On the Reuse of Symbolic Simulation Results for Incremental Equivalence Verification of Switch-Level Circuits

Ll. Ribas-Xirgo J. Carrabina-Bordoll
Microelectronics Group, Computer Science Dept.
Universitat Autònoma de Barcelona, UAB
08193 Bellaterra, Catalunya, Spain

Abstract

Incremental methods are successfully applied to deal with successive verifications of slightly modified switch-level networks. That is, only those parts affected by the changes are symbolically traversed for verification. In this paper, we present an incremental technique for symbolic simulators which is inspired in both existing incremental techniques for non-symbolic simulators and a token-passing mechanisms in Petri nets.

1 Introduction

Circuit designs suffer a lot of modifications before they meet their specifications. Usually, those changes consist of simple insertions or deletions of a reduced set of nodes and devices. Their effect on the circuit behavior is usually checked by simulation, and their results may drive the designer to introduce further changes in the circuit, thus, repeating the whole process again. The cost of the simulation can be reduced by only considering the changes introduced in the original circuit. That is, only those parts which are affected by a circuit change are fully simulated obtaining data from the previously existing parts whenever necessary.

In fact, incremental techniques used on conventional simulators are based on the assumption that circuits undergo a lot of small changes during their design, hence requiring a proportional effort for their simulation. Obviously, the type of changes, the size of the affected subnetwork, and the efficiency of the storage methods must be correctly weighted before running a full simulation or doing it in an incremental form [1].

At the switch-level, schematics are often modified with meeting performance requirements in mind, but functionality must be preserved along any series of changes. Therefore, a number of simulations are run not only to determine the electrical characteristics of the circuit but to check its function. Unfortunately, circuit simulators are not readily prepared for those kind of checks. Consequently, the verification of circuit functions should be left to specialized tools.

In this paper, we present one such tools that is able to derive the Boolean function of a switch-level network through an event-driven symbolic simulator. In our approach, the incremental method is rather similar to that of conventional simulators. However, we have had to introduce a "back-evaluation" technique to solve the problem of event ordering posed by deletions of devices involved in self-loops. Note that each transistor requires two of them for representing source and drain behavior. The back-evaluation forces the activation of the remaining devices involved in the affected self-loops to avoid additional, costly procedures for monitoring the event queue and nodes to ensure their coherence. We shall show that our solution has a small impact on the complexity of the final algorithm.

A straightforward application in formal verification of switch-level designs would compare the results obtained in one run to the ones from the previous one, or to the functional specifications of the circuit to check their equivalence [2, 3, 4]. Results would also be reused in subsequent design changes. Other applications such as model generation [5], circuit simplification by clock abstraction [6], and even test generation [7, 8, 9] may also take advantage of faster repeated simulations of incrementally modified circuit versions.

In the following, we first introduce some definitions and notations used further in the paper. Then, we shall depict a general scheme of incremental simulation tools. Finally, we describe our approach for symbolic simulators and conclude with some execution results.

2 Definitions and Notation

We assume the reader is familiar with Boolean functions and Boolean networks (BNs), so we focus on the definitions and notation which are new or particularly used in this text.

2.1 Circuit Representation

Extended Boolean networks are graphs EBN(V, E) that represent logic circuits where each vertex $v_i \in V$ corresponds to a logic gate implementing function f_i and each edge, to a physical connection. The *primary inputs* (PI) and the *primary outputs* (PO) are distinct subsets of V.

Because EBNs cannot efficiently handle multi-outputs devices such transistors, we use more generic *circuit graphs* (CGs). A CG is a directed bipartite graph CG(V, E) where V can be decomposed into two disjoint subsets, N and D, and $E \subseteq (N \times D) \cup (D \times N)$.

N is the set of (electrical or logical) nodes that can hold a value. Values of nodes $n_i \in N$ are represented by Boolean variables y_i.

D is the set of devices $d_i \in D$ that are associated Boolean functions $\phi_i(\underline{z}) : B^{|\mathrm{adj}^-(d_i)|} \mapsto B^{|\mathrm{adj}^+(d_i)|}$, where B is the carrier of a Boolean algebra, $\underline{z}$ is a vector of variables from $\sup(\phi_i) = \{y_j \mid n_j \in \mathrm{adj}^-(d_i)\}$, and $|\mathrm{adj}^-(d_i)|$ and $|\mathrm{adj}^+(d_i)|$ denote the number of ingoing and outgoing edges of d_i, respectively.

As implicitely defined before, the set of adjacent input vertices of $v \in V$, $\mathrm{adj}^-(v)$, contains every vertex for which vertex v is the last endpoint in an edge. Conversely, the set of adjacent output vertices of v is $\mathrm{adj}^+(v) = \{w \in V \mid \exists (v, w) \in E\}$. Note that the adjacency sets of a vertex belonging to subset N are subsets of D, and vice versa.

Figure 1 shows a NMOS transistor as seen in a CG with nodes as black dots in contrast to its representation in an EBN, with self-loops made evident.

A *path in a CG* is a sequence of edges with common endpoints except the first and the last ones. We define paths in terms of only nodes or devices. For instance, a *node path* $R = (n_{i_0}, \ldots, n_{i_{L-1}})$ is a sequence of nodes n_{i_k} connected through devices.

Although fanin(v) and fanout(v) would correspond to $\mathrm{adj}^-(v)$ and $\mathrm{adj}^+(v)$ regarding the relation between EBNs and CGs, we shall redefine fanin and fanout concepts for CGs as follows.

The *fanin of a node* $n_i \in N$, denoted by fanin(n_i), is defined as the set of nodes from which there is a node path of length unity to n_i. And the *fanout of a node* $n_i \in N$, fanout(n_i), is similarly defined.

Furthermore, the *fanin and fanout cones* of a vertex $v \in V$ (node or device) refer to the set of vertices reachable from it. Again, we can exclude from this set vertices of one class or another by specifying the type of its elements. The concept of reachability means that there exist a path between v and every element in the set. Note that, more formally, these sets are also called *transitive fanin* and *transitive fanout*.

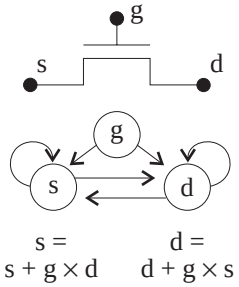

Figure 1: N-type transistor representations.

2.2 Nodal Functions

In symbolic circuit analysis, nodes act as function placeholders. That is, they contain a Boolean function representation that encodes their value for any possible combination of input (PI) and present state (PS) variables of the circuit.

Binary values (0 and 1) are not enough to correctly model some of the common effects occuring at the switch level, thus we extend the number of discrete values to four: low (L), unknown (X), high (H) voltage level, and high impedance state (Z). This extension still has problems with charge sharing because it might need more intermediate logic levels.

In this work, nodal functions, f_i, are Boolean functions of a four-valued Boolean algebra ($B_4 = \{Z, L, H, X\}, +, \cdot, Z, X$) that are uniquely characterized by two binary Boolean functions [10], i.e. any f_i is represented by a pair of functions (f_i^1, f_i^0).

The *on-set function* (f_i^1) indicates the cases for which node n_i is driven to high voltage level, and the *off-set function* (f_i^0) determines the conditions, or variable combinations, for which the node is electrically connected to ground.

The former functions are derived from an event-driven simulation algorithm [11] in which starting events are determined by the PI/PS nodes. The rest of events is caused by device activity, according to their models, which are described below.

2.3 Device Models

In switch-level circuits, transistors process symbolic information as it flows through the network. Their behavior is reduced to a simple switch, i.e. it can be *off* completely isolating both end nodes or *on*, letting data flow in both directions. Hence, transistors are considered bi-directional devices that modify both drain and source nodal functions. The complete set of equations

$\Phi = (\phi_d, \phi_s)$ for a PMOS switch is as follows.

$$\begin{aligned} \phi_d(f_d, f_g, f_s) &= f_d(\underline{x}) + f_g^0(\underline{x}) \times f_s(\underline{x}) \\ \phi_s(f_d, f_g, f_s) &= f_s(\underline{x}) + f_g^0(\underline{x}) \times f_d(\underline{x}) \end{aligned} \quad (1)$$

where products $f_g^0 \times f_s$ and $f_g^0 \times f_d$ are scalar products of functions by a vector of functions. Note that only two device functions are required because gate nodal function is not modified by the transistor activity.

A similar system describes the behavior of a NMOS switch, with the off-set function of the gate substituted in (1) by the on-set function.

Independent voltage sources also play an important role because they are assumed to be attached to PIs and PSs, hence identifying them. They supply a constant value from B_4 or a variable to the network. As they not require any input, their model can be simply described as

$$\phi_v(I(\underline{x})) = x_v = (x_v^1, x_v^0) \quad (2)$$

where $I(\underline{x})$ is the identity function, $\underline{x}$ is a vector of Boolean variables that identify PI and PS nodes, and n_v is the non-ground end of the device.

Derived models, ϕ_b, from transistor subcircuits have equations of the following form:

$$\phi_b(\langle\{y_o\} \cup \mathrm{adj}^-(d_b)\rangle) = f_o(\underline{x}) + \phi_b{}'(\langle \mathrm{adj}^-(d_b)\rangle) \quad (3)$$

where a single output n_o is assumed, and input vector is an ordered set (denoted by angle brackets) of $\mathrm{adj}^-(d_b)$. Note that nodal functions are never set back to their initial value ($f = Z$), but eventually added new functions ($\phi_b{}'$).

3 Incremental Simulation Overview

The incremental simulation is a technique based on the assumption that most of the data from previous simulations can be reused. The main problems are to detect what design changes have been done in a circuit, and to determine which part of the circuit is affected by those changes.

In previous works [1, 12, 13, 14], the incremental update is done by performing a full simulation in the previously obtained fanout cone of a (modified) net.

The incremental simulation procedure for an original circuit $\mathrm{CG}(V, A)$ is sketched below.

– 1. Initialize event queue Q with input nodes.
– 2. Simulate CG with Q.
– 3. Modify circuit, i.e. generate a new one, CG′
– 4. Calculate $\mathrm{CG}' \setminus \mathrm{CG}$.
– 5. Insert new events from nodes in $\mathrm{CG}' \setminus \mathrm{CG}$.
– 6. Simulate CG′ with Q.
– 7. Go to step 3 while check of CG′ fails.

The calculation of $\mathrm{CG}' \setminus \mathrm{CG}$ can be done by recording designer's operations (in step number 3) or by specific algorithms. In the first case, a lot of insert/delete operations with no effect on the resulting circuit need be accounted for. Therefore, they should be collapsed before generating $\mathrm{CG}' \setminus \mathrm{CG}$ to save storage space in the meanwhile. The need for a special process for this data gives validity to the second option: after the designer has saved his/her design, a program calculates the graph difference between the previous circuit and the current one.

The simulation of CG′ is done by inserting the new contents of vertices in $\mathrm{CG}' \setminus \mathrm{CG}$ into the event queue and starting the whole process with it. Data required by elements not present in $\mathrm{CG}' \setminus \mathrm{CG}$ has been previously stored during the simulation of CG and is retrieved upon request.

In [1], a token-passing algorithm is used to traverse the circuit graph and determine de fan-out cone of the vertices that have been changed in any way (inserted, deleted, or with its contents modified). Once it is done, simulation is performed only in the marked vertices, and selective data retrieval is carried on. That is, data from previous simulation can be partially recovered to avoid excessive memory occupancy. Partial data retrieval is done by slicing $\mathrm{CG}' \setminus \mathrm{CG}$ by levels or by clusters of strongly dependent gates.

4 Symbolic Incremental Simulation

Much of the material presented here refer to gate level simulations for the sake of simplicity, assuming that gate models are directly created from their transistor network descriptions. Aspects that concern switch level descriptions will be conveniently outlined.

Let us go over each step in the previous incremental simulation scheme, assuming simulation of the original circuit is already done.

4.1 Circuit Updates

Any circuit modification may be viewed as an element insertion or deletion to/from V which produces V'. Each of these actions is reflected as a series of extra commands in the original circuit netlist description: an insertion is performed by adding a new element, while a deletion is an extension to the (usual) capabilities offered by the base language.

In this work, SPICE [15] is used, and device elimination is performed by an extra "*-" command card, where a list of devices is specified. Nodes are automatically deleted when they are totally disconnected from the rest of the circuit.

For every new device $d_j \in V'$, each new adjacent nodal function is set to constant function Z, and an arbitrary input node is enqueued to cause its model

equation execution. Nothing else is required because any further change it provokes in its fanout cone will cause subsequent events, and so forth.

Deleting an existent device $d_j \in V$ or changing its function for another requires resetting its output nodes and insert the new nodal functions (constant function Z) into the event queue to account for the changes.

However, resetting the contents of d_j's outputs may not be enough to clear off its effect on the rest of the circuit because existent nodal functions are added other functions but never reset or set to different functions. Figure 2 illustrates such a situation on a EBN (each vertex is a device and its only output). In that network, deleted vertex c affects vertices d and e which will take into account a reset on c due to its deletion. Unfortunately, the functions stored in d and e will not be reset but added the result of ϕ_d and ϕ_e with input $f_c = Z$. That is, they will possibly contain erroneous results.

To prevent this from happening, when a node is reset to Z, a reset token must be propagated to other nodes in its fanout. That is, the output of deleted vertices v are marked for resetting vertices in fanout(v). That is, reset$(v) \leftarrow$ TRUE, where reset(v) is a flag that indicates whether a reset of f_v is required before evaluating ϕ_v.

This mechanism avoids traversal of the graph to calculate the fan-out cone of a node that has been reset.

In case a deleted vertex v has an output vertex u that is an input too, i.e. fanout$(v) \cap$ fanin$(u) \neq \emptyset$, the former reset-propagation mechanism would reset the contents of u, but does not ensure this operation be done before other $w \in$ fanout(v) because of the queuing policy used in our symbolic simulation approach. Therefore, to prevent other vertices from using non-previously reset values of u, it must be immediately recalculated from vertices in fanin(u). We call this operation *back-evaluation of* u because it is evaluated right before its use in valuing any other vertex contents in the simulation loop.

In Fig. 3 there is an example of an EBN that contains a node requiring back-evaluation. Vertex a must have a correct value (without the influence of the deleted vertex c) calculated from its fanin vertices $b_1, \ldots, b_N$. By doing so, any node in the fanout cone of c will be correctly set by the normal reset-propagation procedure, regardless of the order of valuation.

Although the back-evaluation may not be required in a gate-level circuit, it is often used in switch-level circuits because of the structure of the transistor models (see Fig. 1). For instance, taking the PMOS transistor out of the EBN in Fig. 4 implies back-evaluating vertex d. Note that vertices g and d are not deleted but arcs (g, s_p) and (d, s_p). Vertex s_p is marked for reset-propagation and its contents set to Z, but never withdrawn from the corresponding CG.

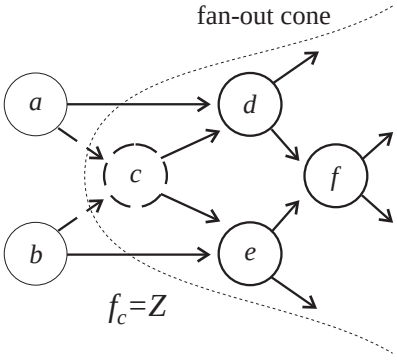

Figure 2: Vertex deletion.

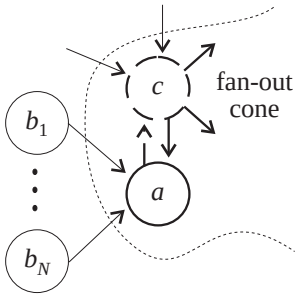

Figure 3: Back-evaluation of a is required to eliminate previous influence from c.

4.2 Simulation with Reset Propagation

A slight modification in the usual event-driven scheme is necessary to cope with reset propagation, while back-evaluation is only considered when deleting a device (see Fig. 5).

As previously outlined, incremental symbolic event-driven simulation has only been added a check for reset propagation (see Fig. 6). The advantages of this algorithm come from the fact that avoids extra graph traversals, apart from the one done while simulating it.

5 Results and Conclusion

In this paper we have adapted the incremental technique used in numerical simulators (as opposed to symbolic ones) to diminish the complexity of repeated

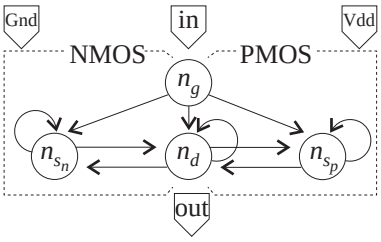

Figure 4: Circuit graph of a CMOS inverter.

```
CG = deleteDevice( CG, d )
    for each n_j ∈ adj+(d) do
        f_j ← Z
        reset(n_j) ← TRUE
        adj−(n_j) ← adj−(n_j) \ {d}
        if n_j ∈ adj−(d) then
            /* Back-evaluate n_j */
            for each d_i ∈ adj−(n_j) do
                apply device model Φ_i(z)
            enddo
        endif
        Q ← Q ∪ {f_j}
    enddo
    for each n_j ∈ adj−(d) do
        adj+(n_j) ← adj+(n_j) \ {d}
    enddo
    CG ← CG \ {d}
end // deleteDevice
```

Figure 5: Algorithm for device deletion.

symbolic simulations of a series of circuit versions.

The incremental method here presented does not require any previous analysis of the circuit but dynamically decides which elements must be resimulated and inserts them into the event queue. The simulation algorithm is only slightly changed to introduce a test for reset tokens that are caused by deleted components. A special step (back-evaluation) must be added to remove the effects of deleted components in case they would not be reset by the previous reset-propagation technique.

The size of the circuits to be analyzed is limited by both the size of the BDDs that represent their functions and the disk space required to store the circuit information between two simulations. As for the last one, note that it is possible to reduce the quantity of nodes by making an extensive use of the hierarchy ex-

```
incrSimulate( CG, CG \ CG′ )
    init. event queue, Q ← ∅
    for each node n_i ∈ CG \ CG′ do
        f_i ← Z
        Q ← Q ∪ {f_i}
    enddo
    while Q ≠ ∅ do
        dequeue next event, Q ← Q \ {f_k}
        if reset(n_k) = TRUE then
            reset(n_k) ← FALSE
            for each n_j ∈ fanout(n_k) do
                f_j ← Z
                reset(n_j) ← TRUE
            enddo
        endif
        for all devices d_i | d_i ∈ adj+(n_k) do
            apply device functions Φ_i(z)
            Q ← Q ∪ {f_j | f_j ∈ adj+(d_i)}
        enddo
    enddo
end // incrSimulate
```

Figure 6: Incremental symbolic simulation.

cept for the critical parts of the design that must be completely checked. However, in this work we were more interested in validating the reset-propagation algorithm before trying to find a good technique to save data storage space.

The results of the incremental symbolic simulation technique here presented for several circuits are shown in Table 1. All circuits are described in a hierarchical form in which the gate-level view is made of subcircuits described at the switch level. Combinational circuits ("c<number>") have a lot of reconvergent paths [16] except those whose names end with "nr" (they have redundancies removed [17]). Sequential circuits [18] ("s<number>") have been added as much state variables as flip-flop outputs they contain, so to account for allowed internal state nodes.

Column "♯C" indicates the number of changes that have been done on the corresponding circuit. Circuit updates are randomly chosen and consist of function (i.e. exchanging an AND gate for a NOR gate) and/or structural (i.e. swapping lines or deleting gates) modifications.

The number of events (column "♯Evt") and the processing time (column "t_{CPU}") are, as expected, much smaller after a design change than for the original circuit. However, the reduction of time may differ from

Table 1: Incremental simulation results (Sun-Ultra 1)

Circuit	Original		+ Design changes		
	t_{CPU}	♯Evt	♯C	t_{CPU} (σ)	♯Evt (σ)
c432nr	0.3s	265	140	0.05s (0.15)	21 (30)
c432	0.4s	275	140	0.05s (0.18)	21 (31)
c499nr	8.5s	258	45	4.59s (9.22)	24 (25)
c499	9.2s	266	22	2.56s (4.47)	23 (24)
c880	67.8s	619	15	15.94s (27.36)	23 (23)
c1355	9.8s	754	15	2.14s (5.05)	37 (67)
c1908	14.4s	1306	816	0.20s (0.97)	11 (14)
s510	1.3s	441	46	0.03s (0.13)	12 (2)
s1196	0.3s	880	57	0.02s (0.04)	17 (21)

change to change. To emphasize this aspect, we include enclosed by parentheses the standard deviation of these numbers along simulations. The apparently strange behavior of c880 (ALU + Control) is due to the complex BDD required to represent functions involved in the ALU operations. It is also important to note that the cost of each change would be approximately the same than the cost for the original circuit if not using any incremental method.

In a nutshell, results show that a significant reduction in time is achieved in all checked circuits. Of course, results are strongly dependent on the circuit behavior and on the fault or design change being considered, as indicated by the standard deviation figures shown. However, on average, the number of events is reduced at a small 5% of the first run, and successive simulations take five times less CPU time.

References

[1] S. Hwang, T. Blank, and K. Choi, "Fast functional simulation: An incremental approach," *IEEE Trans. on CAD/IC*, vol. 7, July 1988.

[2] S. Malik, A. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *ICCAD*, pp. 6–9, 1988.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking," in *27th ACM/IEEE Design Automation Conference*, pp. 46–51, 1990.

[4] Y. Kukimoto, M. Fujita, and H. Tanaka, "Symbolic verification of CMOS synchronous circuits using characteristic functions," in *CICC*, p. 11.5, 1991.

[5] R. E. Bryant, "Extraction of gate level models from transistor circuits by four-valued symbolic analysis," in *ICCAD*, pp. 350–353, 1991.

[6] S. Jain, R. E. Bryant, and A. Jain, "Automatic clock abstraction from sequential circuits," in *32nd ACM/IEEE DAC*, pp. 707–711, 1995.

[7] K. Cho and R. Bryant, "Test pattern generation for sequential MOS circuits by symbolic fault simulation," in *26th DAC*, pp. 418–423, 1989.

[8] S. Srinivasan, G. Swaminathan, J. Aylor, and M. Mercer, "Algebraic ATPG of combinational circuits using binary decision diagrams," in *European Test Conference*, pp. 240–248, 1993.

[9] C. Bolchini, F. Fummi, R. Gemelli, and F. Salice, "A BDD based algorithm for detecting difficult faults," in *ISCAS'95*, pp. 2015–2018, 1995.

[10] L. Ribas, R. Peset, and J. Carrabina, "Two-rail switch-level symbolic analysis," in *3rd Int'l Workshop on Symbolic Methods and Applications to Circuit Design (SMACD)*, pp. 307–314, 1994.

[11] L. Ribas and J. Carrabina, "Analysis of switch-level faults by symbolic simulation," in *32nd ACM/IEEE DAC*, pp. 352–357, 1995.

[12] K. Choi, S. Hwang, and T. Blank, "Incremental-in-time algorithm for digital simulation," in *25th ACM/IEEE DAC*, pp. 501–505, 1988.

[13] W. Cheng and M. Yu, "Differential fault simulation - a fast method using minimal memory," in *26th ACM/IEEE DAC*, pp. 424–428, 1989.

[14] J. Lee and D. Tang, "An algorithm for incremental timing analysis," in *32nd ACM/IEEE DAC*, pp. 696–701, 1995.

[15] E. Cohen, A. Vladimirescu, and D. Pederson, *User's Guide for SPICE*. Univ. of California, March 1979.

[16] F. Brglez and L. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN," in *ISCAS*, pp. 662–698, 1985.

[17] G. Tromp and A. van de Goor, "Logic synthesis of 100-percent testable logic networks," in *ICCD*, 1991.

[18] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *ISCAS*, pp. 1929–1934, 1989.