

Making Cyclic Circuits Acyclic

Stephen A. Edwards*

Columbia University, Department of Computer Science
New York, NY 10027

sedwards@cs.columbia.edu

ABSTRACT

Cyclic circuits that do not hold state or oscillate are often the most convenient representation for certain functions, such as arbiters, and can easily be produced inadvertently in high-level synthesis, yet are troublesome for most circuit analysis tools.

This paper presents an algorithm that generates an acyclic circuit that computes the same function as a given cyclic circuit for those inputs where the cyclic circuit does not oscillate or hold state. The algorithm identifies all patterns on inputs and internal nodes that lead to acyclic evaluation orders for the cyclic circuit, which are represented as acyclic circuit fragments, then combines these to produce an acyclic circuit that can exhibit all of these behaviors.

Experimental results suggest this potentially exponential algorithm is practical for small circuits and may be improved to handle larger circuits. This algorithm should make dealing with cyclic combinational circuits nearly as easy as dealing with their acyclic counterparts.

Categories and Subject Descriptors

B.6.3 [Logic Design]: Design Aids

General Terms

Cyclic Circuits, Resynthesis, Acyclic circuits, Constructiveness

1. INTRODUCTION

The algorithm presented in this paper takes a cyclic circuit that neither oscillates nor holds state for certain inputs and builds a small acyclic circuit that computes the same function for these inputs. Cyclic circuits are minimal representations for certain functions [5, 7, 8] and can be produced by high-level synthesis tools [1, 13]. Yet many circuit analysis tools simply prohibit cyclic circuits.

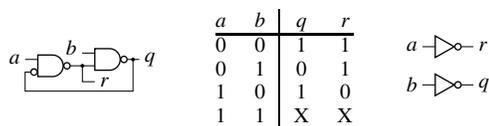
This paper uses a definition due to Malik [6]: a circuit is *combinational* for a particular input vector if three-valued simulation of the circuit starting with all internal nodes set to X resolves the

output of every gate in the circuit to either 0 or 1. For example, the output of every gate in circuit in Fig. 1a resolves to 0 or 1 when input x is 0 or when input z is 0, as suggested by the truth table in Fig. 1b. Otherwise, some of the gates output an X.

This definition is attractive because it is mathematically strong (0, 1, and X comprise a Scott domain and three-valued simulation computes the unique least fixed point of the circuit) and abstracts a physical model: Shiple et al. [9, 12] show that a circuit is stable under Malik's definition if and only if it stabilizes in a unique way under every delay assignment in Brzozowski and Seger's [4] up-bounded inertial delay model. For these reasons, Berry adopted it as the semantics for his synchronous language Esterel [2].

The two-stage algorithm in this paper builds an acyclic circuit that reproduces a cyclic circuit's combinational behavior. The algorithm first looks for a small collection of sets of assignments to inputs and internal nodes that produce all combinational behavior. The main insight (Theorem 1) is that it only necessary to consider applying controlling values to the inputs of each strongly-connected group of gates in the circuit. In the second stage, the acyclic circuit fragments implied by each set of assignments are merged using a heuristic to produce an equivalent acyclic circuit with a minimal number of gates. Identifying all patterns can be exponential in the size of the circuit, but the number of patterns is often quite small. Determining how to merge the acyclic fragments to produce the smallest circuit appears to be NP-complete, so a quadratic heuristic is used.

Below is a simple example. The contrived cyclic circuit on the left behaves combinational unless both inputs are 1, as illustrated by the truth table. The algorithm generates the acyclic circuit on the right, which reproduces only the non-oscillatory behavior.



The algorithm only reproduces the behavior of combinational input patterns and implicitly assumes that all others are not of interest. Determining whether a circuit with state-holding elements can ever reach a state with non-combinational behavior can be difficult. Shiple et al. [10, 11] use BDDs to calculate the reachable state space of a cyclic circuit containing flip-flops to determine whether any state produces non-combinational behavior. Toma [14] implemented this algorithm in the Esterel V5 compiler to check whether an apparently cyclic program could ever deadlock. If not, Toma's algorithm builds an equivalent acyclic circuit from the BDDs.

The remaining sections of this paper present how the algorithm works on the example in Fig. 1, describe the algorithm in detail, present some experimental results, and come to some conclusions.

*The author is supported by the NSF under CCR-0133348, the SRC, New York's MDC, and Intel Corporation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2003, June 2-6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0006 ...\$5.00.

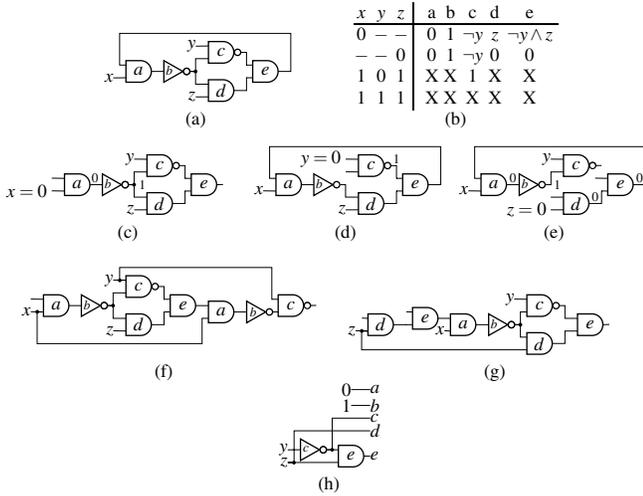


Figure 1: An example. (a) A cyclic circuit and (b) its truth table generated by three-valued simulation. The algorithm first places controlling values on inputs x , y , and z , producing two acyclic fragments (c,e), and a cyclic one (d) that does not lead to any new fragments. Fragments (c) and (e) can be merged in two ways (f,g). (g) is smaller, and can be further reduced by setting the (unlabeled) don't-care inputs to 0, producing (h). By construction, this much smaller circuit computes the same function as the circuit in (a) when the gate outputs are not X.

2. AN EXAMPLE

Fig. 1 shows how the algorithm works on a simple circuit. It first determines when (i.e., under what input conditions) the circuit is combinational, which amounts to asking which input values produce completely-defined gate outputs or equivalently, which break the strong connectivity. The following theorem is the key to answering this efficiently.

THEOREM 1. *For a circuit with a strongly-connected component (SCC) to behave combinational, at least one input to a gate in the SCC must be driven to a controlling value.*

PROOF. Assume that each external input to each gate in the strongly-connected component has a non-controlling value. By definition, this means the output of each of these gates is determined by its other inputs, which are internal to the strongly-connected component. Since there are no other sources of 0s and 1s in the circuit (constants are treated as inputs), the output of the gates in the component must simulate to all X's and therefore be non-combinational. It follows that for the circuit to behave combinational at least one input to the SCC must take a controlling value. \square

This theorem tells us combinational behavior demands we apply a controlling input to each strongly-connected component, suggesting a way to search for combinational behavior.

We start searching for combinational behavior in Fig. 1a by applying the controlling value 0 to inputs x , y , and z . Fig. 1c shows the effect of setting $x = 0$ and propagating it as far as possible. Because $x = 0$ is a controlling value, gate a ignores the output of gate e (the input on a becomes a don't-care), the output of a is 0, and the output of b is 1. The result is an acyclic circuit fragment. Setting $z = 0$ similarly produces another acyclic fragment, shown in Fig. 1e. This time, the top inputs on gates d and e become don't-cares.

In Fig. 1d, setting $y = 0$ removes the dependency from b to c , but results in a still-cyclic fragment.

Further analysis of Fig. 1d will produce no new combinational behavior. It is strongly connected and therefore subject to Theorem 1, however, the only two uncontrolled inputs to its SCC are x and z , either of which we know can produce combinational behavior by themselves. The one other input to the SCC, the output of gate c , has been set to the non-controlling value 1 by the $y = 0$ assignment, so we are not free to change it. We are finished searching for combinational behavior.

The search procedure found that together, Fig. 1c and Fig. 1e are enough to produce all combinational behavior, so the next step is to generate a circuit that behaves like both of them by fusing the two fragments. When fusing the two 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. 1f and g. Fig. 1f comes from appending Fig. 1e to the end of Fig. 1c, and Fig. 1g is Fig. 1c appended to Fig. 1e. Fig. 1g is smaller (seven gates versus eight), so we discard Fig. 1f.

The two unlabeled inputs to Fig. 1g are don't-cares because we know the other inputs on their gates will be set to controlling values when combinational input patterns are applied, so we may set them as we like. Setting both to 0 is the judicious choice, giving the very small circuit in Fig. 1h. Note that as desired, this circuit follows the truth table in Fig. 1b when no gate's output is X.

3. THE ALGORITHM

Fig. 2 shows the two-part algorithm that derives an acyclic circuit with a minimal number of duplicated gates from a cyclic one. It first finds a small set acyclic circuit fragments that produce all combinational behavior, then merges these to produce the final circuit. The search procedure is worst-case exponential, but grows slowly in practice. Furthermore, merging fragments optimally appears to be NP-complete, so a quadratic heuristic is used instead.

3.1 Finding Combinational Behavior

The first part of the algorithm uses a breadth-first search to find a "covering" for all the circuit's combinational behavior. Specifically, it finds partial assignments to inputs and internal nodes that imply combinational behavior. A *partial assignment* is a set of assignments to primary inputs or internal wires, e.g., $\{x = 0, y = 1\}$. The algorithm determines the effect of a partial assignment by propagating the information as far as it will go and severing non-controlling gate inputs. This produces a circuit fragment (such as Fig. 1c) that is combinational if and only if it is acyclic.

The algorithm considers partial assignments that control both primary inputs and internal nodes, but does not attempt to determine whether they are self-consistent. The structure of the circuit may prevent certain patterns, but this only leads to the algorithm considering more behavior than actually possible and may produce a larger, but not incorrect, final circuit.

The search minimizes the number of partial assignments it considers by only assigning controlling values to strongly-connected components, which Theorem 1 implies are sufficient, and by considering assignments in a breadth-first order, which allows it to prune the search space according to the following theorem.

THEOREM 2. *If a partial assignment p is combinational, then any further assignments that do not contradict any in p can also be computed combinational by the fragment implied by p .*

PROOF. If a partial assignment implies an acyclic circuit fragment, there is an order for evaluating gates such that enough information is known about the inputs to each gate to compute its output when the gate appears in the schedule. Additional assignments

```

{Compute the set of combinational partial assignments  $C$ }
 $C = \emptyset$  {Combinational partial assignments}
 $P = \{\{\}\}$  {A single, vacuous partial assignment.}
while  $P$  contains at least one partial assignment do
   $E = P$  {set of existing partial assignments}
   $P = \emptyset$  {set of new partial assignments}
  for each partial assignment  $p$  in  $E$  do
    if  $p$  implies a circuit fragment  $f$  with a nontrivial SCC then
      for each controlling input  $n = v$  on the first SCC in  $f$  do
        Create  $p'$  by adding  $n = v$  to  $p$ 
        if there is no  $p'' \in C$  such that  $p'' \subset p'$  then
          Add  $p'$  to  $P$ 
          if  $p'$  implies combinational behavior then
            Add  $p'$  to  $C$ 

{Assemble fragments induced by partial assignments in  $C$ }
 $s$  = schedule for first partial assignment in  $C$ 
for each other partial assignment  $p$  in  $C$  do
   $s_1$  = merge(schedule for  $p$ ,  $s$ )
   $s_2$  = merge( $s$ , schedule for  $p$ )
   $s$  = smaller of  $s_1, s_2$ 
{ $s$  is the schedule that will produce the acyclic circuit}

```

Figure 2: The algorithm. It searches for small set of partial assignments that produce all combinational behavior, then merges the evaluation orders these imply into a small schedule from which the final acyclic circuit is produced.

may add information about the value of a previously-unknown input, but once the output of a gate is established, setting the value of previously-unknown inputs cannot change it. This follows from the monotonic three-valued functions of logic gates. \square

This theorem implies there is no need to consider any partial assignment that is a superset of a known-acyclic one. So in addition to not trying to add any assignments to an already known-combinational partial assignment, the algorithm also stops when it reaches a superset of any known-combinational partial assignment. Considering partial assignments in a breadth-first order (i.e., considering all partial assignments with one assignment before any with two assignments, before any with three, etc.) exposes such superset relationships because all partial assignments with less than n assignments are tested before any with n assignments.

This is why the algorithm does not consider Fig. 1d any further. For Fig. 1, the algorithm first considers $\{x = 0\}$, $\{y = 0\}$, and $\{z = 0\}$. The first and third produce acyclic fragments, so the algorithm does not consider adding any further assignments to these two. The algorithm briefly considers the partial assignments $\{y = 0, x = 0\}$ and $\{y = 0, z = 0\}$, but both are supersets of known-combinational partial assignments so the algorithm discards them.

During the search, the algorithm only considers the first SCC in a cyclic circuit (i.e., the first one in some topological sort of the SCCs, which can be nondeterministic). While it could consider applying controlling inputs to every SCC, this would be wasteful because later SCCs are automatically considered after earlier ones become combinational. Furthermore, applying inputs to earlier SCCs can affect inputs on later SCCs to reduce the number of partial assignments that must be considered on later SCCs.

3.2 Merging Acyclic Circuit Fragments

Once the first part of the algorithm has identified a set of partial assignments that cover all combinational behavior of the cyclic circuit, the algorithm merges the acyclic circuit fragments these imply.

```

function merge( $s, s'$ )
  Clear mapping  $m$ 
  for each gate  $g'$  in  $s'$  in scheduled order do
    for each gate  $g = g'$  in  $s$  in scheduled order do
      if for all drivers  $d'$  of  $g'$ ,  $m(d')$  appears earlier than  $g$  in  $s$ 
        then
          Set  $m(g') = g$ 
          Done searching, continue with next  $g'$ 
      {We did not find a suitable match for  $g'$ }
    Append  $g'$  to  $s$ 
    Set  $m(g')$  to this newly-added gate

```

Figure 3: The schedule merging algorithm. The algorithm in Fig. 2 uses this to combine two circuits (actually, linear evaluation orders of the gates in the circuit) into a result that covers the behavior of both.

The second part of the algorithm manipulates schedules implied by the partial assignments found in the first part. A *schedule* is a linear evaluation order for all the gates in the circuit that ensures the output of each gate can be computed when it appears, i.e., when a gate is to be evaluated, either one of its inputs is known to take a controlling value or all its inputs are known to be non-controlling. Any partial assignment that implies combinational behavior has at least one schedule. For example, the circuit fragment in Fig. 1e has the schedule *deabc*.

The second part of the algorithm uses a greedy technique to find a short schedule that covers all the schedules implied by the combinational partial assignments found in the first part. It builds a final schedule s by trying to merge the schedule for each partial assignment first before and then after s . To keep the final circuit small, the shorter of these two schedules becomes the new s .

Fig. 3 is 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 schedule s' without introducing a cycle. For each gate in s' , it finds the earliest identical one in s whose use would not create a cycle, and otherwise adds a copy of the gate to the end. The merging algorithm is not optimal because in general, a correct merge could add a new copy of a gate in many places in the schedule, not just at the end, but the best place is not obvious.

The mapping m records the most-recently-used copy of each gate g to avoid introducing a cycle. The *if* test guarantees that every input wire to the gate g' comes from gates earlier in the schedule. Thus, the schedule s always implies an acyclic netlist since a gate's inputs always come from gates earlier in the schedule.

3.3 Generating a Circuit from a Schedule

Constructing an acyclic circuit from a schedule is a mechanical procedure that steps through the gates in the final schedule s in order, adding a copy of each corresponding gate in the cyclic circuit to the acyclic circuit under construction. The inputs of this new gate are connected to the most-recent-added copies of the corresponding gates in the acyclic circuit. So if a gate in the cyclic circuit is driven by the outputs of gates a and b , then its copy in the acyclic circuit is driven by the most recent copies of a and b .

Consider how this algorithm generates Fig. 1g from the schedule *deabcde*. It first copies gate d and connects its bottom input to the primary input z . Its top input comes from gate b , which has not yet been copied, so this input stays a don't-care. Gates e , a , b , and c are copied next, each driven by the last. The second time gate d is copied, there is a copy of gate b to which to connect its top input. Finally, when gate e is copied, there is a copy of c to drive its top input and its bottom input is driven by the most recent copy of d .

Table 1: For some circuits, the number of gates in the cyclic circuit, in the SCC, and the generated acyclic circuit; the number of partial assignments tested; the number of acyclic partial assignments found; and the time taken.

Name	Gates	SCC Size	Acyclic Gates	Total PAs	Acyclic PAs	Time
cy8	16	3	20	3	2	0.5s
cy7	21	4	28	3	2	0.5s
cy1	99	6	187	7	6	0.7s
arb2	94	10	138	15	6	0.7s
arb4	176	20	245	93	12	1.2s
arb6	246	30	340	745	16	6.0s
arb7	281	35	389	2205	18	51s

4. EXPERIMENTAL RESULTS

I implemented a prototype of the algorithm to test its behavior on some examples generated from Esterel [3]. Although I give execution times, the speed of the implementation could be improved.

Table 1 shows the results for a few small circuits generated from Esterel programs. cy1, 7, and 8 are all small circuits. arb2 through 7 are cyclic arbiters that contain an SCC that looks like . The gates at the tips of the diamonds are the challenge for the algorithm because it tries all combinations of tips on different diamonds, even after finding that two tips on the same diamond is sufficient. Esterel's hardware synthesis procedure [1] generates this odd structure, which is effectively a simple loop: the inputs to the gates at the tips of each diamond always take the same value.

The algorithm currently does not attempt to further reduce circuit size by applying constants to don't-care inputs, which could substantially reduce the size of the circuit. For example, the algorithm currently returns Fig. 1g instead of Fig. 1h.

Although the exponentially-growing number of partial assignments the algorithm considers is disturbing, the fairly small number of acyclic partial assignments it produces is encouraging since it suggests many acyclic circuits have small acyclic equivalents.

5. CONCLUSIONS AND FUTURE WORK

This paper presented an algorithm for constructing an acyclic circuit that computes the same function as a cyclic one for inputs where the cyclic circuit behaves combinational, i.e., when it cannot exhibit oscillatory or state-holding behavior. It finds a set of partial assignments to both primary inputs and internal nodes that imply combinational behavior, then builds an acyclic circuit that contains all this behavior. The search space is pruned by stopping at subsets of already-identified behavior and by only applying controlling values to inputs of gates in strongly-connected components, which a theorem shows is sufficient.

Preliminary experimental results suggest that while it appears the number of partial assignments considered can grow exponentially with the size of a strongly-connected component, the number of acyclic partial assignments grows much more slowly, suggesting it is fairly easy to produce small acyclic equivalents. Additional algorithmic and implementation improvements are also possible.

This paper does not establish whether a circuit always behaves combinational (due, perhaps, to states it cannot reach or environmental constraints), but the algorithm could be applied to aid this test. The algorithm derives necessary conditions for the circuit to be cyclic: one of the acyclic partial assignments must hold for the circuit to be combinational. This could be treated as an invariant that must hold for the circuit to be combinational. Showing that the

circuit starts in a combinational state and that that combinational states only transition to other states would suffice, eliminating the need to compute the exact set of reachable states. But such an invariant may be too weak.

The algorithm implicitly assumes that there always exists some input pattern that can independently put any controlling value on any input to each SCC, which is not true in general because of the structure of the circuit. While this does not affect the algorithm's correctness, it may cause it to search longer and produce larger circuits. An obvious next step is to add SAT/ATPG-like reasoning to the algorithm to avoid this case.

Overall, this appears to be a promising approach to handling well-behaved acyclic circuits. Further algorithm improvements and a more careful implementation promise to make it more practical.

Acknowledgements

Avraham Shinnar provided the initial inspiration for this paper; it was his idea to cover schedules with circuits. Sharad Malik, Mike Kishinevsky, and Loïc Henry-Gerard also contributed significantly. Years of discussion with Gérard Berry and Tom Shiple also helped.

REFERENCES

- [1] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–103, April 1992. Issue 1652, Mechanized Reasoning and Hardware Design.
- [2] Gérard Berry. The constructive semantics of pure Esterel. Draft book, 1999.
- [3] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
- [4] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.
- [5] W. H. Kautz. The necessity of closed loops in minimal combinational circuits. *IEEE Transactions on Computers*, C-19(2):162–164, February 1970.
- [6] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.
- [7] Marc D. Riedel and Jeoshua Bruck. The synthesis of cyclic combinational circuits. In *Proceedings of the 40th Design Automation Conference*, Anaheim, California, jun 2003.
- [8] Ronald L. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers*, 26(6):606–607, 1977.
- [9] Thomas R. Shiple, Gérard Berry, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Logical analysis of combinational cycles. Technical Report UCB/ERL M02/21, University of California, Berkeley, June 2002.
- [10] Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference*, pages 328–333, Paris, France, March 1996.
- [11] Thomas R. Shiple, Vigyan Singhal, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Analysis of combinational cycles in sequential circuits. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, volume IV, pages 592–595, May 1996.
- [12] Thomas Robert Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California, Berkeley, October 1996. Memorandum UCB/ERL M96/76.
- [13] Leon Stok. False loops through resource sharing. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 345–348, San Jose, California, November 1992.
- [14] Horia Toma. *Analyse constructive et optimisation séquentielle des circuits générés à partir du langage synchrone réactif Esterel [Constructive Analysis and Sequential Optimization of Circuits Generated from the Synchronous Reactive Language Esterel]*. PhD thesis, École des Mines de Paris, 1997.