The Analysis of Cyclic Circuits with Boolean Satisfiability

John Backes, Brian Fett, and Marc D. Riedel

Department of Electrical and Computer Engineering
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455
{back0145, fett, mriedel}@umn.edu

Abstract—The accepted wisdom is that combinational circuits must have acyclic (i.e., loop-free or feed-forward) topologies. And yet simple examples suggest that this need not be so. In previous work, we advocated the design of cyclic combinational circuits (i.e., circuits with loops or feedback paths). We proposed a synthesis methodology and demonstrated that it produces significant improvements in area and in delay. The analysis method that we used to validate cyclic circuits was based on binary decision diagrams. In this paper, we propose a much more efficient technique for analysis based on Boolean satisfiability (SAT).

I. INTRODUCTION

A. Cyclic Combinational Circuits

A collection of logic gates forms a combinational circuit if the outputs can be described as Boolean functions of the current input values only. A common misconception is that combinational circuits must have acyclic topologies; that is to say, they must be designed without any loops or feedback paths. In fact, the idea that “combinational” and “acyclic” are synonymous terms is so thoroughly ingrained that many textbooks provide the latter as a definition of the former (e.g., 16, p. 14; 25, p. 193).

Indeed, any acyclic circuit is clearly combinational. Regardless of the initial values on the wires, once the values of the inputs are fixed, the signals propagate to the outputs. The behavior of a circuit with feedback is generally more complicated. Such a circuit may exhibit sequential behavior, as in the case of an S-R latch, or it may be unstable, as in the case of an oscillator.

And yet, circuits with cyclic topologies can be combinational. Consider the example in Figure 1. It is combinational in the strictest sense: it produces the required output values regardless of the prior values on the wires and for any choice of delay parameters. If \( x = 0 \) then \( g_1 \) produces an output of 0, because 0 is a controlling value for an AND gate. If \( x = 1 \) then \( g_1 \) produces a value of 1, because 1 is a controlling value for an OR gate. In both cases, the cycle is broken and the circuit produces definite outputs. Since \( x \) must assume one of these two values, we conclude that the circuit always produces definite outputs. In fact, it implements two functions that both depend on all five variables:

\[
\begin{align*}
  f_1 &= b(a + x(d + c)), \\
  f_2 &= d + c(x + ba)
\end{align*}
\]

(+ denotes OR, (.) denotes AND)

Note that the computation of the two functions overlaps. If we were to implement these functions with an acyclic circuit, we would need eight two-input gates.

B. Analyzing Cyclic Circuits

In previous work, we showed that combinational circuits can be optimized significantly if cycles are introduced [15], [17]. A pivotal step in the synthesis methodology is determining whether cyclic circuits that are found behave combinationally.
This analysis problem is conceptually straight-forward: correctness is ascertained by following all controlling values as they propagate through the circuit from the primary inputs — zeros controlling the outputs of AND gates, ones controlling the outputs of OR gates, these values controlling other gates, and so on. Of course, stepping through all possible input assignments is not a tractable proposition for real circuits: given \( n \) primary inputs, there would be \( 2^n \) input assignments to consider.

This is a specific problem but one that shares many properties with a broad class of problems in logic verification: it has an affirmative answer if a property holds for all possible input assignments; it has a negative answer if the property does not hold for any input assignment. The property — in this case, whether the circuit produces combinational behavior or not — is one directly ascribed to logical operations on the circuit — in this case, how controlling values propagate.

So-called SAT-based techniques, based on heuristic solutions to the Boolean satisfiability problem, have been deployed very successfully for problems in this vein \([8],[12]\). In theory, such algorithms can take time that is exponential in the number of variables to complete. In practice, they have shown themselves to be remarkably efficient for problems in circuit verification — often handling problem instances with thousands of variables with ease.

Given a cyclic circuit, how can the analysis for combinational behavior proceed? We adopt the following strategy:

- We find a feedback arc set, that is to say, wires that we can cut to make the circuit acyclic.
- We introduce new dummy variables at those cut locations.
- We encode the entire computation of the circuit in terms of ternary-valued logic: zeros, ones and “undefined” values. These ternary values are encoded with “dual-rail” binary values: zero is encoded as \([0,0]\), one as \([1,1]\), and “undefined” as either \([1,0]\) or \([0,1]\).
- We set up an acyclic circuit that answers the question: given undefined values for the dummy variables (in the ternary encoding) is there any input assignment that produces undefined values (again in the ternary encoding) at the output? This circuit forms the SAT question.

The algorithm is described in detail in Section II. The complexity is entirely dependent on the runtime of the SAT solver. Setting up the circuit for the SAT instance is comparatively trivial: it entails but a single pass through the circuit to compute a feedback arc set. The circuit for the SAT question is larger than the original circuit: for every gate in the original circuit, approximately six gates are needed to formulate the ternary-valued encoding; in addition to the primary inputs, the dummy variables at the cut locations are included. Given the efficiency of SAT solvers, this is a winning strategy in spite of the increase in the number of variables. In Section IV, we compare runtimes on benchmark circuits for this method and the previous BDD-based method.

C. Prior and Related Work

In an earlier era, theoreticians commented on the possibility of having cycles in combinational logic and conjectured that this might be a useful property \([5],[7],[21]\). Both McCaw and Rivest presented examples of cyclic circuits with provably fewer gates than is possible with equivalent acyclic circuits \([10],[18]\). (We have extended and generalized these theoretical results. Most notably, we have constructed a family of circuits with cyclic topologies having half as many gates as is possible with acyclic topologies \([17]\).)

In a later era, practitioners observed that cycles sometimes appear in combinational circuits synthesized from high-level descriptions. Stok noted that cycles can be introduced during resource-sharing optimizations at the level of functional units \([22]\). However, since synthesis and verification tools balk when given combinational logic with cycles, he concluded that those optimizations have to be rejected at the high-level phase.

Motivated by Stok’s observation, Malik discussed analysis techniques for cyclic circuits \([9]\). He formulated a symbolic simulation algorithm based on ternary-valued simulation. He proposed a topological approach, beginning with a transformation from a cyclic specification to an equivalent acyclic one. Edwards followed a similar strategy, discussing techniques specifically targeted at cyclic circuits that are produced inadvertently during high-level design \([3]\). Shipile refined and formalized Malik’s results and extended the concepts to combinational logic embedded in sequential circuits \([20]\).

In previous work, we described a methodology for synthesizing cyclic circuits \([15]\). Our approach for synthesis isconceptually general. Cycles are introduced through the incremental application of restructuring and minimization operations, optimizing a design for area and delay. These optimizations are carried through to the decomposition and technology mapping phases. The methodology is implemented as a package called CYCLIFY, built within the Berkeley SIS environment \([19]\). Trials on benchmark circuits as well as examples from industry demonstrated that cyclic solutions are not a rarity; they can readily be found for most circuits of practical interest. CYCLIFY reduced the area of standard benchmark circuits by as much as 30% and the delay by as much as 25%. For analysis, we discussed techniques for validating cyclic circuits based on symbolic event propagation with binary decision diagrams (BDDs) \([14]\). We also discussed techniques for performing timing analysis of cyclic circuits \([16]\).

D. Circuit Model

The concepts discussed in this paper are not tied to any particular physical model or computing substrate. Generally the exposition is at a symbolic level, that is to say, in terms of Boolean expressions. However, we first discuss the circuit model in an explicit sense — in terms of signal values.

We work with the digital abstraction of zeros and ones. Nevertheless, our model recognizes that the underlying signals are, in fact, analog: each signal is a continuous real-valued function of time, corresponding to a voltage level. For analysis, we adopt a ternary framework, extending the set of Boolean values \( \mathbb{B} = \{0,1\} \) to the set of ternary values \( \mathbb{T} = \{0,1,\bot\} \). Here \( \bot \) represents either an ambiguous value, e.g., a voltage value between logical 0 and logical 1, or else an uncertain value, i.e., a signal that might be 0 or 1 — but we do not know which.
The idea of three-valued logic for circuit analysis is well established. It was originally proposed for the analysis of hazards in combinational logic [26]. Bryant popularized its use for verification [1], and it has been widely adopted for the analysis of asynchronous circuits [2]. For a theoretical treatment, see [13]. Malik and Shiple discuss the analysis of cyclic circuits in this framework [9], [20].

Central to the analysis is the concept of controlling values. In [17], a formalism is presented for computing the controlling values of arbitrary logic functions, in a symbolic context. For simplicity, in this paper we assume that the network has been decomposed into primitive gates, namely AND/OR/NAND/NOR gates and inverters. Recall that 0 is the controlling value for an AND gate, as shown in Figure 2. Similarly, 1 is the controlling value for an OR gate.

Our analysis characterizes the functional and temporal behavior of circuits according to the so-called “floating-mode” assumption [2], [4]: at the outset, all wires in a circuit are assumed to have unknown or possibly undefined values, and so are assigned the value ⊥. This assumption ensures that the analysis does not infer stability in cases where ambiguous or unstable signals might persist.

Consider the circuit fragment in Figure 3. One might be tempted to reason as follows: the output of the AND gate $g_1$ is fed in complemented and uncomplemented form into the OR gate $g_2$. Thus, one of the inputs to the OR gate must be 1, and so its output must be 1.

And yet, by definition, ⊥ designates an undefined value. For instance, it could indicate a voltage value exactly halfway between logical 0 and logical 1. Within the floating-mode framework, we remain agnostic: the output of the OR gate is ⊥.

We define the validity of a cyclic circuit as follows:

- If, for some assignment to the primary inputs, there are ⊥ values in the fixed point that the circuit settles at, then the circuit is “Invalid.”
- Conversely, if for every assignment to the primary inputs there are no ⊥ values in the fixed point that it settles at, then the circuit is “Valid.”

Of course, if there are “don’t-care” conditions, then validity only applies to assignments in the “care” set. We could adopt a less stringent definition, only insisting that no ⊥ values persist at the primary outputs; this would not alter our algorithm materially, so here we use the more stringent definition that no ⊥ values can persist on any of the wires in the circuit, whether these be internal or at the primary outputs.

II. ALGORITHM

Given a cyclic circuit, the objective of the analysis is to produce an acyclic circuit that computes an output value that is identically zero if and only if the cyclic circuit is valid. This acyclic circuit will then be fed into a SAT solver; we will refer to it as the “SAT circuit”.

1) The first step is to find wires that, if cut from the circuit, would break all the cycles. Such a set can be found through a simple depth-first search [24]. (Finding the smallest set of wires is, of course, difficult: this is the minimum feedback arc set problem, known to be NP-hard. The fewer wires in the cut set, the fewer dummy variables we introduce and hence the smaller the size of the SAT instance; however, given the efficiency of SAT solvers, spending time finding the very best cut sets may not be expeditious; the construction will work for any cut set.)

2) The next step is to convert every gate in the circuit into a corresponding module that operates on the dual-rail encoded ternary logic. Using the encoding scheme given in Figure 4, this step is straightforward. Consider the encoding for an AND operation on ternary-valued inputs $a$ and $b$. We use pairs of inputs for each value: $a_0$ and $a_1$ corresponding to $a$, and $b_0$ and $b_1$ corresponding to $b$. The outputs are encoded by the functions:

$$f_0 = a_0b_0 + a_1b_1$$
$$f_1 = a_1b_1 + a_0b_0$$
Other gates, such as OR, NAND, NOR, etc., can be implemented similarly. The NOT operation is particularly easy—we simply complement the bit on each rail.

3) Each primary input is simply considered twice to obtain its dual-rail encoding. This way, if the primary input is assigned logic 1, the value (11) is fed; if it is assigned logic 0 the value (00) is fed.

4) At every cut location, we introduce a pair of dummy variables feeding into the corresponding dual-rail module. This allows for the possibility that the value in the circuit is \( \bot \), encoded as different values assigned to each of the dummies, (01) or (10).

5) For every pair of dummy variables, we set up an equivalence checker: this is a module that evaluates to 1 if and only if the value assigned to dummies agrees with the value computed by the circuit at the cut location. The circuit may be computing \( \bot \), encoded as (01) or (10); in this case, the equivalence checker evaluates to 1 if the dummies have different values. Call the output of the equivalence checker \( x_i \) for each cut location \( i \). For dummy variables \( d_1 \) and \( d_2 \) and gate outputs \( f_1 \) and \( f_2 \), the logic for the equivalence checker is

\[
x_i = \overline{d_1}d_2f_1f_2 + d_1d_2f_1\overline{f_2} + \\
\overline{d_1}d_2\overline{f_1}f_2 + d_1d_2\overline{f_1}\overline{f_2} + \\
d_1\overline{d_2}f_1\overline{f_2} + d_1d_2f_1\overline{f_2},
\]

6) For every pair of dummy variables, we set up a \( \bot \)-checker: this is simply an exclusive-OR gate on the two dummies; it evaluates to 1 if and only if the dummies are assigned different values. Call the output of the \( \bot \)-checker \( y_i \) for each cut location \( i \).

7) Finally, as illustrated in Figure 5, the output of the circuit is the AND of the AND of the \( x_i \)'s and the OR of the \( y_i \)'s.

**Example 1**

Consider the circuit in Figure 6, consisting of four NAND gates. Note that there are two cycles. Cutting these and inserting dummy variables \( d \) and \( e \), we obtain the circuit in Figure 7. Next, we replace each gate with a dual-rail version; we feed in pairs of dummy variables, \( d_0, d_1, e_0, \) and \( e_1 \), corresponding to each of the previous dummy variables; we double up the primary inputs \( a \) and \( b \); we add two equivalence-checkers, producing \( x_0 \) and \( x_1 \); we add two \( \bot \)-checkers (i.e., exclusive-OR gates) producing \( y_0 \) and \( y_1 \); and we add three logic gates \( g_1, g_2, \) and \( g_3 \) to form the final output.

This circuit, shown in Figure 8, forms the SAT instance with six variables: \( a, b, d_0, d_1, e_0, \) and \( e_1 \). We see that for \( a = b = 1, \) \( d_0 = d_1, \) and \( e_0 = e_1, \) we get \( \bot \) values on each pair of rails into the equivalence checkers, indicating that the inputs to each are equivalent; so \( x_0 \) and \( x_1 \) produce outputs of 1; \( y_0 \) and \( y_1 \) produce outputs of 1 as well; so the final output is 1.

Therefore, the SAT instance is satisfiable and the circuit is invalid. Indeed, \( a = b = 1 \) are non-controlling values for the NAND gates, so this is the outcome that we expect.

**III. PROOF OF CORRECTNESS**

First, we argue that a SAT circuit that evaluates to 1 never corresponds to a valid cyclic circuit. Indeed, if a SAT circuit evaluates to 1, then both the gates \( g_1 \) and \( g_2 \) are at 1. If \( g_1 \) is at 1, then the corresponding values in the cyclic circuit are at a fixed point; however, if \( g_2 \) is at 1, then some of the values in the fixed point are \( \bot. \) By definition, the cyclic circuit is invalid.

Next we argue that every invalid cyclic circuit translates into a SAT circuit that evaluates to 1 for a specific input assignment. Indeed, if the circuit is invalid then it has a fixed point with \( \bot \) values on some of the wires of the cut set. (A fixed point that contains \( \bot \) values somewhere must also have these on the cut set.) In the SAT circuit, consider such an input assignment; assign the dummy values that correspond to the values from the fixed point; this ensures that \( g_1 \) is at 1. Because some of these values are \( \bot, \) \( g_2 \) is also at 1 and so the SAT circuit evaluates to 1.
Fig. 8. The SAT circuit corresponding to the cyclic circuit in Figure 6.

IV. IMPLEMENTATION AND RESULTS

We have implemented the algorithm described in Section II in the Berkeley ABC environment [11]. ABC invokes the "MiniSat" SAT Solver [23]. We performed trials on cyclic circuits produced by our tool, CYCLIFY, from benchmark circuits in the IWLS collection [27]. (For circuits with latches, we extracted the combinational part.) All circuits had been mapped to 2-input NAND and NOR gates and inverters. We count the area of the NAND/NOR gates as 2, and that of inverters as 1. We compare the runtimes for the new SAT-based method to those using our previous BDD-based method [16]. Trials were performed on a 2.93 GHz Intel Core 2 Duo Processor running Linux.

V. DISCUSSION

Early work in the 1960's and 70's established the premise of combinational circuits with cycles, and suggested the possible benefits. Still, combinational circuits are not designed with cycles in practice. Perhaps designers have eschewed feedback due to the apparent complexity of reasoning about cyclic structures. And yet, feedback provides significant opportunities for optimization, both for area and for delay. Indeed, contrary to the conventional wisdom, cyclic solutions are not a rarity; they can readily be found for most circuits that are not trivially simple or sparse. We have run trials with our program, called CYCLIFY, on a range of randomly generated examples and benchmark circuits. We note that solutions for most of the examples have deeply nested loops, with dozens or even hundreds of cycles.

Our synthesis strategy is to introduce feedback in the restructuring and minimization phases. A branch-and-bound search is performed, with analysis used to validate and rank potential solutions. Using BDDs, the analysis portion completely dominated the running time of the synthesis flow. The SAT-based methodology proposed here can tackle much larger benchmark circuits and it runs orders of magnitude faster (as anyone familiar with SAT-based methods might have expected). For all of the circuits that we have tried, the time that it took to convert the original circuit description into the SAT instance took less than .01 seconds. Future work will include developing specific heuristics for finding smaller feedback arc sets; improvements here might positively impact
the runtime of the algorithm. We are working on integrating the SAT-based analysis method with synthesis; we will report the results in the near future.

REFERENCES