

# Transforming Cyclic Circuits Into Acyclic Equivalents

Osama Neiroukh, Stephen A. Edwards, *Senior Member, IEEE*, and Xiaoyu Song, *Senior Member, IEEE*

**Abstract**—Designers and high-level synthesis tools can introduce unwanted cycles in digital circuits, and for certain combinational functions, cyclic circuits that are stable and do not hold state are the smallest or most natural representations. Cyclic combinational circuits have well-defined functional behavior yet wreak havoc with most logic synthesis and timing tools, which require combinational logic to be acyclic. As such, some sort of cycle-removal step is necessary to handle these circuits with existing tools. We present a two-stage algorithm for transforming a combinational cyclic circuit into an equivalent acyclic circuit. The first part quickly and exactly characterizes all combinational behavior of a cyclic circuit. It starts by applying input patterns to each input and examining the boundary between gates whose outputs are and are not defined to find additional input patterns that make the circuit behave combinational. It produces sets of assignments to inputs that together cover all combinational behavior. This can be used to report errors, as an optimization aid, or to restructure the circuit into an acyclic equivalent. The second stage of our algorithm does this restructuring by creating an acyclic circuit fragment from each of these assignments and assembles these fragments into an acyclic circuit that reproduces all the combinational behavior of the original cyclic circuit. Experiments show that our algorithm runs in seconds on real-life cyclic circuits, making it useful in practice.

**Index Terms**—Acyclic circuits, combinational logic, constructiveness, cyclic circuits, resynthesis.

## I. INTRODUCTION

**A** CYCLIC circuit composed of logic gates is usually used to hold state or oscillate, but, like an acyclic circuit, it can also behave combinational. Such cyclic combinational circuits compute a function that only depends on their current inputs [1], at least for certain input patterns. These circuits can arise in high-level synthesis [2], [3] and are the most compact representation for certain functions such as arbiters [4], [5].

Even though a cyclic combinational circuit computes a well-defined function of its inputs much like an acyclic combinational circuit, most circuit analysis tools forbid cycles. The central challenge of a cyclic circuit is how the evaluation order

Manuscript received July 30, 2007; revised December 17, 2007 and April 18, 2008. Current version published September 19, 2008. The work of O. Neiroukh and X. Song and his group was supported by Intel Corporation. The work of S. A. Edwards and his group was supported in part by the NSF and in part by an award from the SRC. This paper was recommended by Associate Editor A. Kuehlman.

O. Neiroukh is with Intel Corporation, Hillsboro, OR 97124 USA (e-mail: osama.neiroukh@intel.com).

S. A. Edwards is with the Department of Computer Science, Columbia University, New York, NY 10027 USA.

X. Song is with the Department of Electrical and Computer Engineering, Portland State University, Portland, OR 97207 USA.

Digital Object Identifier 10.1109/TCAD.2008.2003305

of its gates depends on its inputs; unlike an acyclic circuit, no one order works for all legal input patterns. This is what enables cyclic combinational circuits to be more compact, but it causes difficulties for tools such as static timing analyzers that rely on a static evaluation order. Furthermore, simulating cyclic circuits is more expensive and complicated than simulating acyclic ones.

We present an algorithm that can transform a cyclic combinational circuit, whether created deliberately or inadvertently, into an equivalent acyclic one suitable for a circuit analysis tool that insists on acyclic circuits. By running this early in a synthesis flow with tools that insist on acyclic circuits, this would allow a designer or a high-level synthesis procedure to create false loops as desired. Alternatively, in a synthesis flow that encourages cyclic circuits [6], it could be used to export an acyclic version of the circuit to a tool that demands one.

Our algorithm produces a useful side effect: an exact characterization of the inputs for which the cyclic circuit behaves combinational. We use this internally to make sure the circuit we generate “covers” all combinational behavior, but it could also be passed to a simulator or formal verification tool to check that the environment of a cyclic circuit always induces the circuit to behave combinational.

Consider the cyclic circuit in Fig. 1(a). From its truth table, Fig. 1(b), we see that the circuit is well behaved unless  $a = 0$ ,  $b = 1$ ,  $c = 0$ , and  $d = 1$ . For all other patterns, at least one of the gates has a controlling input that breaks the cycle.

This circuit is therefore combinational if  $a = 1$ ,  $b = 0$ ,  $c = 1$ , or  $d = 0$ . Fig. 1(c) considers each case. We call each case a *partial assignment* (PA), where we set one or more inputs to known values. These particular PAs each induce combinational behavior; each assignment makes the cyclic circuit behave like each of the acyclic circuit fragments in the middle column of Fig. 1(c).

Because each of these circuit fragments are acyclic, the gates they contain can be evaluated following the schedules listed in the right column of Fig. 1(c). A *schedule* is a gate evaluation order for the circuit. By themselves, each only covers part of the complete combinational behavior of the original circuit, but we can merge them into the schedule  $wxyzwxy$ , which we chose because it includes each of the schedules in Fig. 1(c). This comprehensive schedule can be evaluated by the acyclic circuit in Fig. 1(d), which has the same combinational behavior as the original circuit. The other input to the leftmost  $w$  gate is a don't care, and by choosing it to be 1, we obtain the simplified result in Fig. 1(e).

Our algorithm takes an acyclic circuit and synthesizes an acyclic one that computes the same function for inputs that

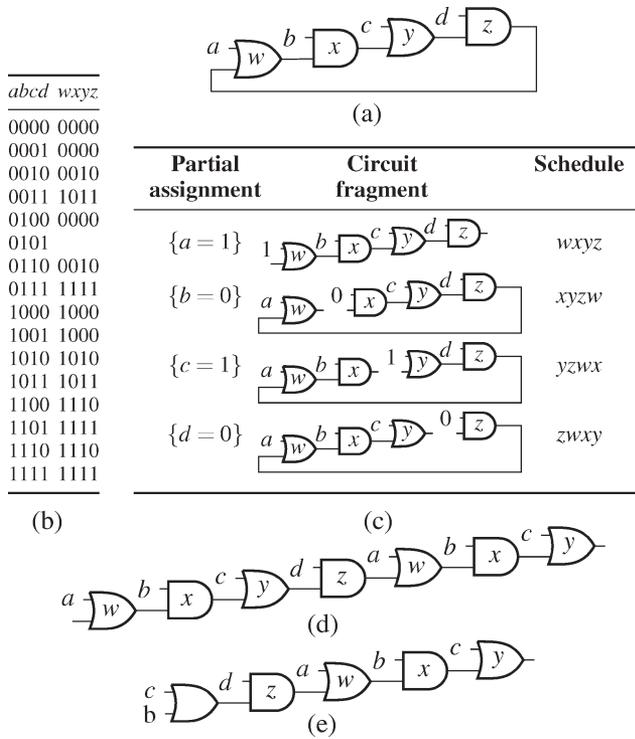


Fig. 1. (a) Simple cyclic circuit after Rivest’s [5]. (b) Its truth table. It is combinational for every input pattern except 0101. (c) Four PAs that together cover all combinational behavior and the gate evaluation schedules they induce. (d) Circuit generated from the merged schedule  $wxyzwxy$ . (e) Final simplified acyclic equivalent to (a).

produce combinational behavior. We treat inputs that produce noncombinational behavior as don’t cares: We assume that these inputs were never meant to be applied.

We solve this problem by addressing two main challenges: identifying a small set of behaviors that together “cover” all possible combinational behaviors of the original cyclic circuit and merging these behaviors into a final acyclic circuit. Because it must consider all behaviors, the former problem is more difficult than the latter. There are simple ways to solve these problems, but they produce circuits that are too big (quadratically larger) to be practical.

We organized this paper as follows. Section II reviews related research. We give an example of our method in Section III. We present our circuit model and theoretical foundations in Section IV and give the theory and an algorithm for computing schedules and transforming them to circuits in Sections V and VI, respectively. We describe the key to our algorithm—an algorithm for identifying all combinational behavior of a cyclic circuit—in Section VII. A mechanism for combining the acyclic fragments that this algorithm generates is described in Section VIII, and we present experimental results in Section IX.

## II. PRIOR WORK

The challenges of dealing with cyclic switching circuits dates back to at least the 1950s [7]. The basic problem is that cycles are necessary for building state-holding elements yet, due to inevitable uncontrollable delays, bring with them a host of challenges such as races and hazards [8], [9]. Analyzing and synthesizing such circuits is now well understood [10].

Our work rests on techniques for identifying when a circuit may have (generally unwanted) delay-dependent behavior. Eichelberger [11] was among the first to propose the now-standard three-valued simulation technique that answered this question (although Yoeli and Rinon [12] had also considered it), which was quickly put into commercial use [13].

Eichelberger’s algorithm is simple: To simulate a transition, set all changing inputs to  $X$ , propagate the effects of this throughout the circuit, then set all changing inputs to their final values and again propagate their effects. He showed that each gate output that becomes 0 or 1 after such simulation will stabilize to that value for any assignment of (finite) delays, i.e., the absence of  $X$ ’s means the circuit is race free. Brzozowski and Yoeli [14] later generalized this to allow the circuit to start in a transient (unstable) state. Bryant [15] applies this algorithm to MOS transistor circuits instead of logic gates.

Today, most circuits are built by connecting carefully designed state-holding elements such as D flip-flops with acyclic combinational logic, but there are reasons for considering cyclic combinational logic. Kautz [4] was the first to show that the minimal form of certain circuits contained combinational loops. Later, Rivest [5] came to a similar conclusion, suggesting that combinational loops are more than just a nuisance.

Stok [2] observes how false loops can arise from resource sharing in high-level synthesis and notes that “most logic synthesis systems and delay calculators (timing analyzers) are not able to handle them.” Stok proposes a specialized algorithm that avoids loops in this setting and showed that doing so did not negatively affect circuit quality. Because it has a better picture of the computation being performed, Stok’s algorithm almost certainly produces better circuits than ours. However, it only applies to loops caused by resource sharing; ours can remove any kind of loop.

Malik [1] considers the problem of analyzing when cyclic circuits behave combinational. He shows an equivalence between combinational cyclic circuits and the least fixed points in three-valued simulation. Shiple *et al.* [16] apply this to the Esterel language [17], [18], whose hardware translation [3] often produces combinational cycles. They use a symbolic state-space traversal followed by an  $O(n^2)$  replication procedure to remove cycles. By contrast, our technique heeds the structure of the original circuit when constructing an acyclic equivalent.

The binary decision diagram (BDD)-based algorithm of Halbwachs and Maraninchi [19] takes a brute-force approach, ignoring the structure of the circuit. Namjoshi and Kurshan [20] take a very different approach, showing that any fixed point is interesting, not just the least. Their analysis answers whether a circuit is combinational, but they do not attempt to use their results to construct an equivalent acyclic circuit.

Riedel and Bruck [6] apply Rivest’s observations to synthesize very compact combinational circuits that contain cycles. As part of their synthesis step, they check whether the circuit they generated is combinational using a BDD construction; our algorithm could be used in that setting. The cyclic combinational circuits they generate have topologies complex enough to stymie the decyclification algorithm of Edwards [21]—the starting point for our work. Our improved algorithm now easily handles these circuits.

Riedel and Bruck [6] also present a symbolic approach to deriving conditions under which a circuit is combinational. They use BDDs [22] and apply their technique to synthesizing a cyclic circuit from an acyclic starting point. Their network model is composed of node functions, which describe the logical function of each node in terms of primary inputs as well as other nodes. Like ours, their algorithm also computes the conditions under which a cyclic circuit behaves combinational. However, their algorithm relies on target functions, which describe the logical function of each of the internal nodes based on the input variables only, i.e., the target function for a node is the acyclic equivalent of its node function.

Riedel and Bruck’s algorithm requires both node and target functions. These are usually easy to compute and maintain starting from an acyclic specification (for a large circuit, such primary-input-only BDDs may be too large to construct), but they are much more difficult to compute in our setting, where we start from a cyclic circuit. In some sense, most of our algorithm is devoted to computing the equivalent of their target functions; therefore, to apply their algorithm on a cyclic circuit, it might be necessary to run our algorithm anyway. In any case, it is difficult to compare the efficiency of their technique with ours because they do not report runtimes.

Like us, Gupta and Selvidge [23] transform cyclic circuits into equivalent acyclic ones but take a very different approach. They decompose feedback paths into multiplexers, combine them, and transform the select inputs into enable pins on latches. Adding latches to the circuit actually results in a sequential circuit, not just an (combinational) acyclic one, which can be undesirable when the cycles never function as state-holding elements. Furthermore, their approach rejects cyclic circuits that appear to be able to oscillate, but their analysis is conservative and may reject circuits that are actually well behaved in practice. Finally, their technique enumerates all the loops in a circuit. While the number of loops in the commercial benchmarks they tried was modest, we suspect their technique would not work well on the highly connected circuits produced by Riedel and Bruck.

Much of the literature on cyclic circuits has been concerned with efficiency [4]–[6] or their production as a side effect of high-level synthesis [3]. In both settings, combinational circuits are not intended to be storage elements. We treat input conditions for which cyclic circuits behave noncombinationaly as don’t-care conditions.

Our work builds on the work of Neiroukh *et al.* [24], which is, in turn, built upon Edwards’ [21]. The earlier paper laid out the two stages of our algorithm: an exhaustive enumeration of acyclic circuit fragments that together cover all the combinational behavior of the cyclic circuit followed by a heuristic merge of these fragments to produce the final result. The acyclic fragment search procedure in the earlier paper often wasted a lot of time establishing that it had found all the necessary fragments, a problem considered and greatly improved upon in the later paper. Our current work further improves the efficiency of the algorithm by introducing the use of zero-suppressed decision trees (ZBDDs) to merge PAs, a subject we discuss at length in Section VII.

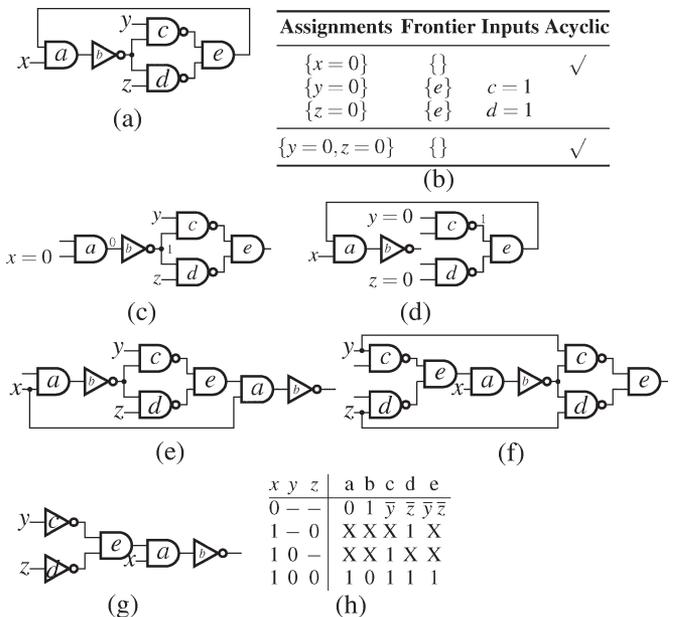


Fig. 2. Illustration of our algorithm. From the cyclic circuit in (a), our algorithm applies controlling values as listed in (b) to produce the two acyclic fragments in (c) and (d). These two fragments may be merged to produce either (e) or (f). Circuit (e) is smaller and may be further simplified to produce the circuit in (g), which behaves like (a) when gate outputs are not X. (h) is the truth table for (a).

Our approach has a number of unique strengths. It combines explicit enumeration to minimize the number of assignments to consider by following the structure of the circuit with an implicit ZBDD-based technique to greatly reduce the danger of a combinatorial explosion. A strength of this approach is the ease with which it handles cycles that are mostly or completely topological. For instance, it quickly discovers cyclic circuits that are combinational under all input patterns.

Malik [1] shows the determination of whether a cyclic circuit that is combinational is co-NP-complete; therefore, it is not surprising that our algorithm can exhibit exponential behavior on a cyclic circuit that is combinational under only one PA that assigns all strongly connected component (SCC) inputs. However, our algorithm is able to prune the search space on most circuits and run faster.

### III. EXAMPLE

The example in Fig. 1 illustrates how our algorithm works at a high level: A set of acyclic circuit fragments that cover all combinational behavior is derived. The fragments are then merged into a single acyclic circuit and simplified by applying controlling values to don’t-care inputs.

In this section, we present a more complicated example—Fig. 2—that illustrates more of the details in our algorithm. It is still too simple to show all the challenges that occur when resynthesizing big circuits; later sections cover these in detail.

As before, our first goal is to find a small set of PAs of values to inputs that, together, cover all the combinational behavior of the circuit. That is, we want an input vector to be combinational if and only if it is contained in one of our PAs.

Our algorithm begins by applying a controlling value to each input separately. Such a controlling value—a 0 input on an AND gate and a 1 applied to an OR gate—by definition forces the output of the gate to a given value regardless of the other inputs. Such inputs are required to “cut” the SCC and make it behave combinational. We formalize this later in Theorem 3.

The top of Fig. 2(b) summarizes the results of these initial assignments. First, when the  $x$  input is 0, the circuit is always combinational because 0 is a controlling value on gate  $a$ , which breaks the  $e$ -to- $a$  feedback loop. Fig. 2(c) shows the circuit fragment induced by this assignment. We include the assignment  $\{x = 0\}$  as part of our minimal cover and will not consider any further assignments that contain  $\{x = 0\}$ .

However, unlike the example in Fig. 1, not all of these input assignments induce combinational behavior by themselves. Consider what happens when  $y = 0$ . Although this is a controlling value for gate  $c$  (its output becomes 1 regardless of  $b$ ), by itself, this is not enough to force the whole circuit to behave combinational because a 1 on  $c$  is a noncontrolling value on the AND gate  $e$ . We refer to all such gates as the *frontier* induced by a PA (see Definition 4) because they define the boundary between combinational and possibly noncombinational behavior. Such gates can be thought of as being the cause of a logjam.

The key step in this phase attempts to break logjams by looking for promising combinations of PAs that affect the same frontier gates. This is a covering problem that requires us to enumerate all satisfying assignments, a fairly difficult problem that is important to solve quickly; we discuss this at length in Section VII-C. Here, only one gate  $e$  appears in any frontier; we will attempt to set the output of these gates by judiciously combining sets of PAs that might completely define values at the inputs of these gates.

To break the logjam at  $e$ , we consider PAs that affected  $e$ 's inputs:  $\{y = 0\}$  and  $\{z = 0\}$ . Each of these sets at least one of the inputs to  $e$  to a noncontrolling value (1, because  $e$  is an AND gate). We can break the logjam by looking for PAs that combine to set *all* of  $e$ 's inputs to noncontrolling values, i.e., by setting  $c = 1$  and  $d = 1$ , and do not have any conflicts. Thus, we decide to consider the PAs  $\{y = 0, z = 0\}$  in the next step.

The bottom row of Fig. 2(b) lists this new PA. This leads to an empty frontier and, therefore, an acyclic circuit. The first part of our algorithm terminates and returns the two acyclic PAs shown in Fig. 2(b). Together, these two PAs represent all combinational behavior in the original cyclic circuit.

The scheduling algorithm fuses circuit segments constructed from PAs into schedules. When fusing circuits, a gate with identical inputs in each fragment can be shared, and don't-care gate inputs can be assigned as desired to produce behavior from either fragment.

The simple merging procedure used by the algorithm produces two circuits: Fig. 2(e) and (f). Fig. 2(e) comes from appending Fig. 2(c) to the end of Fig. 2(d), and Fig. 2(f) is Fig. 2(d) appended to Fig. 2(c). Fig. 2(e) is smaller (seven gates versus eight); so we choose to discard Fig. 2(f).

The unlabeled input on the left of Fig. 2(e) is a don't care because we know that the other input on that gate will be set to a controlling value when combinational input patterns are

```

function SIMULATE( $\langle G, I, W \rangle, x, s$ )
 $v_0(n) = \begin{cases} x(n) & \text{if } n \in I, \\ \perp & \text{if } n \in G. \end{cases}$ 
 $i \leftarrow 0$ 
while for some  $g$ , EVAL( $W, v, g$ )  $\neq v$  do
   $i \leftarrow i + 1$ 
   $v_i = \text{EVAL}(W, v_{i-1}, s_i)$ 
return  $v_i$ 

function EVAL( $W, v, g$ )
 $o = \begin{cases} 0 & \text{if } v(d) = 1 \text{ for all } d \text{ s.t. } (d, g) \in W, \\ 1 & \text{if } v(d) = 0 \text{ for some } d \text{ s.t. } (d, g) \in W, \\ \perp & \text{otherwise.} \end{cases}$ 
Let  $v'(g) = o$  and  $v'(n) = v(n)$  otherwise.
return  $v'$ 

```

Fig. 3. Three-valued simulation algorithm, which takes a circuit  $\langle G, I, W \rangle$ , an input function  $x$ , and an infinite schedule of gates  $s$ . It evaluates gates until it reaches a fixed point using EVAL, which updates a single (NAND) gate.

applied; therefore, we may set it as we like. Setting to 0 is the judicious choice, giving the circuit in Fig. 2(g). Note that, as desired, this circuit follows the truth table in Fig. 2(h) when no gate's output is  $X$ .

#### IV. OUR CIRCUIT MODEL

While we feel the approach in our algorithm—applying input patterns that induce combinational behavior then merging the resulting acyclic circuits—is intuitively correct, in the succeeding paragraphs, we justify it formally by defining exactly what we mean by circuits and combinational behavior. Specifically, we describe our model of cyclic gate circuits and define their semantics with the usual three-valued simulation algorithm (Fig. 3). We show that this simulation algorithm produces the same result for every possible gate evaluation order (Theorem 1), making it a reasonable definition for the semantics of these circuits, and that there is always some finite gate evaluation sequence (schedule) that works for every possible input for a particular circuit (Corollary 1). In Section V, we will show how such a schedule can be used to produce an acyclic circuit. Finding such schedules—the main challenge—is the subject of Sections VI–VIII.

We use a simple gate-level circuit model: A circuit  $C$  is a tuple  $C = \langle G, I, W \rangle$ , where  $G$  is a finite set of gates,  $I$  is a finite set of primary inputs, and  $W \subseteq (G \cup I) \times G$  is the set of wires. Each gate computes the logical NAND of its inputs; we assume that more complex gates have been dismantled into NANDs. Note that primary inputs have no incoming edges. We consider every gate to be an output.

We treat gates as taking one of three values: 0, 1, and  $\perp$ . The first two values are self-explanatory; we write  $\perp$  instead of the  $X$  usually used in three-valued logic simulation to emphasize the connection with lattices and partial orders.

The three wire values are partially ordered with a relation  $\sqsubseteq$  that satisfies  $\perp \sqsubseteq 0$  and  $\perp \sqsubseteq 1$  and is transitive ( $x \sqsubseteq y$  and  $y \sqsubseteq z$  implies  $x \sqsubseteq z$ ), reflexive ( $x \sqsubseteq x$ ), and antisymmetric ( $x \sqsubseteq y$  and  $y \sqsubseteq x$  implies  $x = y$ ).

The relation  $\sqsubseteq$  can be thought of as an information ordering:  $\perp$  is less defined than 0 or 1, but neither  $0 \sqsubseteq 1$  nor  $1 \sqsubseteq 0$  since both represent the same amount of information, i.e.,

a defined value. The pointwise extension of this relation to vectors reinforces this intuition:  $(x_1, \dots, x_n) \sqsubseteq (y_1, \dots, y_n)$  iff  $x_1 \sqsubseteq y_1, \dots$ , and  $x_n \sqsubseteq y_n$ . More informally, if  $X \sqsubseteq Y$ , then each element of  $Y$  is either the same as its counterpart in  $X$  or a  $\perp$  in  $X$  is a 0 or 1 in  $Y$ .

*Definition 1:* A *controlling value* for a gate  $G$  is the non- $\perp$  value that, when applied to any input of  $G$ , uniquely sets  $G$ 's output to a non- $\perp$  value independent of assignment to other inputs.

It follows from this definition that, for a gate's output to be set to non- $\perp$ , either all inputs must be set to noncontrolling values or at least one input must be set to a controlling value. For a NAND gate, 0 is a controlling value and 1 is noncontrolling.

*Definition 2:* An *SCC* of a circuit  $C$  is a maximal subset of gates  $V \subseteq G$  such that there is a (wire) path from any gate in  $V$  to any other gate in  $V$ . Inputs of an SCC are inputs of gates that are part of the SCC, which are either primary inputs or are driven by gates inside the SCC.

### A. Three-Valued Simulation

Following Malik [1] and Eichelberger [11], we define the semantics of our circuits as being the results of three-valued simulation. Fig. 3 shows the algorithm. The three arguments to SIMULATE are a circuit, a function  $x : I \rightarrow \{\perp, 0, 1\}$  that defines the state of the inputs, and an infinite sequence of gates  $s = (s_1, s_2, \dots)$  where  $s_k \in G$  that defines a fair evaluation schedule. Specifically, for any  $g \in G$  and any  $j > 0$ , there is some  $k > j$  for which  $s_k = g$ .

Shiple [25] shows that a three-valued simulation is a tight approximation of Brzozowski and Seger's [26] up-bounded inertial delay model, which assumes that the delay of a gate or wire in a circuit may range from infinitesimal to some upper bound. Specifically, if the result of three-valued simulation with non- $\perp$  inputs does not contain any  $\perp$ s, the up-bounded model will produce this result for any assignment of actual delays. Conversely, if three-valued simulation produces a node with value  $\perp$ , then there is a delay assignment that will make the value of that node behave unpredictably.

SIMULATE is a chaotic iteration procedure that first initializes  $v_0$ , the state of the nodes in the circuit, then enters a loop in which it evaluates each gate according to the schedule  $s$ . All the gates are set to  $\perp$  in the initial state. The EVAL function updates the state of gate  $g$  by examining the state of the gates and inputs that drive it. The loop ends when the process converges, i.e., when updating all gates has no effect.

By starting all the gates in the  $\perp$  state, we make this algorithm equivalent to Eichelberger's [11] when all inputs transition simultaneously. We make this conservative assumption because we are concerned with circuits that completely ignore their previous state, i.e., are combinational.

*Lemma 1:*  $v_0 \sqsubseteq v_1 \sqsubseteq \dots \sqsubseteq v_k \sqsubseteq \dots$ .

*Proof:* By induction. First, EVAL only changes  $v'(g)$ , and  $g \in G$  from SIMULATE; therefore,  $v_k(i) = x(i)$  for all  $k \geq 0$  and  $i \in I$ .

Base case:  $v_0(g) = \perp$  for all  $g \in G$ ; therefore, it must be that  $v_1 \sqsubseteq v_0$  since  $\perp \sqsubseteq x$  for any  $x \in \{\perp, 0, 1\}$ .

*Induction:* Assume that  $v_0 \sqsubseteq v_1 \sqsubseteq \dots \sqsubseteq v_{i-1}$ .  $v_i$  is  $v_{i-1}$  with  $v_{i-1}(s_i)$  replaced by  $o$  computed by EVAL. If  $v_{i-1}(s_i) = \perp$ , then the induction hypothesis holds trivially.

If  $v_{i-1}(s_i) = 0$ , then there was some  $j < i$  for which  $v_j(d) = 1$  for all  $d \in \{d : (d, s_i) \in W\}$ . From the induction hypothesis, it follows that  $v_j(d) \sqsubseteq v_{j+1}(d) \sqsubseteq \dots \sqsubseteq v_{i-1}(d)$ . However, since  $v_j(d) = 1$ , this means that  $v_{i-1}(d) = 1$ , and therefore, EVAL sets  $v_i(s_i) = 0$  and the induction hypothesis holds for  $i$ .

Similarly, if  $v_{i-1}(s_i) = 1$ , there was some  $j < i$  for which  $v_j(d) = 0$  for some  $d \in \{d : (d, s_i) \in W\}$ . Similar reasoning finds  $v_i(s_i) = 1$ , and again, the induction hypothesis holds. ■

*Lemma 2:* The SIMULATE function terminates.

*Proof:* In each iteration of the *while* loop, a single gate is evaluated. Lemma 1 implies that each gate can only change its value once and only from  $\perp$  to 0 or 1. Since there are a finite number of gates and the  $s_i$ 's are distributed fairly, the termination condition for the *while* loop will eventually be reached and the function will terminate. ■

*Lemma 3:* If  $v \sqsubseteq w$ ,  $\text{eval}(W, v, g)(g) \sqsubseteq \text{eval}(W, w, g)(g)$ .

*Proof:* If  $v \sqsubseteq w$ ,  $v(d) \sqsubseteq w(d)$  for any  $d$ . From the definition of  $o$  in EVAL,  $\text{eval}(C, v, g)(g) = 0$  implies  $v(d) = 1$  for all  $d \in D$ . Since  $v(d) \sqsubseteq w(d)$ , it follows that  $w(d) = 1$  for all  $d \in D$ ; therefore,  $\text{eval}(C, w, g)(g) = 0$ . Similarly, if  $\text{eval}(C, v, g)(g) = 1$ , then  $v(d) = 0$  for some  $d \in D$ . Similarly, it follows that  $w(d) = 0$  for that  $d$ ; therefore,  $\text{eval}(C, w, g)(g) = 1$ . Finally, if  $\text{eval}(C, v, g)(g) = \perp$ ,  $\perp \sqsubseteq o$  for any  $o = \text{eval}(C, w, g)(g)$ . ■

*Theorem 1:* The result of the SIMULATE function for a particular circuit and input is the same for all fair schedules  $s$ .

*Proof:* We will show that SIMULATE always computes the (unique) least fixed point of a monotonic function on a complete partial order. Let

$$f(\langle G, I, W \rangle, x, v)(n) = \begin{cases} \text{EVAL}(W, v, n), & \text{if } n \in G \\ x(n), & \text{if } n \in I. \end{cases}$$

Computing  $f$  amounts to evaluating every gate, and the termination condition on the *while* loop is equivalent to  $f(\langle G, I, W \rangle, x, v) = v$ . From Lemma 2, it follows that SIMULATE always returns some fixed point of  $f$ .

From Lemma 3,  $\text{eval}(W, v, g)$  is monotonic on the  $g$  component; therefore, it follows from the definition of  $f$  that  $f$  is monotonic in  $v$ , i.e., if  $v \sqsubseteq w$ , then  $f(C, x, v) \sqsubseteq (C, x, w)$ .

Since  $f$  is a monotonic function on the complete partial order whose elements are  $v : G \cup I \rightarrow \{\perp, 0, 1\}$ , it follows from the folk theorem variously attributed to Knaster, Tarski, and Kleene [27] that  $f$  has a unique least fixed point. Call that least fixed point  $f^*(C, x)$ .

From that same theorem, we know

$$v_0 \sqsubseteq f(C, x, v_0) \sqsubseteq f(C, x, f(C, x, v_0)) \sqsubseteq \dots \sqsubseteq f^*(C, x).$$

By definition,  $v_1 = \text{eval}(W, v_0, s_i)$ . From the definitions of  $f$  and EVAL, it follows that  $v_1 \sqsubseteq f(C, x, v_0)$ .

Now, assume that  $v_{i-1} \sqsubseteq f^{i-1}(C, x, v_0)$  where  $f^i(C, x, v_0) = f(C, x, f(C, x, f(\dots, f(C, x, v_0) \dots)))$  ( $f$  applied  $i$  times).

From Lemma 3, this implies that

$$\begin{aligned} \text{eval}(W, v_{i-1}, s_i) &\sqsubseteq \text{eval}(W, f^{i-1}(C, x, v_0), s_i) \\ v_i &\sqsubseteq \text{eval}(W, f^{i-1}(C, x, v_0), s_i) \\ &\sqsubseteq f^i(C, x, v_0). \end{aligned}$$

By induction, it follows that  $v_i \sqsubseteq f^i(C, x, v_0)$  for all  $i$ , and thus,  $v_i \sqsubseteq f^*(C, x)$ . However, since we know that SIMULATE always terminates at the first fixed point it finds and  $f^*(C, x)$  is the least such fixed point, SIMULATE must always terminate at  $f^*(C, x)$ , the unique least fixed point of  $f$ . ■

Because of Theorem 1, we can define the (unique) behavior of a circuit as follows:

$$f(C, x) = \text{SIMULATE}(C, x, s) \text{ for all fair } s. \quad (1)$$

Note that  $f$  is a function that takes a circuit and a function defining its inputs and returns a function defining its inputs and gates. Formally

$$f : \langle G, I, W \rangle \times (I \rightarrow \{\perp, 0, 1\}) \rightarrow (G \cup I \rightarrow \{\perp, 0, 1\}).$$

*Corollary 1:* For a particular input  $x$ , there is a finite-length schedule  $s$  that leads to convergence in which each gate appears exactly once.

*Proof:* Lemma 2 tells us that a finite  $s$  suffices. Lemma 1 suggests that each gate can change its value at most once during the evaluation procedure; therefore, we can simply remove from any schedule  $s$  all gates that do not cause a change in  $v$ . Theorem 1 tells us that this does not affect the final result. ■

### B. Combinational Circuits

Like Malik [1], we say that a circuit  $C = \langle G, I, W \rangle$  is combinational for an input  $x$  if  $f(C, x)(g) \neq \perp$  for all  $g \in G$  (i.e., three-valued simulation does not lead to any  $\perp$ -valued gates). Again, because of Shiple [25], this is equivalent to insisting that the circuit always stabilizes and never holds state for any delay assignment. Literature on cyclic circuits also refers to this behavior as “well behaved” and “constructive” [16].

Since we consider all gate outputs to be primary outputs, our definition of combinational insists that every part of the circuit stabilizes. This is actually a conservative definition of combinational behavior: If the environment does not observe the output of, for example, an oscillator, should its presence really matter? Arguments can be made on both sides, but we suspect that a designer who wants a combinational circuit does not want any state-holding or oscillatory behavior; therefore, we disallow it.

Our goal is to produce an acyclic circuit whose behavior matches that of a cyclic circuit for inputs that are combinational. We assume that noncombinational behavior, if any, was unintended and treat inputs that induce it as don’t-care patterns.

Fig. 4 shows a circuit consisting of a single SCC whose inputs are  $a$ ,  $b$ , and  $c$ . When analyzing a circuit, we first decompose it into SCCs using a standard algorithm [28]. If the input circuit contains more than one SCC, we consider each SCC separately in a topological order.

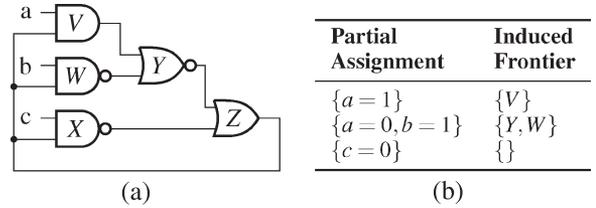


Fig. 4. (a) Cyclic circuit. (b) PAs and their induced frontiers—the boundary between defined and  $X$ -valued gates after applying inputs.

**function** SIMULATE-INITIAL( $\langle G, I, W \rangle, x, i, s$ )

$$v'_0(n) = \begin{cases} x(n) & \text{if } n \in I, \\ i(n) & \text{if } n \in G. \end{cases}$$

$j \leftarrow 0$

**while** for some  $g$ ,  $\text{EVAL}(W, v', g) \neq v'$  **do**

$j \leftarrow j + 1$

$v'_j = \text{EVAL}(W, v'_{j-1}, s_j)$

**return**  $v_j$

Fig. 5. Variation of the simulation algorithm that starts the value of each gate at an initial state  $i$ .

### V. IMPLEMENTING SCHEDULES AS ACYCLIC CIRCUITS

Corollary 1 tells us that each circuit has some finite-length gate evaluation order that evaluates the circuit for every possible input. In this section, we show how to use such a schedule to build an acyclic circuit that computes the same function as the original cyclic circuit. Finding good schedules is challenging; we address this problem in Sections VI–VIII.

The SIMULATE function exhibits a simple sequential behavior: It evaluates gates in scheduled order, taking the inputs of each gate from the most recently computed state of all the gates in the circuit.

Consider Fig. 5—a variant of the three-valued simulation algorithm from Fig. 3 that begins with each gate in an initial state given by the function  $i : G \rightarrow \{0, 1\}$ .

*Theorem 2:* If  $x(i) \neq \perp$  for all  $i \in I$  and  $f(C, x)(g) \neq \perp$  for all  $g \in G$ , then  $\text{simulate-initial}(C, x, i, s) = f(C, x)$  for all  $i : G \rightarrow \{0, 1\}$ .

*Proof:* First, note that  $v_0 \sqsubseteq v'_0$  because  $v_0(n) = v'_0(n)$  for  $n \in I$  by construction and  $v_0(n) = \perp \sqsubseteq v'_0(n)$  for  $n \in G$ . From Lemma 3, it follows that  $\text{eval}(W, v_0, s_1) \sqsubseteq \text{eval}(W, v'_0, s_1)$ ; therefore,  $v_1 \sqsubseteq v'_1$ , and by induction,  $v_j \sqsubseteq v'_j$  for all  $j$ . However, by assumption,  $f(C, x)(g) \neq \perp$ , there is some  $v_j$  that does not contain any  $\perp$  elements (i.e., when the procedure has converged). However,  $v_j \sqsubseteq v'_j$  from the aforementioned inductive argument, and since  $v_j(n) \in \{0, 1\}$ , it follows that  $v'_j = v_j$  since only  $0 \sqsubseteq 0$  and  $1 \sqsubseteq 1$ . ■

Theorem 2, suggested by Gérard Berry, makes it possible to simulate circuits that are known to be combinational without resorting to three-valued evaluation. Since the  $v'_j$ 's never contain any  $\perp$ s, SIMULATE-INITIAL reduces to two-valued simulation.

At the heart of our technique is a straightforward trick: It is easy to construct an acyclic gate-level circuit that behaves like SIMULATE-INITIAL by following the schedule.

The procedure is simple: For each gate in the schedule in scheduled order, add a copy of that gate to the circuit by connecting its inputs to the output of the “most recent copy” of that gate (or primary input) and call that gate the new most

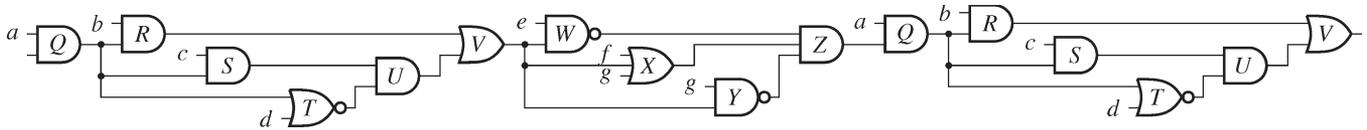


Fig. 6. Acyclic circuit generated from the schedule  $QRSTUVWXYZQRSTU$  for the cyclic circuit in Fig. 11(a).

recent copy. To begin with, the most recent copy of each gate is a wire initialized to either 0 or 1. Theorem 2 tells us that any choice of these initial values will produce the same result. Thus, these are exactly don't-care inputs and can be used to simplify the logic later.

Fig. 6 shows the circuit constructed from the schedule  $QRSTUVWXYZQRSTU$  applied to the cyclic circuit in Fig. 11(a). Note that all wires go from left to right, that the gates appear in scheduled order from left to right, and that the function and inputs of each gate exactly follow those in the cyclic circuit. We note that the bottom input on the leftmost  $Q$  gate has been left unconnected—this represents a don't-care input, and to simplify the circuit, we may set it to 0.

### VI. COMPUTING SCHEDULES

Previously, we described machinery for synthesizing an acyclic circuit whose behavior matches that of a cyclic circuit: a mechanical procedure for constructing an acyclic circuit from a schedule—a gate evaluation order for the `SIMULATE` procedure that computes the function of the circuit.

We have two objectives in choosing these schedules: They must be correct (i.e., correctly compute the function of the circuit), and we would like them to be short since the longer the schedule, the larger and slower the generated circuit. We consider the correctness issue in this section and present an efficient algorithm for computing short schedules in Sections VII and VIII.

For acyclic circuits, the scheduling problem is straightforward: Any topological sort of the gates in the circuit is a correct schedule, and unless any outputs are ignored, this is also a minimal schedule. For cyclic circuits, there is no topological sort of the gates in the circuit, and the scheduling problem is more subtle. A key difference between cyclic and acyclic circuits is that the evaluation order for a cyclic circuit depends on its inputs; an acyclic circuit always has some input-independent evaluation order.

Thus, the key objective in transforming a cyclic circuit into an acyclic one that computes the same function is making sure that every needed evaluation order of the acyclic circuit is somehow contained in the “unrolled” acyclic version. Unfortunately, this requires duplicating gates, which is consistent with the minimality result of Kautz [4]. The problem then becomes finding an evaluation sequence that contains all the possible evaluation sequences as (possibly noncontiguous) subsequences. For example, if 123 and 132 were evaluation sequences, the sequence 1232 would cover them.

#### A. Covering All Permutations

For a schedule to be correct—for it to generate an acyclic circuit whose function matches that of a cyclic circuit—it must

TABLE I  
SHORTEST SEQUENCES COVERING ALL PERMUTATIONS OF  $1, \dots, n$

$n$	Shortest sequence	Length
2	121	3
3	1231231	8
4	123412314321	12
5	1234512341523142513	19

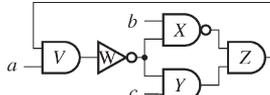
contain at least one correct evaluation order (i.e., one that brings the simulation procedure to a fixed point) for every possible combinational input pattern. This does not mean the schedule needs to contain every legal evaluation order. For example, any topological sort is a possible evaluation order of an acyclic circuit, but a valid schedule only needs to contain one topological sort.

The scheduling problem can be divided into two problems: finding all the evaluation orders the original circuit can exhibit and constructing a small schedule that contains all of them. It is critical that the final schedule contain a valid evaluation order for every possible combinational input; if one were missing, the generated circuit might compute a different value for an input pattern that needed that evaluation order. Including more evaluation orders than necessary does not affect correctness, but we wish to eliminate as many of these as possible as they tend to (needlessly) increase the size of the circuit.

For any input pattern applied to the cyclic circuit, it is always possible to compute the circuit output by evaluating each gate at most once. Thus, the only evaluation sequences we need to consider are permutations of all the gates, i.e., for a particular input pattern, we never need to consider sequences in which some gates appear more than once.

A brute-force solution to the scheduling problem then amounts to finding schedules that contain all possible permutations of the original set of gates. One simple schedule always works:  $n$  repetitions of all  $n$  gates in the circuit. It is easy to see that this includes all permutations as subsequences: For a particular permutation, the first gate appears somewhere in the first repetition of all the gates, the second gate appears in the second, and so forth. Unfortunately, this simple approach produces a schedule of length  $n^2$ , which is usually much longer than necessary.

Finding a sequence that covers all permutations of  $1, \dots, n$  is a studied problem in discrete mathematics, but all solutions are  $O(n^2)$ . Koutas and Hu [29] and, later, Mohanty [30], give constructions that produce strings of length  $n^2 - 2n + 4$ . These constructions are complicated and produce seemingly chaotic sequences; Table I gives examples of these strings for some small  $n$ . Cai [31] lowers the bound slightly for  $n \geq 8$ , but does not give a construction. Erra *et al.* [32] reach similar conclusions.



Technique	Schedule	Cost
Quadratic	vwxzyzvwxyzvwxyzvwxyzvwxyz	25
Mohanty [30]	vwxzyzvwxyzvwxyzvwxyz	19
Bourdoncle [33]	wxyzvwxyz	9
Ours	yzvwxyz	7

Fig. 7. Result of various scheduling techniques applied to a simple cyclic circuit. The quadratic solution is the most brute force; Mohanty's is a more clever way to cover all permutations; Bourdoncle's technique considers the structure of the circuit; ours also considers the function of the circuit.

### B. SCC Decomposition

An obvious optimization is to decompose the circuit into SCCs and deal with each separately. While evaluating each SCC can be complicated, the graph of SCCs is a directed acyclic graph; therefore, the SCCs can be evaluated in topological order, i.e., so that each SCC is evaluated exactly once. The simplest case is when the circuit is acyclic: Each SCC is a single gate, and a single gate only needs to be evaluated once. Each SCC, however, needs to be scheduled more cleverly.

A few scheduling techniques that perform recursive SCC decomposition have been developed. Bourdoncle's technique [33] selectively removes a single gate from an SCC, then recurses on the (hopefully simpler) circuit that results. Edwards and Lee [34] extend this technique by selecting two or more gates to be removed, often producing better results.

Fig. 7 shows a five-gate circuit and some correct schedules. The first two schedules attempt to cover all permutations, but are overkill because the original circuit cannot exhibit many of these permutations (e.g.,  $wzxyv$  is covered by both of these schedules but need not be). The third schedule takes the structure of the circuit into account; the fourth also considers its function. Both approaches reduce the length of the schedule, but considering function is superior.

## VII. FINDING A SMALL COMBINATIONAL COVER

We now present an algorithm that efficiently finds a cover for all combinational behavior of a cyclic circuit.

### A. Theoretical Background

*Definition 3:* Let the set  $\{x_1, \dots, x_n\}$  represent the inputs of an SCC. We define a PA as a set

$$\text{PA} = \{x_i = v_k : x_i \in (x_1, \dots, x_n) \wedge v_k \in \{0, 1\}\}.$$

In this work, we are only concerned with PAs to inputs of SCCs. A PA is always associated with some SCC. For example, a valid PA for the circuit in Fig. 4 is an assignment to one or more of the inputs  $\{a, b, c\}$ , such as  $\{a = 0\}$ ,  $\{b = 0, c = 1\}$ , and  $\{b = 1, c = 1\}$ .

Our algorithm relies on the following two theorems, which are keys to its correctness and efficiency.

*Theorem 3 (Edwards [21]):* For a circuit with an SCC to behave combinational, at least one input to a gate in the SCC must be driven to a controlling value.

For example, controlling assignments to SCC inputs for the circuit in Fig. 4 are  $a = 0$ ,  $b = 0$ , and  $c = 0$ . Theorem 3 tells us that at least one of these is required for combinational behavior.

*Theorem 4 (Edwards [21]):* If a PA  $p$  is combinational, then any further assignments that do not contradict any in  $p$  can also be computed combinational by the circuit fragment implied by  $p$ .

Consider the PA  $\{c = 0\}$  applied to Fig. 4. This breaks the connectivity of the SCC, making the circuit behave combinational. This theorem indicates that additional assignments beyond  $\{c = 0\}$  cannot reverse the combinational behavior already implied by this PA. This permits us to avoid further consideration of acyclic PAs once we have identified them. This supports one of our objectives for the algorithm: generation of *minimal* PAs that capture all combinational behavior. We explain the notion of minimal PAs in Section VII-C.

The main difficulty with SCCs is the lack of order in which they can be analyzed; SCC gates cannot be sorted topologically. To get around this, we introduce a concept that simplifies SCC analysis.

*Definition 4:* The cyclic controllability *frontier* of a PA is the set of SCC gates that have at least one non- $\perp$  input but whose output is  $\perp$ .

The frontier captures the notion of a boundary between gates whose output is defined and those whose output is not. A frontier is always associated with a PA. When calculating the frontier for a PA, we use ternary simulation to propagate PAs from SCC inputs as far as possible and then check for cyclic behavior. Fig. 4(b) shows some frontiers induced by PAs for the circuit in Fig. 4(a).

*Theorem 5:* A PA makes an SCC combinational if and only if its frontier is empty.

*Proof:* If part: If the frontier is empty, then either no gates have any inputs assigned or none has an output of  $\perp$ . From Theorem 3, we know that at least one gate must be driven by a controlling value for combinational behavior. If none has an output of  $\perp$ , then the circuit under that PA is combinational by definition.

Only if part: This follows directly from the definition of combinational behavior. ■

This theorem tells us that nonempty frontiers only exist in the presence of SCCs. For example, the PA  $\{c = 0\}$  in Fig. 4 yields an empty frontier. Stated differently, we broke the loop without having to assign specific values to the inputs  $\{a, b\}$ .

### B. Searching for Combinational Behavior

We use Theorem 3 to seed our search space with a pool of PAs, each corresponding to a controlling assignment to an SCC input. Any combinational behavior is guaranteed to be present in supersets of one or more of these PAs. Our algorithm proceeds by recording the frontier associated with each PA and uses them to look for opportunities to merge PAs in an attempt to find empty frontiers.

```

1:  $A = \emptyset$                                 ▷ Set of acyclic PAs, the eventual result
2:  $K = \emptyset$                              ▷ All known cyclic PAs, used for merging
3: Clear  $F$                                   ▷ A map from frontier gate  $\rightarrow$  set of PAs
4: while circuit has SCCs do
5:   Find next SCC
6:    $P =$  controlling values for SCC inputs    ▷ Initial PAs
7:   while  $P \neq \emptyset$  do
8:      $G = \emptyset$                           ▷ Frontier gates for this iteration
9:     for all  $p \in P$  do                    ▷ Consider each candidate PA
10:       simulate  $p$ 
11:       if circuit is combinational under  $p$  then
12:         add  $p$  to  $A$ 
13:       else
14:         add  $p$  to  $K$     ▷ Remember the PA for merging
15:         for all gate  $g$  in the frontier induced by  $p$  do
16:           add  $g$  to  $G$     ▷ Record the frontier gate
17:           add  $p$  to  $F(g)$   ▷ Remember  $p$  induced  $g$ 
18:        $P = \emptyset$                                 ▷ Compute new candidate PAs
19:       for all frontier gate  $g \in G$  do
20:         if  $|F(g)| > 1$  then    ▷ Need  $\geq 2$  PAs to merge
21:           add each PA from mergeAtGate( $K, g$ ) to  $P$ 
22: return  $A$ 

```

Fig. 8. Our algorithm for finding a minimal set of PAs for a circuit (SCC) that together cover all combinational behavior.

Fig. 8 shows our algorithm for identifying all combinational behavior. It takes a circuit with any number of SCCs and produces a set of PAs under which the circuit is combinational. These PAs control SCC inputs.

The algorithm attacks one SCC at a time (line 4), finding a minimal set of covering PAs for each. For each SCC, it begins by considering PAs that place a single controlling value on each SCC input (line 6), then enters into a loop (lines 7–21) that alternates between testing whether any of the currently considered PAs (set  $P$ ) induce combinational behavior (lines 10–17) and attempting to merge PAs that are already observed (set  $K$ ) to generate a new set of PAs (lines 18–21). Its goal in this second phase is to break logjams by combining PAs to set the outputs of the latest set of frontier gates it has discovered. The map  $F$  records PAs that affect frontier gates: If  $g$  is a gate, then  $F(g)$  is the set of all PAs that put at least one noncontrolling value at an input of  $g$ .

The algorithm in Fig. 8 will always find all combinational behavior within the subject circuit. Starting from individual controlling inputs into SCCs, our frontiers allow us to identify all opportunities where PAs can merge to extend controllability over more gates in an SCC. As we merge these PAs and continue the searching, other acyclic PAs are explored. We continue this cycle of search and merge, terminating when we fail to generate new PAs.

### C. Merging PAs

Here, we describe a key operation used in our main algorithm (Fig. 8): the generation of new PAs to break the logjam at a frontier gate. Given a set of PAs and a gate, the algorithm in Fig. 10 generates a set of PAs that apply noncontrolling values to *every* input of the gate, thus setting its output. This algorithm is the key improvement over the technique we presented earlier [24].

We store PAs in a simulated state that captures all assigned nodes and their values. The main algorithm (Fig. 8) only tries

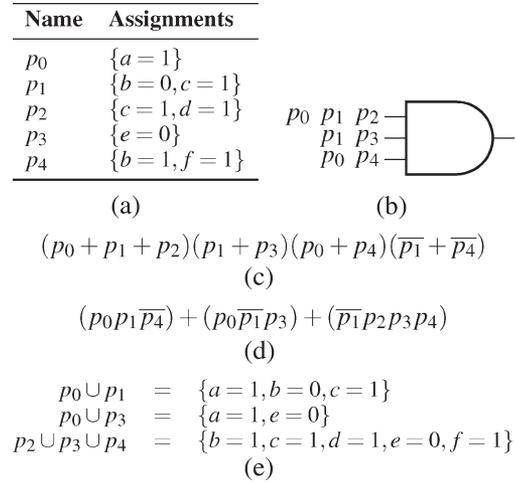


Fig. 9. (a) Merging PAs at (b) a gate. Our algorithm constructs (c) a product of sums to capture the merging constraints, then transforms it into (d) an ISOP. Each product represents a way to merge PAs (negated terms are ignored), giving (e) a new set.

to merge PAs for a gate when at least two PAs set an input on the gate. Merging attempts to produce new PAs by propagating known values across these frontier gates to extend the set of gates whose output is not  $\perp$ .

Consider the example in Fig. 9. Fig. 9(b) shows a three-input (frontier) gate  $g$  for PAs  $p_0, \dots, p_4$ . As always, these PAs control inputs (here  $a, \dots, f$ ) to the SCC that contains  $g$  and not usually the gate’s inputs. Note that a gate can only be a frontier for a PA if that PA puts a noncontrolling value on one or more of the gate’s inputs. We wish to consider merging these PAs to extend the frontier beyond  $g$ . A desirable merge of PAs at a gate  $g$  must be

- 1) *A gate cover.* Merged PAs must define every input of  $g$ .
- 2) *Consistent.* Merged PAs must not contain conflicting assignments to inputs. In Fig. 9, PAs  $p_1$  and  $p_4$  cannot be combined due to a conflicting assignment for  $b$ .
- 3) *Complete.* PAs must be merged such that all permissible combinations are considered. The example in Fig. 9 illustrates that there are different ways to cover every input. All must be considered, which ensures that our final PAs encapsulate both necessary and sufficient conditions for combinational behavior.
- 4) *Minimal.* Merged PAs must not contain any PA that can be removed while satisfying the previous conditions. For example, the merge candidate  $p_0 \cup p_3 \cup p_4$  is rejected since  $p_0$  dominates  $p_4$  (i.e.,  $p_0$  controls both first and third gate inputs;  $p_4$  only controls the third). This condition keeps the final output PAs as concise as possible by not including redundant conditions. Such redundancy has two drawbacks: It burdens subsequent stages of the algorithm because it increases memory usage, and it makes testing the merge conditions against other candidate PAs more tedious.

The gate cover, consistency, and completeness conditions are necessary for correctness (without the first two, the analysis does not make sense; the third one guarantees that we do not miss any necessary PAs), but minimality is merely desirable—it

```

1: function MERGEATGATE( $K, g$ )
2:    $R = \emptyset$  ▷ Generated set of PAs
3:    $POS = 1$  ▷ Product of Sums
4:   for all input  $i$  of gate  $g$  do
5:      $p_i =$  PAs in  $K$  that set  $i$  and induce  $g$  as a frontier
6:     if  $p_i = \emptyset$  then return  $\emptyset$  ▷ Cannot control some input
7:      $P = \bigvee p_i$ 
8:      $POS = POS \wedge P$ 
9:   for all Conflicting PA pairs  $\{p_i, p_j\} \in K$  do
10:     $POS = POS \wedge (\overline{p_i} \vee \overline{p_j})$ 
11:    $zddISOP = ISOP(POS)$ 
12:   remove negated literals and duplicates from  $zddISOP$ 
13:   add products to  $R$ 
14:   return  $R$ 

```

Fig. 10. Our PA merging algorithm: Return a set of PAs that apply noncontrolling values to every input of a gate.

improves both the running time of our algorithm and the quality of the final result. Our algorithm satisfies the first three conditions and approximates the minimality by computing an irredundant sum of products (ISOP), as we describe in the succeeding paragraphs.

We can merge PAs by merely verifying that there are no conflicts to the assigned primary inputs of the SCC. In other words, we do not need to check for conflicts of every internal node. This greatly speeds our consistency checking procedure.

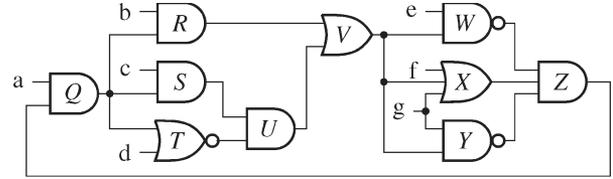
The argument for this is a proof by contradiction. Let two PAs  $A$  and  $B$  have nonconflicting controlling assignments to SCC inputs, and assume that some intermediate node  $I$  has conflicting values under assignments  $A$  and  $B$  (i.e., one is 0, the other 1; there is no conflict if either is  $\perp$ ). The gate that produces  $I$  must either have one input set to a controlling value or all inputs set to noncontrolling values. We can repeat the analysis on those inputs until we find conflicting inputs at SCC inputs, which contradicts the original assumption.

Merging PAs is a kind of binate covering problem because we must cover all gate inputs while avoiding conflicts. However, our need for a complete enumeration is atypical.

Fig. 10 shows our algorithm for merging PAs. We construct a BDD expressing our covering problem at the gates and the conflicts therein as a product of sums (POS). The covering at each gate input is encoded as a sum term that includes all PAs that can control that input. By definition, these are all noncontrolling input assignments, since otherwise, the PA would have continued past this gate. To set the gate's output to a deterministic value, we need to select PAs covering all the gate's inputs, hence, the sum of products. However, we must account for the PAs containing conflicting and, therefore, noncompatible assignments to the inputs into the SCC. We thus augment our POS expression with clauses that capture the conflicts as pairwise sums of the negation of PAs that conflict.

We then use the Minato–Morreale algorithm [35] to generate an ISOP in ZDD [36] form and use these to continue propagation. Note that the addition of conflicts causes the ISOP to contain negated terms, which we discard.

Fig. 9 shows an example. The five PAs in Fig. 9(a) control the inputs of the three-input AND gate in Fig. 9(b). Our merging algorithm (Fig. 10) starts by expressing the constraint at the AND gate with the POS in Fig. 9(c): Each input must be controlled



(a)

Label	Assignment	Frontier	At Frontier	Acyclic
$p_0$	$\{a = 0\}$	$\{\}$		✓
$p_1$	$\{b = 0\}$	$\{V\}$	$R = 0$	
$p_2$	$\{c = 0\}$	$\{V\}$	$U = 0$	
$p_3$	$\{d = 1\}$	$\{V\}$	$U = 0$	
$p_4$	$\{e = 0\}$	$\{Z\}$	$W = 1$	
$p_5$	$\{f = 1\}$	$\{Z\}$	$X = 1$	
$p_6$	$\{g = 0\}$	$\{Z\}$	$Y = 1$	
$p_7$	$\{g = 1\}$	$\{Z\}$	$X = 1$	

(b)

$$(p_1)(p_2 + p_3) \implies (p_1 p_2) + (p_1 p_3)$$

(c)

$$(p_4)(p_5 + p_7)(p_6)(\overline{p_6} + \overline{p_7}) \implies (p_4 p_5 p_6)$$

(d)

Product Term	Assignment
$p_0$	$\{a = 0\}$
$p_1 p_2$	$\{b = 0, c = 0\}$
$p_1 p_3$	$\{b = 0, d = 1\}$
$p_4 p_5 p_6$	$\{e = 0, f = 1, g = 0\}$

(e)

Fig. 11. (a) PA extraction on a small cyclic circuit. (b) PAs from applying controlling values to each input in isolation. All frontiers are either gate  $V$  or gate  $Z$ . (c) POS and final ISOP for frontier gate  $V$ . (d) POS and ISOP for  $Z$ . (e) Minimal set of PAs that reproduce all combinational behavior.

by at least one PA (the first three terms) and conflicting PAs (i.e., those that insist on contradictory assignments to inputs: here,  $p_1$  sets  $b = 0$  and  $p_4$  sets  $b = 1$ ; therefore, both  $p_1$  and  $p_4$  are illegal together) are prohibited. Next, these constraints are transformed to the ISOP in Fig. 9(d). Finally, negations are removed from each term in the ISOP, leading to the new set of PAs in Fig. 9(e). By construction, each of these PAs controls all three gate inputs and has no conflicting input assignments.

#### D. Another Example

We will use the cyclic circuit in Fig. 11 to illustrate how we use frontiers to extract PAs, how negated literals arise, and how we deal with them.

We start by applying a controlling value to each input separately. Fig. 11(b) shows the results. Note that when  $a$  is 0, the circuit is combinational since the feedback path is broken; therefore, we include the assignment  $\{a = 0\}$  as part of our minimal cover and will not consider any further assignments that contain  $\{a = 0\}$  (Theorem 4).

Consider setting  $b = 0$ . Although this is a controlling value for gate  $R$  (its output becomes 0 regardless of  $Q$ ), by itself, this is not enough to force the whole circuit to behave combinational because a 0 on  $R$  is a noncontrolling value on the OR gate  $V$ . Each of the assignments  $c = 0$  and  $d = 0$  also have a

frontier of  $V$ . A similar analysis shows different assignments to  $e$ ,  $f$ , and  $g$ , all yielding  $Z$  as their frontier.

The SCC input  $g$  has both 0 and 1 present as controlling assignments since it is connected to a NAND and an OR. Constructing a PA that includes such conflicting assignments is meaningless because the circuit could never reach such a state. Our algorithm tracks and caches conflicting PAs to guard against composing a PA from such assignments. As we stated previously, positive and negated literals in our initial POS indicate the presence or absence of PAs, respectively, not inverted assignments.

Next, we analyze the frontiers we have obtained from logic simulation. Only two gates  $V$  and  $Z$  appear in any frontier; we will attempt to set the outputs of these gates by judiciously combining sets of PAs that might completely define values at inputs of these gates. At every frontier gate, we compose a covering problem in the form of a POS, where each sum represents candidate PAs that define a given input of that gate. To this POS, we add pairwise conflicts between PAs that cannot be merged.

At gate  $V$ , the top input can only be defined by assignment  $p_1$ ; therefore, this becomes the first sum term in our POS: Fig. 11(c). The lower input can be defined by either of  $p_2$  or  $p_3$ ; therefore, we add  $(p_2 + p_3)$  as our second sum term. We note that none of these assignments conflict; therefore, there is no need to add any additional assignments. As a matter of computation runtime, however, we have found that adding conflicts does not materially affect the subsequent AllSat computation. The alternative, which is to compute and add only relevant conflicts at every frontier gate input, was found empirically to be more expensive. We store conflicting assignments in a cache which we update as we create new assignments. These are added to all POS expressions. This is not shown in Fig. 11 for brevity, where we only show relevant conflicts. A similar analysis at gate  $Z$  yields the POS in Fig. 11(d).

Our algorithm now computes all satisfying assignments to each of the POS expressions at frontier gates. We remove negated literals as well as identical products from within each sum. The output of this computation is shown next to each POS in Fig. 11(c) and (d). This computation yields three new PAs. Each leads to an empty frontier and (therefore) an acyclic circuit. Finally, our algorithm terminates and returns the PAs shown in Fig. 11(e).

## VIII. FROM COMBINATIONAL COVERS TO SCHEDULES

Previously, we showed how to find a small set of PAs that together cover all the combinational behavior of an acyclic circuit. Each such PA implies an acyclic circuit fragment that corresponds to a schedule, but it is only a partial schedule for the original circuit. In this section, we describe a heuristic algorithm for merging these partial schedules into one that can be used to synthesize the final circuit using the procedure we described in Section V.

Fig. 12 shows the algorithm for merging schedule  $s'$  to the “end” of schedule  $s$ . It strives to use existing gates in  $s$  to implement the function of the added schedule  $s'$  without introducing a cycle. It tries to match each gate in  $s'$  with the

```

1: function MERGE( $s, s'$ )
2:   Clear mapping  $m$ 
3:   for each gate  $g'$  in  $s'$  in scheduled order do
4:     for each gate  $g = g'$  in  $s$  in scheduled order do
5:       if for all drivers  $d'$  of  $g'$ ,
6:          $m(d')$  appears earlier than  $g$  in  $s$  then
7:         Set  $m(g') = g$ 
8:         continue with next  $g'$ 
9:       (Here, we did not find an appropriate match for  $g'$ )
10:      Append  $g'$  to  $s$ 
11:      Set  $m(g')$  to this newly-added gate

```

Fig. 12. Schedule merging algorithm.

earliest identical one in  $s$  that would not create a cycle and adds a new copy of the gate if no suitable one is found.

The mapping  $m$  (initialized in line 2, accessed in line 6, and updated in lines 7–11) prevents cycles in the generated circuit. It records the most recently used copy of each gate  $g$ ; therefore, the test in line 6 guarantees that every input wire to the gate  $g'$  comes from gates earlier in the schedule. By maintaining this invariant, the resulting schedule always implies an acyclic circuit since a gate’s inputs always come from earlier gates.

This algorithm is a heuristic in two ways. First, it can only place new gates later in the schedule  $s$ , not earlier. This problem is ameliorated by having the main algorithm try combining  $s$  and  $s'$  in both orders and picking the smallest, but this is not a complete solution. More serious is the choice in line 10 to always add the new gate at the end of schedule  $s$ . Other, earlier choices are possible, but it is not clear which would lead to a smaller overall circuit. The optimal merging problem seems very difficult however, and given that our overall algorithm is already potentially exponential, it is not clear whether it is worth trying to solve exactly. Fortunately, a nonoptimal solution is still correct, only larger.

## IX. EXPERIMENTAL RESULTS

We implemented our algorithm in C++ and tested it on a variety of circuits. We used the CU decision diagram (CUDD) package [37] for BDD construction, ISOP calculation, and ZDD computations. We present results in Table II. The first four circuits come from Esterel programs [17] and contain simple loops. The rest are more complex, coming from Riedel’s *cyclify* [6]. We used a Pentium D machine running dual-core 3.0-GHz Intel processors with an 800-MHz front side bus and 1-GB RAM running Windows XP.

Table II characterizes our examples and how our algorithm performed on them. The “Before” and “After” columns list the total number of gates in the circuit before and after our algorithm runs, providing a rough estimate of circuit complexity and area. The “in SCC” column lists the number of gates in the SCC of the input circuit (each circuit had exactly one nontrivial SCC). These are the only really challenging gates in each circuit. The “PAs” columns list the number of PAs our algorithm considered while running (“Tested”) and the number of acyclic PAs it found necessary to cover all combinational behavior (“Acyclic”). The Tested number provides a rough measure of how well our algorithm is able to prune the search space; testing fewer PAs takes less time. The Acyclic number

TABLE II  
EXPERIMENTAL RESULTS

Example	Gates			PAs		Time
	Before	After	in SCC	Tested	Acyclic	
arbiter2	96	143	10	10	6	0.09s
arbiter4	178	263	20	20	12	0.10
arbiter5	213	310	25	24	14	0.12
arbiter6	248	366	30	28	16	0.12
arbiter7	283	422	35	32	18	0.14
arbiter8	318	478	40	36	20	0.17
cyclic1	101	148	6	6	6	0.07
cyclic7	23	34	4	2	2	0.07
cyclic8	18	27	3	2	2	0.07
dacex	8	17	5	4	2	0.09
dc2a3	170	341	58	252	29	0.26
dc2o	64	114	28	45	34	0.11
ex1a2	427	1050	97	196	13	0.26
ex1o	150	434	47	1365	173	1.95
ex1o2	415	986	87	196	13	0.23
expa2	454	1761	159	18236	381	8.14
garya2	854	1329	88	313	11	0.37
in3a2	618	826	111	178	14	0.32
planeta2	820	1906	130	1032	23	0.75
s1488a2	999	2151	135	600	95	0.64
s1488a3	773	1457	115	2681	135	1.62
s1488o	272	428	61	1316	124	3.12
s1488o2	974	2052	123	600	95	0.64
s386o	77	137	20	46	11	0.10
ssea2	178	323	54	36	4	0.09
sseo	72	140	26	122	17	0.17
t4o	55	100	12	13	11	0.06
table3a2	2111	2289	151	2443	50	2.82
table3o2	2097	2269	146	2443	50	2.84

TABLE III  
COMPARISON WITH EDWARDS' [21]

Example	Gates		Edwards [21]		Our Approach		Acyclic PAs
	Total	SCC	PAs	Time	PAs	Time	
arbiter5	213	25	257	1.3s	25	0.1s	14
arbiter6	248	30	745	8	29	0.1	16
arbiter7	283	35	2205	69	33	0.2	18
arbiter8	318	40	6581	656	37	0.3	20
expo	124	69	54517	2868	23260	2.0	338
ex1o	150	47	43777	2341	232	1.0	10
garyo	177	32	-	> 1h	290	0.6	11
planeto	253	51	-	> 1h	1489	0.3	22
s1488o	272	61	-	> 1h	588	0.2	89
table3o	311	49	-	> 1h	3604	1.0	38

is more of a function of the structure of the circuit and suggests how well PAs characterize the combinational behavior of the circuit. The last column reports the runtimes in seconds, which includes both time to find the PAs and assemble the fragments they induce into the final circuit.

Table III compares our approach with an earlier version of the algorithm from Edwards [21]. Our algorithm consistently runs over two orders of magnitude faster and is able to process many more candidate PAs in less time, which we attribute to removing the more expensive operations such as the superset check against known combinational PAs.

We were not able to compare our results with those of Gupta and Selvidge [23] because they did not test publicly available circuits. However, unlike ours, their approach introduces latches to break combinational loops. Also, our approach analyzes the circuit for combinational behavior, producing conditions under which the circuit behaves combinational; their

algorithm does not. They do not list the number of gates in SCC loops, and they report that the number of loops they can process per second varies by over a factor of 50 (from 4 to 220). Also, they count each feedback path as a loop where we count SCCs, which may contain many loops.

## X. CONCLUSION

We presented a new method for transforming a cyclic circuit into an equivalent acyclic one that retains all the combinational behavior of the cyclic version. This is useful when cyclic circuits arise from high-level synthesis [17], [18] or are generated by algorithms designed to produce cyclic circuits [6], yet need to be fed to synthesis and analysis tools that insist on acyclic circuits.

As an intermediate result, we characterize sufficient and necessary conditions for the original circuit to behave combinational. Our method only guarantees that the generated circuit produces the same outputs for inputs that produce combinational behavior; it implicitly assumes that the original cyclic circuit is only used under such conditions. These combinational conditions could be used to check, either formally or in simulation, whether an environment would ever present illegal input patterns to the original cyclic circuit and lead it to behave noncombinational.

Our algorithm produces an acyclic circuit by combining fragments corresponding to the evaluation schedules of the original circuit. This is advantageous because it composes the acyclic circuit from the same gates from which the original cyclic circuit was built. This is useful if the original circuit was produced using specific library cells since our final circuit will not require any different ones.

Our algorithm analyzes all possible inputs into SCCs without considering whether such patterns can actually occur in the actual circuit (i.e., whether they are controllability don't cares). This speeds the analysis by avoiding the need for an image computation on the surrounding circuit, but it is possible that considering the don't-care set would reduce the number of PAs we consider, further speeding the search and reducing the size of the final acyclic circuit. We have not explored the tradeoff between computing don't cares and reducing the number of PAs and potential reduction in the computation and size of the resulting acyclic circuit since such a tradeoff depends on the context of the cyclic circuit.

## ACKNOWLEDGMENT

The authors would like to thank A. Shinnar for providing the initial inspiration for some of the algorithms in this paper; it was his idea to cover schedules with circuits. The authors would also like to thank S. Malik, M. Kishinevsky, and L. Henry-Gérard for their contributions, G. Berry and T. Shiple for the discussions, M. Riedel and J. Bruck for their seminal research on the synthesis of cyclic circuits, which reignited interest in these circuits and provided some of the benchmarks used in this research, and A. Mishchenko for fixing a number of things in the CUDD package, which we used, and for also porting it to Windows.

## REFERENCES

- [1] S. Malik, "Analysis of cyclic combinational circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 7, pp. 950–956, Jul. 1994.
- [2] L. Stok, "False loops through resource sharing," in *Proc. IEEE/ACM ICCAD*, San Jose, CA, Nov. 1992, pp. 345–348.
- [3] G. Berry, "Esterel on hardware," *Philos. Trans. Roy. Soc. London A, Math. Phys. Sci.*, vol. 339, no. 1652, pp. 87–103, Apr. 1992. Mechanized Reasoning and Hardware Design.
- [4] W. H. Kautz, "The necessity of closed circuit loops in minimal combinational circuits," *IEEE Trans. Comput.*, vol. C-19, no. 2, pp. 162–164, Feb. 1970.
- [5] R. L. Rivest, "The necessity of feedback in minimal monotone combinational circuits," *IEEE Trans. Comput.*, vol. C-26, no. 6, pp. 606–607, Jun. 1977.
- [6] M. D. Riedel and J. Bruck, "The synthesis of cyclic combinational circuits," in *Proc. 40th Des. Autom. Conf.*, Anaheim, CA, Jun. 2003, pp. 163–168.
- [7] D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.*, vol. 257, no. 3/4, pp. 161–190, Mar./Apr. 1954. 275-303, also MIT RLE-TR-274 (Jan. 10, 1954).
- [8] D. A. Huffman, "The design and use of hazard-free switching networks," *J. Assoc. Comput. Mach.*, vol. 4, no. 1, pp. 47–62, Jan. 1957.
- [9] S. H. Unger, "Hazards and delays in asynchronous sequential switching circuits," *IRE Trans. Circuit Theory*, vol. 6, no. 1, pp. 12–25, Mar. 1959.
- [10] C. J. Myers, *Asynchronous Circuit Design*. New York: Wiley, 2001.
- [11] E. B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM J. Res. Develop.*, vol. 9, no. 2, pp. 90–99, Mar. 1965.
- [12] M. Yoeli and S. Rinon, "Application of ternary algebra to the study of static hazards," *J. Assoc. Comput. Mach.*, vol. 11, no. 1, pp. 84–97, Jan. 1964.
- [13] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg, "A three-value computer design verification system," *IBM Syst. J.*, vol. 8, no. 3, pp. 178–188, 1969.
- [14] J. A. Brzozowski and M. Yoeli, "On a ternary model of gate networks," *IEEE Trans. Comput.*, vol. C-28, no. 3, pp. 178–184, Mar. 1979.
- [15] R. E. Bryant, "Race detection in MOS circuits by ternary simulation," in *Proc. Int. Conf. VLSI Des. (VLSI)*, Trondheim, Norway, Aug. 1983, pp. 85–95.
- [16] T. R. Shiple, G. Berry, and H. Touati, "Constructive analysis of cyclic circuits," in *Proc. Eur. Des. Test Conf.*, Paris, France, Mar. 1996, pp. 328–333.
- [17] G. Berry, *The Constructive Semantics of Pure Esterel*, 1999. draft book. [Online]. Available: <http://www.esterel-technologies.com/files/book.zip>
- [18] G. Berry, "The foundations of Esterel," in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA: MIT Press, 2000.
- [19] N. Halbwachs and F. Maraninchi, "On the symbolic analysis of combinational loops in circuits and synchronous programs," in *Proc. 21st Euromicro Conf.*, Como, Italy, Sep. 1995.
- [20] K. S. Namjoshi and R. P. Kurshan, "Efficient analysis of cyclic definitions," in *Computer Aided Verification*, vol. 1633. Trento, Italy: Springer-Verlag, Jul. 1999, pp. 394–405.
- [21] S. A. Edwards, "Making cyclic circuits acyclic," in *Proc. 40th Des. Autom. Conf.*, Anaheim, CA, Jun. 2003, pp. 159–162. [Online]. Available: <http://doi.acm.org/10.1145/775832.775874>
- [22] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
- [23] A. Gupta and C. Selvidge, "Acyclic modeling of combinational loops," in *Proc. IEEE/ACM ICCAD*, San Jose, CA, Nov. 2005, pp. 343–347.
- [24] O. Neiroukh, S. A. Edwards, and X. Song, "An efficient algorithm for the analysis of cyclic circuits," in *Proc. Symposium VLSI (ISVLSI)*, Karlsruhe, Germany, Mar. 2006, pp. 303–308. [Online]. Available: <http://dx.doi.org/10.1109/ISVLSI.2006.18>
- [25] T. R. Shiple, "Formal analysis of synchronous circuits," Ph.D. dissertation, Univ. California, Berkeley, CA, Oct. 1996. memorandum UCB/ERL M96/76.
- [26] J. A. Brzozowski and C.-J. H. Seger, *Asynchronous Circuits*. New York: Springer-Verlag, 1995.
- [27] J.-L. Lassez, V. L. Nguyen, and E. A. Sonnenberg, "Fixed point theorems and semantics: A folk tale," *Inf. Process. Lett.*, vol. 14, no. 3, pp. 112–116, May 1982.
- [28] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
- [29] P. J. Koutas and T. C. Hu, "Shortest string containing all permutations," *Discrete Math.*, vol. 11, no. 2, pp. 125–132, 1975.
- [30] S. P. Mohanty, "Shortest string containing all permutations," *Discrete Math.*, vol. 31, no. 1, pp. 91–95, 1980.
- [31] M. Cai, "A new bound on the length of the shortest string containing all  $r$ -permutations," *Discrete Math.*, vol. 39, no. 3, pp. 329–330, May 1982.
- [32] R. Erra, N. Lygeros, and N. Stewart, "On minimal strings containing the elements of  $S_n$  by decimation," in *Discrete Math. and Theoretical Comput. Sci.*, Paris, France, Jul. 2001, vol. AA, pp. 165–176.
- [33] F. Bourdoncle, "Efficient chaotic iteration strategies with widenings," in *Formal Methods in Programming and Their Applications: International Conference Proceedings*, vol. 735. Novosibirsk, Russia: Springer-Verlag, Jun. 1993. [Online]. Available: <http://www.ensmp.fr/~bourdonc/fmpa93.ps.Z>
- [34] S. A. Edwards and E. A. Lee, "The semantics and execution of a synchronous block-diagram language," *Sci. Comput. Program.*, vol. 48, no. 1, pp. 21–42, Jul. 2003. [Online]. Available: [http://dx.doi.org/10.1016/S0167-6423\(02\)00096-5](http://dx.doi.org/10.1016/S0167-6423(02)00096-5)
- [35] S. Minato, "Fast generation of irredundant sum-of-products forms from binary decision diagrams," in *Proc. SASIMI*, Kobe, Japan, Apr. 1992, pp. 64–73.
- [36] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," in *Proc. 30th Des. Autom. Conf.*, Dallas, TX, 1993, pp. 272–277.
- [37] F. Somenzi, *CUDD: CU Decision Diagram Package Release*, 1998. Available: <http://vlsi.colorado.edu/~fabio/CUDD/>



**Osama Neiroukh** received the B.Eng. degree in computer systems engineering from Bristol University, Bristol, U.K., in 1991, the M.S. degree from the University of Tennessee, Knoxville, in 1995, and the Ph.D. degree from Portland State University, Portland, OR, in 2008.

He has been with Intel Corporation, Hillsboro, OR, since 1996, where he is a Staff Engineer working on CPU design.



**Stephen A. Edwards** (S'93–M'97–SM'06) received the B.S. degree in electrical engineering from the California Institute of Technology, Pasadena, in 1992, and the M.S. and Ph.D. degrees in electrical engineering from the University of California, Berkeley, in 1994 and 1997, respectively.

He has been with the Department of Computer Science, Columbia University, New York, NY, since 2001 after a three-year stint with Synopsys, Inc., Mountain View, CA. His research interests include embedded system design, domain-specific languages, and compilers.



**Xiaoyu Song** (M'92–SM'01) received the Ph.D. degree from the University of Pisa, Pisa, Italy, in 1991.

From 1992 to 1999, he was with the faculty of the University of Montreal, Montreal, QC, Canada. In 1998, he was a Senior Technical Staff with Cadence, San Jose. Since 1999, he has been with the faculty of the Department of Electrical and Computer Engineering, Portland State University, Portland, OR. His research interests include synthesis, verification of digital system designs, and formal methods.

He served as an Associate Editor of IEEE

TRANSACTIONS ON CIRCUITS AND SYSTEMS and IEEE TRANSACTIONS ON VLSI SYSTEMS.