View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Columbia University Academic Commons

An Efficient Algorithm for the Analysis of Cyclic Circuits

Osama Neiroukh*
Intel Corporation
osaman@ichips.intel.com

Stephen A. Edwards†
Columbia University
sedwards@cs.columbia.edu

Xiaoyu Song
Portland State University
song@ece.pdx.edu

Abstract

Compiling high-level hardware languages can produce circuits containing combinational cycles that can never be sensitized. Such circuits do have well-defined functional behavior, but wreak havoc with most logic synthesis and timing tools, which assume acyclic combinational logic. As such, some sort of cycle-removal step is usually necessary for handling these circuits.

We present an algorithm able to quickly and exactly characterize all combinational behavior of a cyclic circuit. It iteratively examines the boundary between gates whose outputs are and are not defined and works backward to find additional input patterns that make the circuit behave combinationally. It produces a minimal set of sets of assignments to inputs that together cover all combinational behavior. This can be used to restructure the circuit into an acyclic equivalent, report errors, or as an optimization aid.

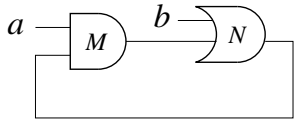
Experiments show our algorithm runs several orders of magnitude faster than existing ones on real-life cyclic circuits, making it useful in practice.

1 Introduction

Cyclic circuits can be produced inadvertently during high-level synthesis and are also the most compact representation for certain circuits such as arbiters [11]. For certain input patterns, such circuits are well-behaved (functional), i.e., do not exhibit oscillations or state-holding behavior. Despite this, most circuit analysis tools forbid the presence of cycles. The central challenge of cyclic circuits is their data-dependent evaluation order, meaning their gates have no topological order. This causes difficulties for many tools such as static timing analyzers that rely on such a static order. Furthermore, applying regular logic simulation to these circuits is cumbersome.

*Neiroukh is sponsored by Intel Corporation

†Edwards is supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program.



a	b	M	N
0	0	0	0
0	1	0	1
1	0	X	X
1	1	1	1

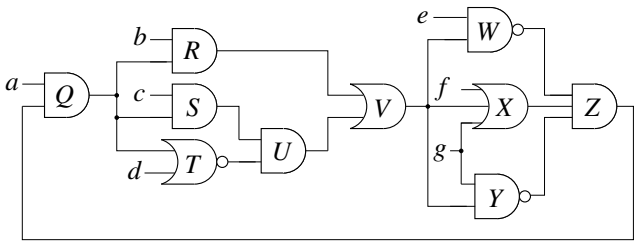
Figure 1: A trivial cyclic circuit and its truth table

Consider the small cyclic circuit in Figure 1. From its truth table, we see the circuit is well-behaved unless $a = 1$ and $b = 0$. For all other input patterns, the circuit behaves combinationally because the feedback loop is broken by a controlling input on one of the gates. A *partial assignment* is an assignment to one or more inputs to the loop; $\{a = 0\}$ is one such partial assignment. Our algorithm produces a set of partial assignments that provide a concise representation of the conditions under which a cyclic circuit is well-behaved. For example, the set of partial assignments $\{\{a = 0\}, \{b = 1\}\}$ constitutes necessary and sufficient conditions for combinational operation of the circuit in Figure 1: at least one of these must hold in order for the circuit to operate functionally.

In this paper, we present a novel algorithm that can rapidly identify all possible combinational behavior of a cyclic circuit. The algorithm takes a circuit containing one or more loops and produces a set of partial assignments that represent every condition under which the circuit behaves combinationally. Our algorithm relies on the fact that gates such as ANDs and ORs have controlling inputs (0 and 1 respectively) that break feedback loops to aggressively prune the search space. The set of partial assignments our algorithm produces can be used to rule out non-constructive operation of circuits produced by high level compilers such as Esterel [2], or they can be used to create an equivalent acyclic circuit [5].

2 An Example

Consider the cyclic circuit in Figure 2. In general, our algorithm analyzes a circuit one strongly-connected com-



(a) A cyclic circuit

Assignment	Frontier	At Frontier	Acyclic
$\{a = 0\}$	$\{\}$		$\surd$
$\{b = 0\}$	$\{V\}$	$R = 0$	
$\{c = 0\}$	$\{V\}$	$U = 0$	
$\{d = 1\}$	$\{V\}$	$U = 0$	
$\{e = 0\}$	$\{Z\}$	$W = 1$	
$\{f = 1\}$	$\{Z\}$	$X = 1$	
$\{g = 0\}$	$\{Z\}$	$Y = 1$	
$\{g = 1\}$	$\{Z\}$	$X = 1$	

(b) First step: applying controlling values to each input in isolation

Gate	Assignment	Frontier	Acyclic
V	$\{b = 0, c = 0\}$	$\{\}$	$\surd$
V	$\{b = 0, d = 1\}$	$\{\}$	$\surd$
Z	$\{e = 0, f = 1, g = 0\}$	$\{\}$	$\surd$

(c) Second step: Merged partial assignments from first step

$\{a = 0\}$
$\{b = 0, c = 0\}$
$\{b = 0, d = 1\}$
$\{e = 0, f = 1, g = 0\}$

(d) Final result: A minimal set of partial assignments that produce all combinational behavior

Figure 2: Illustration of our algorithm.

ponent (SCC) at a time, but this example consists only of a single SCC.

Our goal is to find a small set of partial assignments 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 a subset of one of our partial assignments.

Our algorithm begins by considering applying a controlling value to each input in isolation. Such a controlling value—a 0 input on an AND gate, 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 combinationally. We formalize this later in Theorem 1.

Figure 2b summarizes the results of these initial assignments. First, note that when the a input is 0, the circuit is always combinational because 0 is a controlling value on gate Q, effectively breaking the $Z \rightarrow Q$ feedback loop. We include the assignment $\{a = 0\}$ as part of our minimal cover and will not consider any further assignments that contain $\{a = 0\}$ (Theorem 2).

Consider what happens when we set $b = 0$. 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 combinationally because a 0 on R is a non-controlling value on the OR gate V. We refer to all such gates as the *frontier* induced by a partial assignment (see Definition 4) because they define the boundary between combinational and possibly non-combinational behavior. Think of such gates as being the cause of a logjam; the next step in our algorithm is to break logjams.

The key step in our algorithm, and its main improvement over Edwards [5], attempts to break these logjams by looking for promising combinations of partial assignments that affect the same frontier gates. 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 partial assignments that might completely define values at inputs of these gates.

To break the logjam at V, we consider subsets of the three partial assignments that affected its inputs, i.e., $\{b = 0\}$, $\{c = 0\}$, and $\{d = 1\}$. By definition, each of these set at least one of the inputs to V to a non-controlling value (0, because V is an OR gate). We can break the logjam by setting *all* of V's inputs to non-controlling values, i.e., by setting $R = 0$ and $U = 0$. To set $R = 0$, we need $b = 0$, but there are two ways to set $U = 0$: $c = 0$ and $d = 1$. Thus we decide to consider the partial assignments $\{b = 0, c = 0\}$ and $\{b = 0, d = 1\}$ in the next step.

Similar reasoning about frontier gate Z leads us to want to set $W = 1$, $X = 1$, and $Y = 1$. There appear to be two ways to do this by combining existing assignments, i.e., through $\{e = 0, f = 1, g = 0\}$ and $\{e = 0, g = 1, g = 0\}$. However, the latter one is nonsensical because we cannot set g to be both 0 and 1 simultaneously. We consider the partial assignments $\{g = 0\}$ and $\{g = 1\}$ to be *in conflict* and refuse to merge them.

Figure 2c lists the three new partial assignments we consider along with the frontier gate that induced them. Each partial assignment leads to an empty frontier and (therefore) an acyclic circuit. Our algorithm terminates and returns the partial assignments listed in Figure 2d.

3 Prior Work

In 1970, Kautz [7] showed that the minimal form of certain circuits contained combinational loops. Rivest [11] came to a similar conclusion, suggesting that combinational loops are more than just a nuisance. Stok [13] observed how they can arise from resource-sharing in high-level synthesis, motivating Malik's work [8] on analyzing combinational

circuits, a forerunner of our work. Malik showed an equivalence between combinational cyclic circuits and least-fixed-points in three-valued simulation, an idea that Shiple, Berry, and Touati [12] applied to the Esterel language [2,3], whose hardware translation [1] often produces combinational cycles. Their approach uses a symbolic state-space traversal followed by an $O(n^2)$ replication procedure to remove cycles. Our algorithm pays more attention to both the structure and function of the circuit and, when coupled with the resynthesis technique of Edwards [5], produces smaller circuits. The BDD-based algorithm of Halbwachs and Maraninchi [6] takes a brute-force approach, ignoring the structure of the circuit. Namjoshi and Kurshan [9] take a very different approach, showing that any fixed-point is interesting, not just the least. Their analysis merely answers whether a circuit is combinational.

Recently, Riedel and Bruck [10] applied 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 fairly expensive BDD construction; our algorithm could potentially be used in that setting. More practically, the cyclic combinational circuits they generate have topologies complex enough to stymie the de-cyclification algorithm of Edwards [5], which our work builds on.

Our algorithm is a drop-in replacement for the first half of Edwards [5], which enumerates all the conditions under which a circuit is combinational then merges the resulting circuit fragments. Edwards's algorithm gets mired in considering using every input to an SCC to break a cycle; our algorithm is much more shrewd. When it finds a gate that might participate in a non-combinational cycle, it uses the behavior of simulations it performed earlier to work backward to identify primary inputs that will break the cycle. This reduces the number of input patterns the algorithm considers and hence greatly reduces its running time.

4 Notation and Definitions

This section defines the basic terminology necessary for explaining material in this paper.

We represent circuits with a *directed graph* (*digraph*). A digraph G is a pair (V, E) where V is a set of vertices and E is a set of edges. An *edge* is an element of $V \times V$ with distinct vertices. We represent a circuit as a digraph whose vertices correspond to gates and whose edges correspond to nets. A *controlling value* for a gate G is the value that applied to any input of G uniquely determines G's output independent of other inputs. To simplify our exposition, we only consider simple logic gates: NOT, AND/NAND, and OR/NOR. This is not a limitation as more complex gates can be represented as combinations of these gates.

Definition 1. *A strongly connected component (*SCC*) of a digraph $G = (V, E)$ is a maximal subset of vertices $C \subseteq V$ such that any vertex in C is reachable from any other vertex in C. Inputs of an SCC are inputs of gates that are part of the SCC that are not driven by gates inside the SCC.*

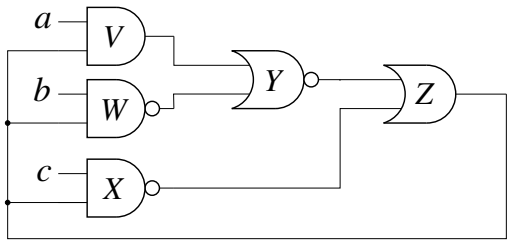


Figure 3: Cyclic circuit for illustrating definitions

Figure 3 shows a circuit with a single SCC. Nets a, b, and c are inputs to the SCC. When analyzing a circuit, we first decompose it into SCCs using a standard algorithm [4]. If the input circuit contains more than one SCC, we consider each SCC separately in a topological order.

Our analysis methodology and logic simulation use a ternary domain consisting of $\{0, 1, X\}$ where X denotes an unknown digital value.

Definition 2 (Malik [8])**.** *A circuit is* combinational for an input assignment *if three-valued simulation starting with all internal nodes set to X resolves the output of every gate in the circuit to either 0 or 1 under the assignment.*

Literature on cyclic circuits also refers to this behavior as "well-behaved" and "constructive" [12]. Combinational behavior is equivalent to stating that the circuit behaves as if it were acyclic with no X's and no oscillations.

Definition 3. *A* partial assignment *(PA) is a set of assignments to one or more nets of a circuit.*

In this work, we shall be only concerned with partial assignments to inputs of SCCs. A valid PA for the circuit in Figure 3 is an assignment to one or more of the inputs $\{a, b, c\}$, such as $\{a = 0\}, \{b = 0, c = 1\}$, or $\{b = 1, c = 1\}$.

Definition 4. *The* controllability frontier *of a PA, or* frontier *for short, is the set of gates that have at least one input assigned but whose output is X.*

The frontier captures the notion of a boundary between gates whose output is defined and those whose output is not. When calculating the frontier for a PA, we use ternary simulation to propagate the SCC inputs as far as possible then check for cyclic behavior. For example, for Figure 3,

Partial Assignment	Frontier
$\{a = 1\}$	$\{V\}$
$\{a = 0, b = 1\}$	$\{Y, W\}$
$\{c = 0\}$	$\{\}$

5 Our Algorithm

Here, we describe our algorithm for rapidly extracting a cover for all combinational behavior of a cyclic circuit.

5.1 Theoretical Background

We start with a set of theorems that are key to the correctness and efficiency of our algorithm. The first two are due to Edwards [5].

Theorem 1 (Edwards [5])**.** *For a circuit with a strongly-connected component (SCC) to behave combinationally, at least one input to a gate in the SCC must be driven to a controlling value.*

Controlling assignments to SCC inputs for the circuit in Figure 3 are $a = 0$, $b = 0$, and $c = 0$. Theorem 1 tells us that at least one of these is required for combinational behavior. We use this property 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 combinations of one or more of these PAs.

Theorem 2 (Edwards [5])**.** *If a partial assignment* p *is combinational, then any further assignments that do not contradict any in* p *can also be computed combinationally by the circuit fragment implied by* p*.*

Consider the PA $\{c = 0\}$ applied to Figure 3. This breaks the connectivity of the SCC, making the circuit behave combinationally. This theorem indicates that additional assignments beyond $\{c = 0\}$ cannot reverse the combinational behavior already implied by this PA. This theorem allows 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 5.3.

This relates frontiers and combinational behavior:

Theorem 3. *A PA makes a circuit 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 have an output of X. From Theorem 1, we know that at least one gate must be driven by a controlling value for combinational behavior. If none have an output of X, then the circuit under that PA is combinational by definition.

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

Our algorithm records the frontier associated with each PA and uses them to look for opportunities to merge PAs to extend their frontiers.

Algorithm 1 Given a circuit, return a minimal set of PAs that together cover all combinational behavior.

```
1:  A = ∅                          ▷ Set of acyclic PAs, the eventual result
2:  K = ∅                          ▷ All known cyclic PAs, used for merging
3:  Clear F                        ▷ A map from frontier gate → set of PAs
4:  while circuit has SCCs
5:      Find next SCC
6:      P = controlling values for SCC inputs        ▷ Initial PAs
7:      while P ≠ ∅
8:          G = ∅                  ▷ Frontier gates for this iteration
9:          foreach p ∈ P          ▷ 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:                 foreach gate g in the frontier induced by p
16:                     add g to G         ▷ Record the frontier gate
17:                     add p to F(g)      ▷ Remember p induced g
18:         P = ∅                  ▷ Compute new candidate PAs
19:         foreach frontier gate g ∈ G
20:             if |F(g)| > 1 then     ▷ Need ≥ 2 PAs to merge
21:                 add each PA from mergeAtGate(K, g) to P
22: return A
```

5.2 Searching for combinational behavior

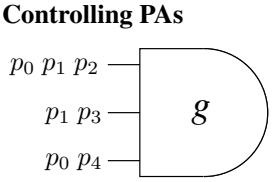
Algorithm 1 is our technique for identifying all combinational behavior. The algorithm takes a circuit with any number of SCCs and produces a set of PAs under which the circuit is combinational. These PAs control SCC inputs.

The algorithm attacks one SCC at a time (line 4), finding a minimal set of covering partial assignments for each. For each SCC, it begins by considering partial assignments that place a single controlling value on each SCC input (line 6), then enters into a loop (lines 7–21) in which it alternates between testing whether any of the currently-considered partial assignments (the set P) induce combinational behavior (lines 10–17) and attempting to merge already-observed partial assignments (the 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 partial assignments that affect frontier gates: if g is a gate, then $F(g)$ is the set of all partial assignments that put at least one non-controlling value at an input of g.

Algorithm 1 is guaranteed to find all combinational behavior within in 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.

Name	Assignment
p_0	$\{a = 1\}$
p_1	$\{b = 0, c = 1\}$
p_2	$\{c = 1, d = 1\}$
p_3	$\{e = 0\}$
p_4	$\{b = 1, f = 1\}$

(a) PAs



(b) How these PAs control the inputs of this gate

$$\begin{aligned} p_0 \cup p_1 &= \{a = 1, b = 0, c = 1\} \\ p_0 \cup p_3 &= \{a = 1, e = 0\} \\ p_2 \cup p_3 \cup p_4 &= \{b = 1, c = 1, d = 1, e = 0, f = 1\} \end{aligned}$$

(c) New PAs generated by merging

Figure 4: Merging PAs at a gate. If the five PAs in (a) control the three inputs on the gate (b), the merging algorithm (Algorithm 2) will generate three new partial assignments (c) by merging the five existing ones. By construction, each controls all three of the gate inputs.

5.3 Merging partial assignments

Here, we describe a key algorithm used by Algorithm 1: the generation of new partial assignments to break the logjam at a frontier gate. Given a set of PAs and a gate, Algorithm 2 generates a set of PAs that apply non-controlling values to *every* input of the gate, thus setting its output.

We store PAs in a simulated state that captures all assigned nodes and their values. Algorithm 1 only tries 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 X.

Consider the example in Figure 4. This shows a 3-input gate g that is a frontier gate for partial assignments $p_0, p_1, \ldots, p_4$. Note that a gate can only be a frontier for a PA if that PA puts a non-controlling value on one or more of the gate's inputs. We wish to consider merging these PAs in order to extend the frontier beyond g. A desirable merge of PAs at a gate g must satisfy the following:

i) *Gate Cover*: The PAs to be merged must define every input of g.

ii) *Consistency*: The PAs to be merged must not contain conflicting assignments to inputs. In the example in Figure 4, partial assignments p_1 and p_4 cannot be combined due to a conflicting assignment for b.

iii) *Completeness*: PAs must be merged such that all permissible combinations are considered. The example in Figure 4 provides some degrees of freedom to cover every input that must all be considered. This ensures that our final PAs encapsulate both necessary and sufficient conditions for combinational behavior.

Algorithm 2 Return a set of PAs that apply non-controlling values to every input of a gate

```
1: function MERGEATGATE(K, g)
2:     R = ∅                                   ▷ Generated set of PAs
3:     foreach input i of gate g
4:         p_i = PAs in K that set i and induce g as a frontier
5:         if p_i = ∅ then return ∅            ▷ Cannot control some input
6:     foreach P ∈ p_1 × p_2 × ··· × p_k       ▷ All combinations
7:         if the partial assignments in P do not conflict then
8:             add minimize(merge(P)) to R
9:     return R
```

iv) *Minimality*: The 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 the first and third gate input; p_4 only control the third). This condition is important for two reasons: it keeps the final output PAs as concise as possible by not including redundant conditions. Such redundancy burdens subsequent stages of the algorithm as it increases memory usage and makes testing of merge conditions against other candidate PAs more tedious.

We note that merging PAs is a sort of Binate Covering Problem (BCP) (it is covering because we must cover all gate inputs; it is binate because conflicts between PAs prevent certain combinations). However, the need for a complete enumeration is not a usual requirement in traditional BCP applications. In the context of merging PAs, the domain of the problem is rather small and makes enumeration tractable. We use an explicit enumeration algorithm with provisions for removing conflicts and minimizing merged PAs to eliminate any dominated PAs (Algorithm 2).

6 Experimental Results

We implemented our algorithm in C++ using the Standard Template Library and tested it on a number of cyclic circuits. We report execution times and the number of partial assignments considered compared to Edwards [5] in Table 1. The first four circuits come from Esterel programs [2] and contain simple loops. The rest are outputs of Riedel's *cyclify* [10] and are more complex. Our algorithm consistently runs more than two orders of magnitude faster than Edwards [5]. Also, our program is able to process many more candidate PAs in less time, which we attribute to removing the more expensive operations in Edwards, including the superset check against known-combinational PAs.

Circuit	Netlist Gates	SCC Gates	Edwards [5]		Our Approach		Acyclic PAs
			PAs considered	runtime	PAs considered	runtime	
arbiter5	213	25	257	1.3	25	0.1	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
exp	124	69	54517	2868	23260	2	338
ex1	150	47	43777	2341	232	1	10
gary	177	32	-	-	290	0.6	11
planet	253	51	-	-	1489	0.3	22
s1488	272	61	-	-	588	0.2	89
table3	311	49	-	-	3604	1	38

Table 1: Experimental Results: Runtimes are in seconds; a dash indicates the algorithm did not terminate after one hour.

7 Conclusions

We presented a new algorithm for identifying all the combinational behavior of a cyclic circuit. The algorithm is useful for evaluating cyclic specifications that often arise from high-level synthesis [2, 3]. One application of our algorithm is transforming cyclic combinational circuits to an acyclic equivalent; it replaces the first half of the procedure described by Edwards [5].

The chief contribution of our work is a speed improvement of several orders of magnitude over Edwards [5] due to much more clever pruning of the search space. It is therefore able to deal with practical-sized cyclic circuits.

Our algorithm analyzes all possible inputs into SCCs without considering whether such patterns can in fact occur in the original circuit (i.e., whether they are controllability don't-cares). This saves us from performing an image computation on the surrounding circuit, making the analysis much faster. However, it is possible that considering the don't-care set would reduce the number of PAs we consider and further speed the search. We have yet to explore the trade-off between computing don't-cares and reducing the number of PAs.

Although our algorithm performs quite well, it can be improved further. The current performance bottleneck arises when merging PAs at a frontier gate to produce more PAs to consider. Most of our PAs are generated here and most are later discarded. A more clever approach, perhaps Espresso-based, might reduce both the number of new PAs generated and the time it takes to derive them.

Independent of these further refinements, we have presented a practical algorithm that is able to quickly characterize all the combinational behavior of a realistic-sized cyclic circuit. Our intended application is the construction of an acyclic equivalent of a cyclic circuit to make it palatable to existing synthesis tools, but we believe our algorithm has other important applications in analysis and formal equivalence verification of cyclic circuits.

References

[1] G. Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–103, Apr. 1992. Issue 1652, Mechanized Reasoning and Hardware Design.

[2] G. Berry. The constructive semantics of pure Esterel. Draft book, 1999.

[3] G. Berry. *The foundations of Esterel*. MIT Press, 2000.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[5] S. Edwards. Making cyclic circuits acyclic. In *Proc. Design Automation Conference*, pages 159–162, 2003.

[6] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Proc. Euromicro*, pages 345–348, 1995.

[7] W. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Trans. Comput.*, C-19:162–164, Feb. 1970.

[8] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, July 1994.

[9] K. S. Namjoshi and R. P. Kurshan. Efficient analysis of cyclic definitions. In *Computer Aided Verification*, volume 1633 of *LNCS*, pages 394–405, Trento, Italy, July 1999.

[10] M. Riedel and J. Bruck. The synthesis of cyclic combinational circuits. In *Proc. Design Automation Conference*, pages 163–168, 2003.

[11] R. L. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Trans. Comp.*, 26(6):606–607, 1977.

[12] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proc. European Design and Test Conf.*, pages 328–333, 1996.

[13] L. Stok. False loops through resource sharing. In *Proc. International Conference on Computer-Aided Design*, pages 345–348, 1992.