}}/(\mathbf{C_{h\_min}} + \mathbf{C_{l\_max}}) \\ \mathbf{V_{c\_max}} &\leftarrow \mathbf{C_{h\_max}}/(\mathbf{C_{h\_max}} + \mathbf{C_{l\_min}}) \\ \mathbf{V_{min}} &\leftarrow \mathit{MtbddITE}(\mathbf{D}, \mathbf{V_{r\_min}}, \mathbf{V_{c\_min}}) \\ \mathbf{V_{max}} &\leftarrow \mathit{MtbddITE}(\mathbf{D}, \mathbf{V_{r\_max}}, \mathbf{V_{c\_max}}) \end{aligned}$$

Lastly, we must convert the voltage range into a dual-railed logical DC value. The normalized high and low voltage thresholds are user-specified, and must satisfy $0 < V_{low} \leq V_{high} < 1.0$. Voltages between V_{high} and V_{low} are considered X's. This conversion is implemented by the following operations:

$$\begin{aligned} DCVal.\mathbf{h} &\leftarrow \mathit{MtbddThreshold}(\mathbf{V_{max}}, V_{low}) \\ DCVal.\mathbf{l} &\leftarrow \overline{\mathit{MtbddThreshold}(\mathbf{V_{min}}, V_{high})} \end{aligned}$$

G.2 Delay Computation with Uncertainty

For delay computation in the presence of unknown node states, resistance ranges are again computed and dealt with in the same manner as above. However, capacitance requires a different treatment.

We maintain two MTBDDs, one for capacitance *potentially* charged high and the other for *potentially* discharged capacitors. This is similar to the method described for the binary version of *GetDC*, except that capacitance in the X state is added to both $\mathbf{C_{hx}}$ and $\mathbf{C_{lx}}$.

$$\begin{aligned} \mathbf{C_{hx}} &\leftarrow \mathit{MtbddITE}(node.value.\mathbf{h}, [node.cap], [0]) \\ \mathbf{C_{lx}} &\leftarrow \mathit{MtbddITE}(node.value.\mathbf{l}, [node.cap], [0]) \end{aligned}$$

IRSIM accounts for node capacitance in the transitions $\{0 \rightarrow 1, 1 \rightarrow \{X, 0\}, X \rightarrow \{X, 1, 0\}\}$, so we can now compute the switching capacitance as :

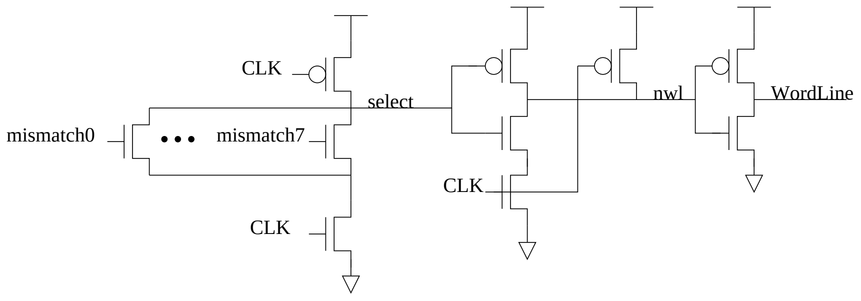


Fig. 19. Wordline Driver Example

$$\mathbf{C_{switch}} \leftarrow \mathit{MtbddITE}(\mathit{DCVal}.\mathbf{l}, \mathbf{C_{hx}}, \mathbf{C_{lx}})$$

Since SirSim does not currently allow event times to be time-ranges, we must select either the maximum or minimum delay possible for each input assignment. Following IRSIM's heuristic, we assume that the maximum delay is the conservative choice when switching to a well-defined value, while the minimum delay is selected when transitioning to an X. Note that the data-dependent delay variations are still accounted for, and this approximation only affects delays for input assignments where one or more node values are X's.

$$\begin{aligned} \mathbf{T_{min}} &\leftarrow \mathbf{R_{min}} * \mathbf{C_{switch}} \\ \mathbf{T_{max}} &\leftarrow \mathbf{R_{max}} * \mathbf{C_{switch}} \\ \mathbf{T_{delay}} &\leftarrow \mathit{MtbddITE}(\mathit{DCVal}.\mathbf{h} \wedge \mathit{DCVal}.\mathbf{l}, \mathbf{T_{min}}, \mathbf{T_{max}}) \end{aligned}$$

III. Application

Symbolic Timing Simulation has applications to both functional and timing verification of transistor-level circuits. While it is substantially more compute intensive than static analysis techniques, it is applicable to a much broader family of circuits.

To verify a block containing arbitrary circuit structures, we simply perform a symbolic timing simulation while monitoring the Boolean functions on the output and state nodes. A specification of the correct output function must be supplied by the user or extracted from the RTL description of the block. 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 as well. 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 an error is detected, a counterexample is generated and the simulation can be run again with non-symbolic inputs to aid in debugging.

We should note that our verification is limited at present to a model having specific delay values for each input pattern, rather than delay ranges that result from process variation and other sources of uncertainty. This is a direct result of our modelling of node transition events as instantaneous. Static techniques, on the other hand, are well suited to analysis based on time ranges, and thus can be more provably conservative. We are looking into extending STS by incorporating delay ranges in the form of min/max delay values. This can be accomplished either by explicitly modelling events as time ranges, or by scheduling additional transitions to an X value at the minimum delay point. The latter approach is substantially easier to implement, but is slightly more pessimistic.

A. Functional Verification

Timing and functionality cannot be easily separated for many circuits, causing problems for methodologies which perform these analyses independently. In some cases, timing information is required simply to model circuit functionality properly.

Consider the example shown in Figure 19 which was used as part of a row-decoder and RAM wordline driver. It is composed of a dynamic NOR stage followed by a static NAND gate. The NAND gate is intended to keep the wordline de-asserted during precharge of the NOR stage. If any of the *mismatch* lines are high when *CLK* rises, then *select* should fall quickly enough to keep *nWL* from going low. The safety of this circuit is ensured by the relative loading of *nWL* and *select*. Since the *WordLine* driver is very large, *nWL* is heavily loaded and will not switch nearly as quickly as *select*.

Existing static functional verification methodologies such as equivalence-checking use either zero-delay or unit-delay models of timing behavior. Zero-delay analysis will fail to model the storage of charge (and thus state) on the precharge node. However, unit-delay analysis assumes that all transitions require the same amount of time, resulting in a unit-glitch on *WordLine*. Capturing the intended behavior of this circuit requires proper modelling of the relative stage delays. Since SirSim implements an Elmore delay model and uses inertial delays, it computes the correct sequence of events for this circuit.

B. Timing Verification

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 internal structures, such as latches, dynamic gates, and self-timed loops.

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 the 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.

However, a symbolic timing simulator need only simulate the circuit and check for correct functionality. Since it avoids having to explicitly determine timing constraints based on circuit topology, it is much more robust with respect to variations from standard design styles.

A further advantage of STS is that its output arrival times will be more accurate than those computed by a timing analyzer, given the same delay computation model. One reason for this is that the simulator is not susceptible to false paths, because their effects will be eliminated by a dynamic sensitization criterion. McGeer demonstrates that the dynamic criterion cannot underestimate the true circuit delay, while the static criterion can [10]. While the dynamic criterion 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 criterion 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 stores the current state of every node in the circuit, it can avoid this pessimism if its delay model correctly accounts for these effects.

C. Complexity

Since we are performing a complete analysis over all possible input combinations without any loss of information, the worst-case complexity of symbolic timing simulation is necessarily exponential in the number of inputs to 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 due to significant amounts of BDD subgraph sharing.

As in any simulation-based approach, runtime complexity can be difficult to evaluate because it is affected by a number of factors. However, it is a natural metric and is of paramount importance to potential users. In general, we have found runtime rather than memory usage to be the limiter for symbolic timing simulation.

STS runtime is affected primarily by the number of events generated and the efficiency of the symbolic encoding. The number of events is in turn dependent on the circuit topology, depth of the logic cones, latching strategies, and circuit size. Of course, if the circuit is astable (3 inverter loops), the simulation may never even terminate. In general, given the same number of transistors, deeper logic cones will generate more events because of the larger number of potential delay paths. Edge-triggered flip-flops help control the event count by resynchronizing multiple events that arrive at its inputs.

Probably the most important factor affecting runtime is the efficiency of the symbolic encoding, which is strongly dependent on the variable order selected for the BDDs and MTBDDs. For some circuits, no sub-exponential variable order exists.

In practice, we have seen runtimes on the order of several minutes for most circuits up to ≈ 10000 transistors. With further algorithmic improvements and tuning, this might be pushed as high as 50-100 K transistors, which is sufficient to handle sub-blocks typically assigned to single designers. Above this size, the key to utilizing STS may be extensions capable of generating abstractions of the timing interfaces suitable for use by a static timing analyzer at a higher level of the hierarchy.

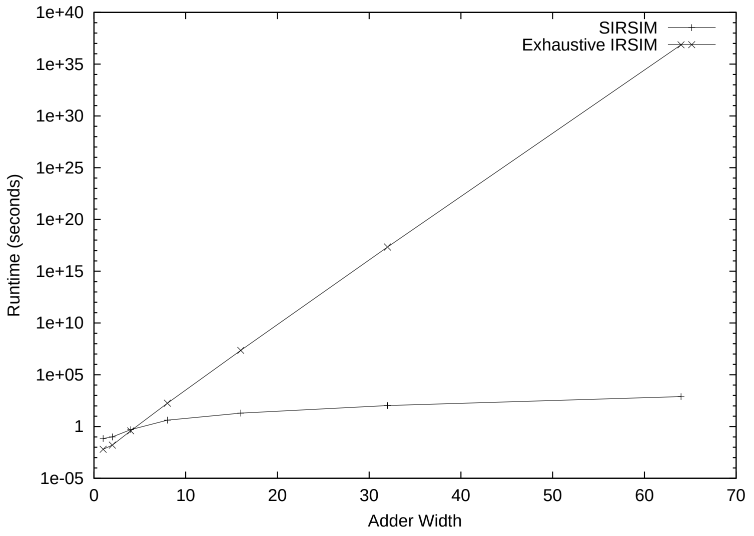


Fig. 20. Runtime vs. Adder Width

IV. Experimental Results

A. Adders

For our first substantial test cases, we ran SirSim on Manchester carry-chain adders[8] of varying widths. 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 excellent, and in fact only grew as the cube of the adder width. They are plotted in Figure 20, along with the time required to perform the equivalent 2^{2n} conventional IRSIM runs, for adders of width $n = 1 \ldots 64$. For the 32-bit adder, SirSim performed a complete analysis in less than 2.5 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 20 minutes, and achieved a speedup of 10^{33}. While the absolute speedup values are so large as to be almost meaningless and can be made arbitrarily large by increasing the size of the circuit, it is worth noting that a previously infeasible analysis on realistically sized adders is now possible.

The following analysis justifies this 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 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 $E = l \sum_{i=1}^{n} i \cdot n = O(n^2)$,and the total runtime $T = EC = O(n^3)$.

In order to evaluate SirSim on a more realistic adder implementation, we also simulated varying widths of a carry bypass design. The runtimes (shown in Table I) were slightly worse 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.

B. 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. Note that SirSim can simulate sequential circuits but our goal was to determine performance on a sample of combinational circuits. For these experiments, we used a simple depth-first-search ordering heuristic for the BDD and MTBDD variables. As can be seen from Table I, 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

TABLE I
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	0.6	0.9
s349	654	24	0.8	1.0
s382	682	24	2.9	2.7
s444	758	24	4.6	6.2
s820	1786	23	4.3	3.9
s1196	2456	32	1.1	5.6
s1238	2574	32	1.6	6.5
s1423	2996	91	0.7	1.3
s1494	3902	14	0.8	1.4
s5378	8902	214	3.6	12.65
sr_incr64	4218	129	39.8	40.5

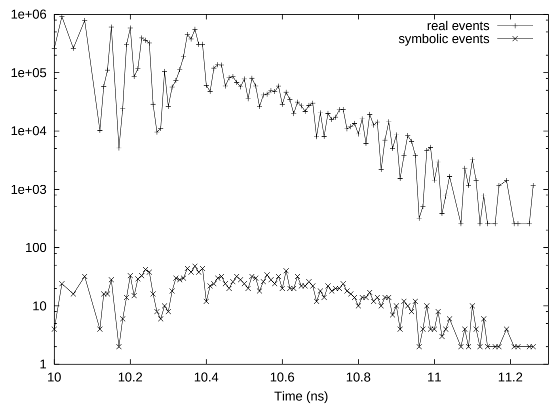


Fig. 21. Symbolic and Real Events

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.

C. Self-Resetting Incrementer

We also implemented a self-resetting 64-bit incrementer designed at IBM [9]. This circuit makes use of self-timed locally-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 rather complicated, special-purpose 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 uses an inertial delay model, its verification power is virtually identical to these checks. This example clearly demonstrates the power of SirSim to handle even highly customized circuit design styles. Furthermore, SirSim's runtime for this 4200-transistor design was around 40 seconds (*sr_incr64* in Table I).

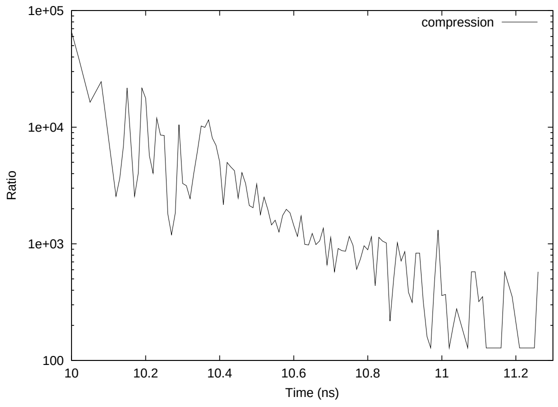


Fig. 22. Symbolic Compression vs. Time

D. *Symbolic Event Compression*

Of perhaps greater theoretical interest are the traces shown in Figures 21–22. In order to gain some intuition into the speedup attained by symbolic timing simulation, we compared the total number of symbolic events per timestep with the total number of real events. We define the real event count as the sum of all events that would occur in a given timestep in an exhaustive conventional simulation suite. This was computed by examining the don't-care sets of the symbolic event trace.

For circuits such as adders, which have highly data-dependent delays, we did not know if each symbolic event would be able to capture a significant number of real events. However, as Figure 22 shows, it was quite successful, resulting in an average symbolic compression (ratio of real events to symbolic events) of over 9600 for the 8-bit adder. This compression increases exponentially with the adder width.

V. Conclusions and Future Work

We have presented an extension to symbolic simulation that properly accounts for general data-dependent logic delays. We have also shown a symbolic procedure for computing data-dependent delays based on the conventional transistor-level timing simulator IRSIM. Our experimental results indicate that symbolic timing simulation can be used to verify timing and functionality for circuits of reasonable size.

Ongoing research is investigating more efficient event-management algorithms and higher-accuracy delay-calculation methodologies. We are also looking into ways of extracting timing models using symbolic timing simulation, which could be used to integrate this new approach into existing static analysis design-flows.

References

[1] 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 188–191, November 1993.

[2] R. E. Bryant. Symbolic Verification of MOS Circuits. *1985 Chapel Hill Conference on VLSI*, pages 418–438, 1985.

[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. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, CAD-6(4):634–639, July 1987.

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

[6] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Certified Timing Verification and the Transition Delay of a Logic Circuit. *Proceedings of the Design Automation Conference*, 1992.

[7] W. Elmore. The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. *Journal of Applied Physics*, January 1948.

[8] L. A. Glasser and D.W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.

[9] 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.

[10] P. C. McGeer and R. K. Brayton. Efficient 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] I. M. Obfuscated. Symbolic Functional and Timing Verification of Transistor Level Circuits. *ACM/IEEE International Conference on Computer Aided Design*, 1999.

[14] J. Rubinstein, P. Penfield, and M. A. Horowitz. Signal Delay in RC Tree Networks. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 2(3):202–211, July 1983.

[15] A. Salz and M. A. Horowitz. IRSIM: An Incremental MOS Switch-level Simulator. *Proceedings of the Design Automation Conference*, pages 173–178, June 1989.

[16] 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.

[17] F. Somenzi. *CUDD: CU Decision Diagram Package - Release 2.2.0, Online User Manual*, May 1998.
[18] C. J. Terman. RSIM - A Logic-Level Timing Simulator. *International Conference on Computer Design*, pages 437–440, October 1983.