The Analysis and Mapping 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 prior work, we advocated the design of *cyclic* combinational circuits (i.e., circuits with loops or feedback paths). We proposed a methodology for synthesizing cyclic circuits and demonstrated that it produces significant improvements in area. Efficient synthesis is predicated on efficient analysis. In prior work, we used binary decision diagram (BDD)-based algorithms to validate cyclic circuits. In this paper, we propose much more efficient techniques based on Boolean satisfiability (SAT). Validation is performed both at a network level, in terms of functional dependencies, as well as at a gate level, after mapping to a library. When mapping breaks the validity of a combinational circuit, SAT-based analysis returns satisfying assignments; we use these assignments to modify the mapping in order to ensure that the circuit remains combinational. We demonstrate the effectiveness of the analysis and mapping algorithms on standard benchmarks.**

***Index Terms*—cyclic circuits, Boolean satisfiability, logic synthesis, technology mapping**

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; [37], 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 circuit 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_4 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:

$$\begin{aligned} f_1 &= b(a + x(d + c)), \\ f_2 &= d + c(x + b\,a) \end{aligned}$$

$(+)$ denotes OR, $(\cdot)$ 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 [27], [28]. A pivotal step in the synthesis methodology is determining whether cyclic circuits that are found are indeed valid, that is to say, they are combinational and implement the requisite Boolean functions. 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-world 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

This research has been funded in part by a grant from the SRC Focus Center Research Program on Functional Engineered Nano-Architectonics (FENA), contract No. 2003-NT-1107 and NSF CAREER Award #0845650.

A preliminary version of this paper appeared in [2]. This manuscript contains two new sections, Sections IV and V, describing a method for applying the results of analysis to fix broken mappings. The introduction has been expanded with a motivating example on mapping. Also, new experimental results were added.

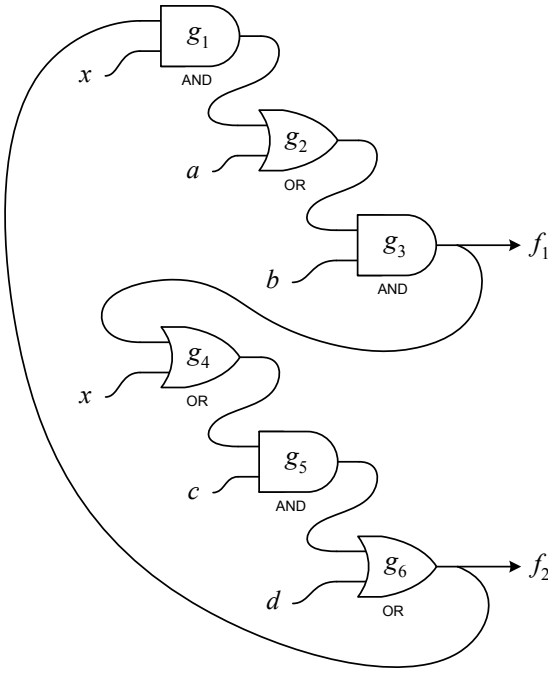


Fig. 1. A cyclic combinational circuit.

hold for *some* 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 [1], [18]. Consider the classic problem in circuit verification: determining whether two circuits A and B are equivalent in the sense that they implement the same Boolean function. To solve this problem, one creates a new circuit C with the outputs of A and B tied together by an exclusive-OR gate. Then one asks the SAT question: is there some assignment of input values that *satisfies* the Boolean function implemented by C (i.e., for which the output of C evaluates to one)? If not, then the two circuits are equivalent. The starting point for SAT-based verification, then, is a circuit that returns identically zero (*UNSAT*) for an affirmative answer to the problem; and not identically zero (*SAT*) for a negative answer. The analysis proceeds by packaging the Boolean function implemented by the circuit as a formula in Conjunctive Normal Form (CNF). This is passed to heuristic algorithms known as SAT-Solvers [10], [24]. 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, handling problem instances containing thousands of variables with ease.

The main contribution of this paper is a SAT-based methodology for verifying whether cyclic circuits are combinational. The question of whether the circuit is combinational is packaged as a CNF formula through a ternary-valued decomposition of the circuit. The algorithm is described in detail in Section II. In rough outline, the steps are:

- 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 these 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.

In the case where the circuit in question is indeed combinational, the SAT solver returns an answer of *UNSAT*. If some assignment of the circuit's primary inputs result in non-combinational behavior, the solver returns an answer of *SAT* and it also provides a satisfying assignment. As we discuss in the next section, we can make use of this satisfying assignment. The flow of the analysis algorithm is illustrated in Figure 2.

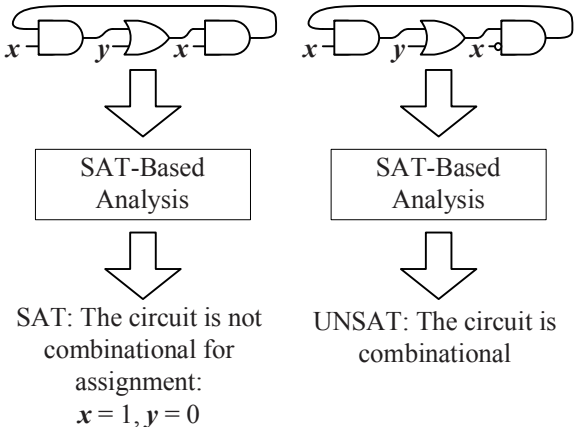


Fig. 2. An illustration of how SAT-based analysis works. If the circuit is combinational, the SAT solver returns an answer of *UNSAT*. If the circuit is not combinational, it returns an answer *SAT* and it provides a satisfying assignment.

The complexity of the analysis is 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. Given the efficiency of SAT solvers, this is a winning strategy in spite of the increase in the circuit's size. In Section VI, we compare runtimes on benchmark circuits for this method compared to binary decision diagram (BDD)-based methods.

C. Mapping Cyclic Circuits

In our synthesis flow, we introduce cycles at the level of functional dependencies in a Boolean network [28], [3]. These designs are then mapped to gates from a library. Cyclic designs must be validated both at the level of functional

dependencies and then again after mapping. This is necessary because mapping sometimes breaks the validity: designs that are combinational at the functional level get mapped onto designs that are not combinational at the gate level. This was first observed in [15].

Consider the functions in Figure 3. The three functions form a cycle: f depends on h, h depends on g, and g depends on f. The reader can verify that for all assignments of the primary inputs a and b, the functions f, g, and h evaluate to definite Boolean values, so we consider this specification to be combinational. Figure 4 shows gate-level mappings for the three functions. Since the functional-level specification is combinational, one might assume that one can simply wire these gate-level mappings together, as shown in Figure 5. But this doesn't work: trying input combinations, we see that the assignment $a = b = 1$ does not result in definite values for the outputs f, g, and h. The individual gate mappings for the functions are correct, but the resulting circuit is not combinational.

The problem arises with the mapping for f. At the functional level, input values of $a = b = 1$ result in $f = (h)(\bar{h}) = 0$. However, at the gate level, the initial values on internal wires are not only unknown but possibly undefined. (These could have voltage values that are not unequivocally 0 or 1 but possibly some value in between.) Here the value of h is undefined, so the value of f is undefined. As we explain in Section I-E, the validity of a circuit can be established with ternary-valued simulation.

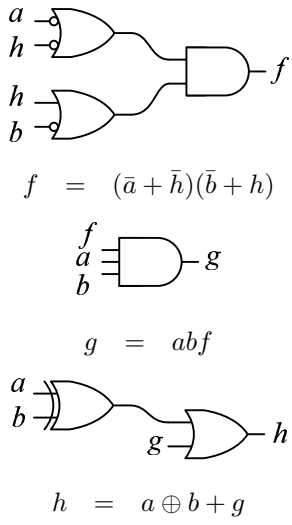
This paper presents a technique for modifying the mapping of cyclic circuits to ensure that they are combinational, based on the results of SAT analysis. The circuit in Figure 5 can be fixed by adding additional logic, as shown in Figure 6. This additional logic can be generated from a set of input assignments that results in non-combinational behavior. Our SAT-based analysis provides exactly such satisfying assignments. For the circuit in Figure 5, SAT-based analysis returns the satisfying assignment $a = b = 1$. This assignment is used to generate the additional logic in Figure 6. The reader can verify that the circuit in this figure is combinational.

$$\begin{aligned} f &= (\bar{a} + \bar{h})(\bar{b} + h) \\ g &= abf \\ h &= a \oplus b + g \end{aligned}$$

Fig. 3. A cyclic specification of three Boolean functions, f, g and h. These evaluate to definite Boolean values for all assignments of the inputs a and b.

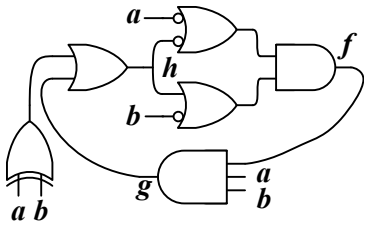
D. Prior and Related Work

In an earlier era, theoreticians commented on the possibility of having cycles in combinational logic and conjectured that this might be a useful property [14], [17], [33]. Both McCaw and Rivest presented examples of cyclic circuits with provably fewer gates than is possible with equivalent acyclic circuits [21], [30]. We have extended and generalized these theoretical results. Most notably, we have constructed a family



$$\begin{aligned} f &= (\bar{a} + \bar{h})(\bar{b} + h) \\ g &= abf \\ h &= a \oplus b + g \end{aligned}$$

Fig. 4. Individual gate mappings for the functions in Figure 3.

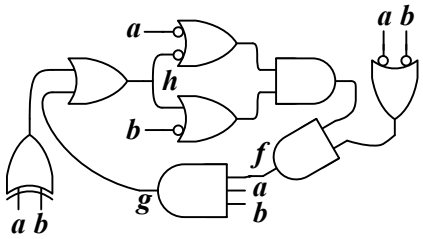


$$\begin{aligned} f &= (\bar{a} + \bar{h})(\bar{b} + h) \\ g &= abf \\ h &= a \oplus b + g \end{aligned}$$

Fig. 5. The circuit obtained by assembling the mappings in Figure 4 together. It is *not* combinational.

of circuits with cyclic topologies having half as many gates as is possible with acyclic topologies [27].

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 [35]. However, since synthesis and verification tools balk when given combinational logic with cycles, he concluded that those optimizations must be rejected at the high-level phase.



$$\begin{aligned} f &= (\bar{a} + \bar{h})(\bar{b} + h) \\ g &= abf \\ h &= a \oplus b + g \end{aligned}$$

Fig. 6. The circuit in Figure 5 with additional logic. It is combinational.

Motivated by Stok's observation, Malik discussed analysis techniques for cyclic circuits [20]. He formulated a symbolic analysis algorithm based on ternary-valued simulation. His approach was topological, beginning with a transformation from a cyclic specification to an equivalent acyclic one. Later Shiple refined and formalized Malik's results and extended the concepts to combinational logic embedded in sequential circuits [32].

More recently, Neiroukh and Edwards discussed analysis strategies targeting cyclic circuits that are produced inadvertently during design [9], [25]. Following a strategy similar to Malik's, they proposed techniques for transforming valid cyclic circuits into functionally equivalent acyclic circuits [25]. Their algorithm enumerates partial Boolean assignments that break the feedback paths in cyclic circuits. The enumeration continues until enough assignments are found to cover the entire input space. Based on these partial assignments, acyclic fragments are assembled into a new acyclic circuit. As a starting point, they presume that the given circuit is combinational and correctly mapped. The enumeration is explicit and so the algorithm is potentially very slow, as it searches through an exponentially large space of partial assignments.

We were the first to suggest a strategy for synthesizing cyclic circuits [28]. In our synthesis method, cycles are introduced through the incremental application of restructuring and minimization operations, optimizing circuits for area and for delay. We implemented the method in a package called CYCLIFY, built within the Berkeley SIS environment [31]. Trials on benchmark circuits as well as examples from industry demonstrated that cyclic solutions are not a rarity; they can readily be found for many circuits of practical interest. CYCLIFY reduced the area of these as 30% and the delay by as much as 25%. However, the analysis in CYCLIFY was binary decision diagram based. This limited the size of the circuits that could be optimized effectively.

This paper aims to bring the analysis and synthesis of cyclic circuits into the modern era by exploiting the efficiency of SAT solving. Complementary to the techniques that we present here, we have been developing a SAT-based method for synthesizing cyclic dependencies through *Craig interpolation* [3].

E. Circuit Model

A Boolean literal is a Boolean variable that is either negated or not negated. An overbar ($\bar{x}$) is used to indicate negation. A plus symbol (+) is used to indicate an OR operation (conjunction) and multiplication is used to indicate an AND operation (disjunction). A product is a disjunction of literals and a sum is a conjunction of literals. A Boolean formula in sum-of-products (SOP) form is a conjunction of products. A Boolean formula in product-of-sums (POS) form is a disjunction of sums. The support set of a Boolean formula is the set of variables present in that formula.

A **partial assignment** of a function's support variables is a valuation of that function over a subset of its support variables. In a partial assignment, unassigned variables are assumed to have value $\perp$, defined below. A product is said to **cover** a partial assignment if that product evaluates to 1 for that partial assignment. Similarly, a sum is said to cover a partial assignment if it evaluates to 0 for that partial assignment. A **prime implicant** is a product of minimal size (in terms of number of the literals) that covers a set of partial assignments. A **prime implicate** is a sum of minimal size (in terms number of the literals) that covers a set of partial assignments.

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 binary values $\mathbb{B} = \{0, 1\}$ to the set of ternary values $\mathbb{T} = \{0, 1, \perp\}$. Here $\perp$ represents either an undefined 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. We say that a variable's value is **definite** if it is 0 or 1 and that it is **indefinite** if it $\perp$.

The idea of three-valued logic for circuit analysis is well established. It was originally proposed for the analysis of *hazards* in combinational logic [38]. Bryant popularized its use for verification [7], and it has been widely adopted for the analysis of asynchronous circuits [8]. For a theoretical treatment, see [22]. Malik and Shiple discuss the analysis of cyclic circuits in this framework [20], [32].

Central to the analysis is the concept of **controlling** values. In [27], 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 7. Similarly, 1 is the controlling value for an OR gate.

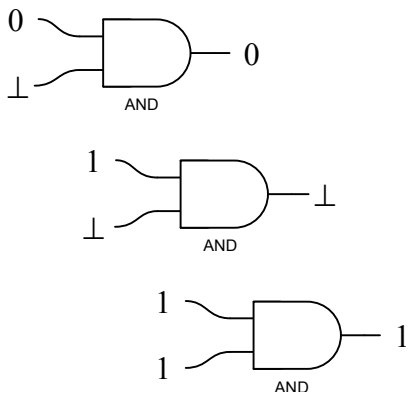


Fig. 7. An AND gate with 0, 1, and $\perp$ inputs.

Our analysis characterizes the functional and temporal behavior of circuits according to the so-called "floating-mode" assumption [8], [11]: at the outset, all wires in a circuit are assumed to have undefined values, and so are assigned the value $\perp$. 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 8. 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, $\perp$ designates an undefined value, perhaps a voltage value exactly half way between logical 0 and logical 1. Within the floating-mode framework, we remain agnostic: the output of the OR gate is $\perp$.

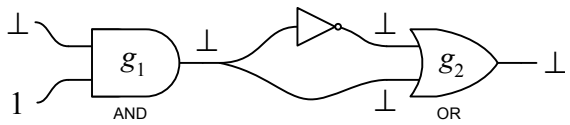


Fig. 8. An illustration of the floating mode.

Conceptually, the analysis that we perform for cyclic circuits is just an algorithmic implementation of the idea illustrated in the opening section of the paper. All the wires initially have value $\perp$. We apply definite values to the inputs, and track the propagation of well-defined signal values. Once a definite value is assigned to an internal wire, this value persists (so long as the input values are held constant). For any input assignment, a circuit reaches a so-called **fixed point** in the ternary framework: a state where no further updates of controlling values are possible. This fixed point is unique [8].

We define the validity of a cyclic circuit as follows:

- If, for some assignment to the primary inputs, there are $\perp$ values in the fixed point that the circuit settles at, then the circuit is "**not combinational**."
- Conversely, if for every assignment to the primary inputs there are no $\perp$ values in the fixed point that the circuit settles at, then the circuit is "**combinational**."

We sometimes abuse this terminology: we say that *specific* input assignments are "combinational" or not, meaning the circuit computes definite Boolean values for these input assignments or not. 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 $\perp$ values persist at the primary outputs; this would not alter our algorithm materially, so here we use the more stringent definition that no $\perp$ values can persist on *any* of wires in the circuit, whether these be internal or at the primary outputs.

II. Analysis 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 [36].
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 9, this step is straight-forward. 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:

Bit 0	Bit 1	Value
0	0	0
0	1	$\perp$
1	0	$\perp$
1	1	1

Fig. 9. Dual-rail encoding scheme for ternary values.

$$\begin{aligned} f_0 &= a_0 b_0 + a_1 b_0 \bar{b}_1 \\ f_1 &= a_1 b_1 + a_0 b_1 \bar{b}_0 \end{aligned}$$

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 $\perp$, 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 $\perp$, 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

$$\begin{aligned} x_i = {} & \bar{d}_1 \bar{d}_2 \bar{f}_1 \bar{f}_2 + d_1 d_2 f_1 f_2 + \\ & \bar{d}_1 d_2 \bar{f}_1 f_2 + \bar{d}_1 d_2 f_1 \bar{f}_2 + \\ & d_1 \bar{d}_2 \bar{f}_1 f_2 + d_1 \bar{d}_2 f_1 \bar{f}_2. \end{aligned}$$

6) For every pair of dummy variables, we set up a **$\perp$-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 (encoding $\perp$). Call the output of the $\perp$-checker y_i for each cut location i.

Note that rather than introducing dummy variables, equivalence checkers, and $\perp$-checkers into the SAT circuit, we could instead append the logically equivalent clauses to the circuit's CNF formula representation to produce the same results. By introducing dummy variables and equivalence gates into the SAT circuit, we are implicitly adding these clauses to the CNF formula. Many modern SAT techniques take advantage of circuit structure alongside the circuit's CNF representation in order to find a result faster [13]. The latter method would not make use of the structural information that

dummy variables, equivalence checkers and $\perp$-checker add to the circuit.

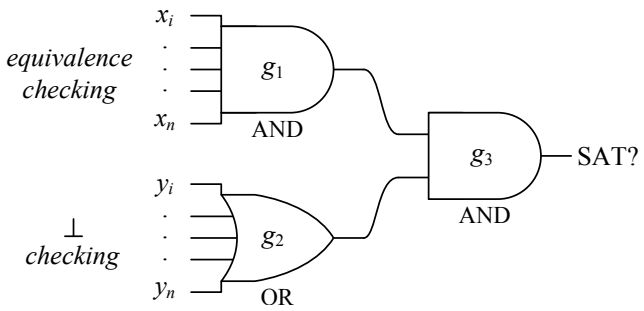


Fig. 10. Constructing the SAT instance.

7) Finally, as illustrated in Figure 10, 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 11, consisting of four NAND *gates. Note that there are four cycles. By inserting dummy variables d and e, we obtain the circuit in Figure 12 (This circuit is acyclic). Next, we replace each gate with a dual-rail version; we feed in pairs of dummy variables, d_0, d_1, and e_0, e_1, corresponding to each of the previous dummy variables; we double the primary inputs a and b; we add two equivalence-checkers, producing x_0 and x_1; we add two $\perp$-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.*

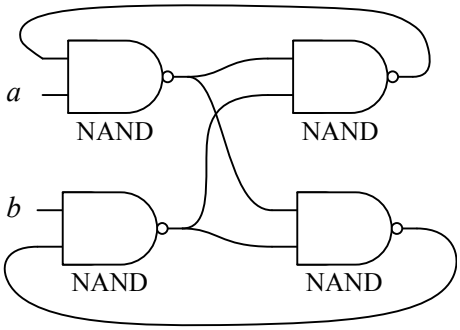


Fig. 11. A cyclic circuit

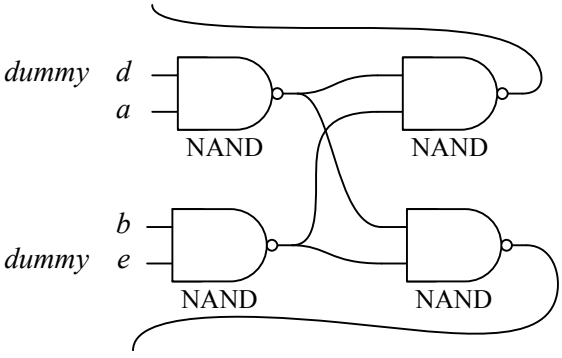


Fig. 12. The circuit in Figure 11 with cycles broken.

This circuit, shown in Figure 13, forms the SAT instance with six primary input variables: a, b, d_0, d_1, e_0, and e_1. We see that for a primary input assignment of $a = b = 1$, $d_0 = \bar{d}_1$, and $e_0 = \bar{e}_1$, $\perp$ values remain on each pair of rails on the inputs of 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 of Analysis

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 $\perp$. 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 $\perp$ values on some of the wires of the cut set. (A fixed point that contains $\perp$ 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 $\perp$, g_2 is also at 1 and so the SAT circuit evaluates to 1.

IV. Mapping Algorithm

As discussed in the introduction, mapping can break the validity of cyclic circuits. For what follows, define an **unmapped circuit** to be a functional-level representation, i.e., a collection of Boolean functions, prior to mapping to gates. Define a **mapped circuit** to be a gate-level representation. Suppose that SAT-based analysis is performed on a mapped circuit and this analysis concludes that the circuit is not combinational. There are two possible explanations: either the original unmapped circuit was not combinational; or the unmapped circuit was combinational and mapping broke it.

In both cases, SAT-based analysis provides a **satisfying assignment**. This assignment lists the values of the primary inputs and the values of the functions at each cut location. In this assignment, the primary inputs all have values in $\{0, 1\}$ while the functions have values in $\{0, 1, \perp\}$. Together, the values of the primary inputs and the functions describe a state of the mapped circuit that is not combinational: a fixed point in which some of the functions have value $\perp$. With the values in this assignment, one can go back and evaluate the original unmapped circuit. If the assignment also corresponds to a state that is not combinational in the unmapped circuit, then no mapping of the corresponding functions will work. However, if the assignment corresponds to a combinational state in the unmapped circuit, then a problem occurred with the mapping. The satisfying assignment can be used to fix the mapping by introducing additional logic.

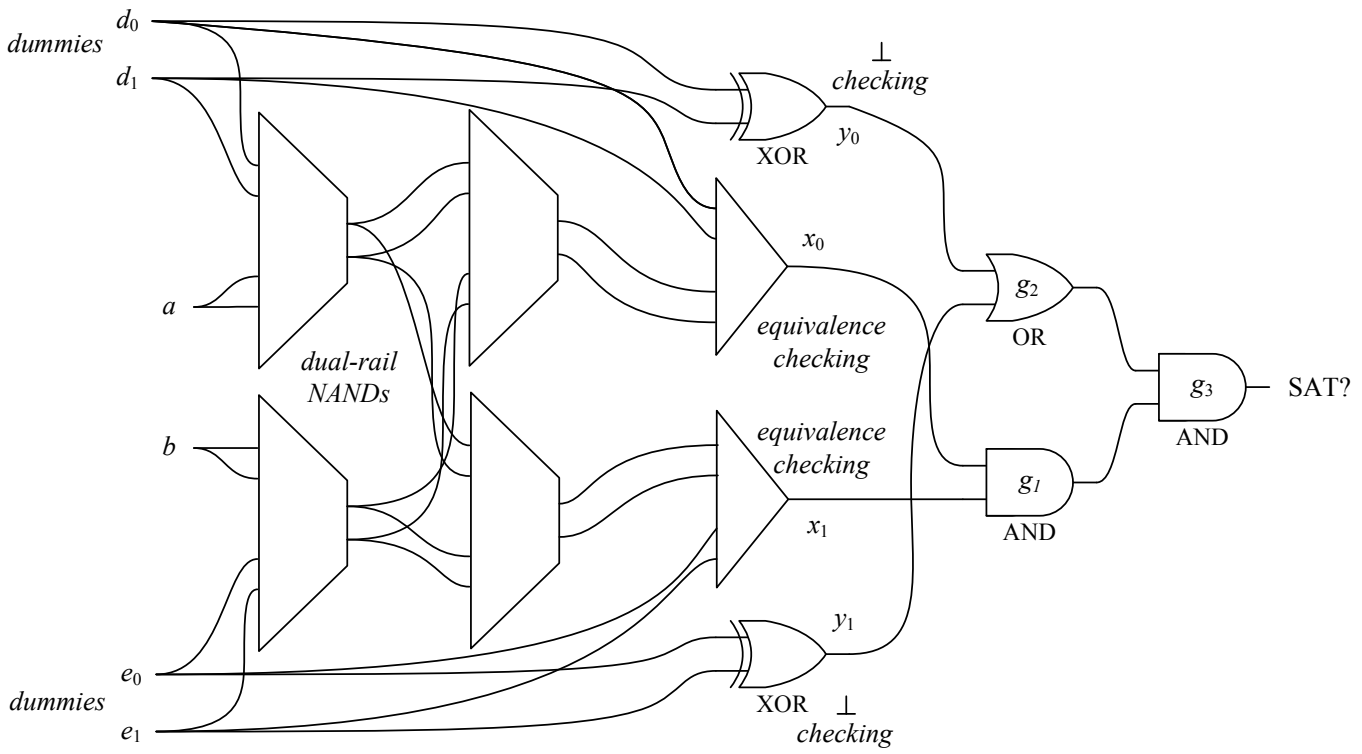


Fig. 13. The SAT circuit corresponding to the cyclic circuit in Figure 11.

Our method for synthesizing this additional logic is as follows.

1) Consider the functions at the cut locations in the unmapped circuit. For each such function f, create an empty list of products and an empty list of sums. Map the circuit to gates. Perform SAT-based analysis to determine if the mapped circuit is combinational.
2) If the SAT solver returns *UNSAT*, skip to Step 4. If the SAT solver returns *SAT*, proceed to Step 3.
3) For each function f at a cut location, set the variables in f's support set to the corresponding values in the satisfying assignment. Then:
   - Let P be a product with literals corresponding to variables with definite values in f's support set. If f evaluates to 1 in the unmapped circuit, add P to f's list of products.
   - Let S be a sum with literals corresponding to the negation of the variables with definite values in f's support set. If f evaluates to 0 in the unmapped circuit, add S to f's list of sums.

   Add the following clause to the SAT instance created in Step 1: a clause that evaluates to 0 for the definite values among the variables in f's support set. Solve the SAT instance again and go back to Step 2.
4) For every function f at a cut location, minimize f's list of products and f's list of sums. In the minimization of the products, select a cover of all the partial assignments that evaluate to 1; in the minimization of the sums, select a cover of all the partial assignments that evaluate to 0.
5) After performing this minimization:
   - For each product P in f's list of products, replace the output of f by $f + P$ in the mapped circuit.
   - For each sum S in f's list of sums, replace the output of f by $(f)(S)$ in the mapped circuit.

   Analyze the circuit again. If the circuit is not combinational, return to Step 1. If the circuit is combinational, then the algorithm is complete.

The intuition behind this approach is that logic can be added to the circuit that *controls* the output of a function for a specific assignment. The assignment is one that, without the additional logic, would result in a value of $\perp$ for the function. The logic added in Step 5 causes the function to evaluate to a definite value for all the partial assignments found in Step 3. Depending on what type of library gates are available, the implementation of Step 5 might differ; if n-input AND and n-input OR are not available, then a balanced tree of ANDs or ORs will have the same effect.

Example 2

Consider again the circuit in Figure 5. If SAT-based analysis is performed on this circuit, the solver will return the satisfying assignment: $a = b = 1$, $f = g = h = \perp$. Apply this assignment to the unmapped circuit consisting of f, g, and h. Observe that, for this assignment, f in the unmapped circuit evaluates to 0. In the mapped circuit, attach an AND *gate to the output of f that evaluates to 0 for the assignment $a = b = 1$. This fixes the mapping. The resulting circuit is shown in Figure 6.*

In Step 4, the logic for fixing the mapping is minimized. This is illustrated in the following example.

Example 3

Consider a cyclic circuit that has been mapped to gates. Suppose that the support set of a function f in the circuit is

$\{a, b, c, d\}$. *Suppose that, after analyzing the circuit, it is found that the value* f *computed by the mapped circuit is* $\perp$ *for the following assignments. Suppose that, for each of these assignments,* f *evaluates to 1 in the unmapped circuit:*

a	b	c	d	*Mapped* f	*Unmapped* f
0	*0*	*0*	$\perp$	$\perp$	*1*
0	*0*	*1*	$\perp$	$\perp$	*1*
0	*1*	*0*	$\perp$	$\perp$	*1*
0	*1*	*1*	$\perp$	$\perp$	*1*

Accordingly, the set of products generated in Step 3 of the algorithm are $\{\bar{a}\bar{b}\bar{c}, \bar{a}\bar{b}c, \bar{a}b\bar{c}, \bar{a}b\bar{c}\}$. *In Step 4, these are minimized to* $\bar{a}$. *In Step 5, the output of* f *is* OR*-ed with* $\bar{a}$ *in the mapped circuit. This fixes the mapping.*

In our experience, relatively few satisfying assignments are ever found for a circuit that needs its mapping fixed. Accordingly, exact methods such as Quine-McCluskey are a viable option [26]. Of course, heuristic methods or multi-level minimization could be used [5], [6]. Note, however, that the minimization in Step 4 is not traditional minimization in a *binary* context. Rather, the requirement is that terms in the sum cover the satisfying assignments in a *ternary* context. This is illustrated in the following example.

Example 4

Consider a cyclic circuit that has been mapped to gates. Suppose that the support set of a function f *in the circuit is* $\{a, b, c, d, e\}$. *Suppose that, after analyzing the circuit, it is found that the value of* f *in the mapped circuit is* $\perp$ *for the following assignments. Suppose that, for each of these assignments,* f *evaluates to 1 in the unmapped circuit:*

a	b	c	d	e	*Mapped* f	*Unmapped* f
1	*1*	$\perp$	*1*	$\perp$	$\perp$	*1*
0	*1*	*1*	*1*	$\perp$	$\perp$	*1*
1	*0*	*0*	*1*	$\perp$	$\perp$	*1*

Here the variables a, b, *and* d, *could be primary inputs or they could be other functions. Clearly,* c *and* e *are functions; primary inputs are never assigned* $\perp$ *values. As in the previous example, these assignments can be minimized to a smaller set. One might assume that assignments of* c *and* e *that are* $\perp$ *can be treated as if they were either 0 or 1 (i.e., treated as don't cares). Assuming this, the set would be minimized to* $\{a\bar{c}d, bcd\}$. *This would result in the additional logic shown in Figure 14. However, in Figure 14, we see that when* $a = b = d = 1$ *and* $c = e = \perp$, *the output of* f *is still* $\perp$. *So the fix did not work!*

In Step 5 of the algorithm, a list of products is OR-ed with the output of a mapped function f. This set of products is meant to cover the partial assignments provided by the SAT solver, that cause f to evaluate $\perp$ when f should evaluate to 1. Call this set of partial assignments $\boldsymbol{A}$. Since the list of products is minimized via two level logic minimization, it only contains prime implicants [26].

Proposition 1

(Necessary condition.) For each partial assignment in $\boldsymbol{A}$*: the list of products in Step 5 must contain a* prime implicant *that evaluates to 1 in order for the mapped circuit to be combinational.*

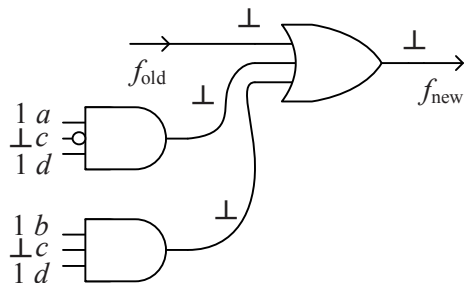


Fig. 14. Example 4: A mapping fix without a product covering assignment $a = b = d = 1$, $c = e = \perp$.

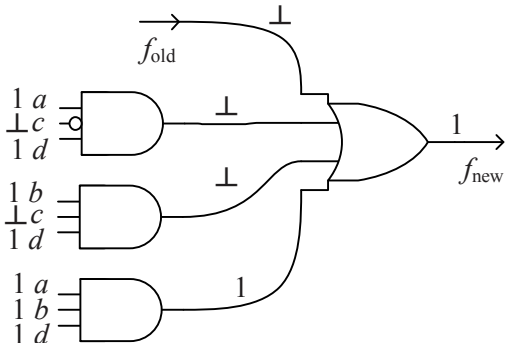


Fig. 15. Example 5: A mapping fix that works.

Proof: Suppose that, for some assignment in $\boldsymbol{A}$, no product evaluates to 1. The output of the OR gate added in Step 5 remains ambiguous, that is, it evaluates to $\perp$: the function f evaluates to $\perp$ for this assignment in the mapped circuit and every product fanning into the new OR gate either evaluates to 0 or $\perp$. Accordingly, this is a necessary condition. ■

An analogous proposition and proof can be made about the list of sums minimized in Step 4 and added to the circuit in Step 5.

We perform two-level minimization applying Proposition 1: instead of the standard criterion of covering minterms and maxterms, we insist on a choice of prime implicants and prime implicates that covers all the partial assignments. We revisit the last example, this time adding logic that fixes the mapping.

Example 5

Consider the set of assignments from Example 4. Applying Proposition 1 during two-level minimization, we obtain the set of products $\{a\bar{c}d, bcd, abd\}$. *Unlike the previous example, the product* abd *is contained in the minimum representation for the partial assignments. Figure 15 shows a mapping when product* abd *is not removed from the two level minimization. For all the assignments listed in the table in the previous example, the newly mapped function behaves correctly in Figure 15.*

V. Proof of Correctness for Mapping

We prove the correctness of our mapping algorithm by demonstrating that 1) it does no harm: it never causes an output to evaluate to $\perp$ that otherwise would not; and 2) it makes progress: each iteration adds logic that corrects partial assignments that were causing non-combinational behavior.

Proposition 2
(Does no harm with products.) Each product P that is OR*-ed with f in Step 5 of the mapping algorithm never evaluates to $\perp$ when f evaluates to 0.*

Proof: Each product P is a redundant product in the computation of f: OR-ing P with f does not expand the set of assignments that causes f to be 1. Consider a function f_{new}, where $f_{\text{new}} = f + P$. Because P is redundant, f_{new} must be equivalent to f. Therefore f_{new} cannot evaluate to $\perp$ while f evaluates to 0 (or else f_{new} and f would not be equivalent). This implies that P cannot be $\perp$ while f is 0. ■

Proposition 3
(Does no harm with sums.) Each sum S that is AND*-ed with f in Step 5 of the mapping algorithm never evaluates to $\perp$ when f evaluates to 1.*

Proof: Each sum S is a redundant sum in the computation of f: AND-ing S with f does not expand the set of assignments that causes f to be 0. Consider a function f_{new}, where $f_{\text{new}} = (f)(S)$. Because S is redundant, f_{new} must be equivalent to f. Therefore f_{new} cannot evaluate to $\perp$ while f evaluates to 1 (or else f_{new} and f would not be equivalent). This implies that S cannot be $\perp$ while f is 1. ■

Propositions 2 and 3 show that each product and each sum that is OR-ed or AND-ed into the mapped circuit never produces non-combinational behavior that was not there before.

Proposition 4
(Makes progress.) Each product (each sum) that is OR*-ed (*AND*-ed) with the output of the mapped function f in Step 5 results in in a definite output for some assignment that otherwise produces $\perp$.*

Proof: Each such product (sum) evaluates to 1 (0) for every partial assignment found in Step 3. Because each such product (sum) fans into the input of an OR (AND) gate that is attached to the output of f, the OR (AND) gate is forced to 1 (0) for every assignment found in Step 3. ■

Finally, we note that algorithm must eventually halt because the circuit has a finite number of possible input assignments.[1] Note that there might not be any unmapped functions that evaluate to a definite value in Step 3. In this case, there is no additional logic to add in Steps 4 and 5; the mapping cannot be fixed. Accordingly, the algorithm terminates with the conclusion that the original unmapped circuit was not combinational.

[1]In our experience the algorithm terminated in very few iterations. For every circuit that we tried, there were less than 10 satisfying assignments that resulted in non-combinational behavior.

VI. Implementation and Results

We implemented the algorithms described in Sections II and IV in the Berkeley ABC environment [23]. ABC invokes the "MiniSAT" SAT Solver [34]. We performed trials on cyclic circuits produced by our tool CYCLIFY from benchmark circuits in the IWLS collection [12]. (For circuits with latches, we extracted the combinational part.) The circuits synthesized by CYCLIFY were read into ABC. We replaced feedback paths with dummy primary inputs. We then ran 10 iterations of `compress2`, a standard and highly effective minimization script on these circuits. For comparison, we also ran 10 iterations of `compress2` on the original acyclic versions. In the following tables, the size that is reported is the number of AND2 gates that are used to represent the circuits in an and-inverter graph (AIG) after logic minimization. We compare the runtimes for the new SAT-based analysis to those using our previous BDD-based approach [29]. Trials were performed on an AMD Athlon 64 X2 6000+ Processor (@ 3Ghz) with 3.6GB of RAM running Linux. Only one core was utilized for the simulations.

Table I lists benchmarks that mapped correctly. Table II lists benchmarks that needed additional logic to correct the mappings. The numbers reported in Table I include the time to:

1) convert the circuits into their ternary equivalent,
2) convert the result to a CNF formula,
3) run the SAT solver to solve the formula.

(CYCLIFY provided a feedback arc set, so a depth-first search to find cut locations was not necessary.) The "Num Gates" column reports the number of AND2 gates in the AIG graph used to represent the benchmark. The "Size Ratio" column is calculated as "Num Gates Cyclic / Num Gates Acyclic." The "Time Ratio" column is calculated as "Time SAT / Time BDD." The results show that for almost every benchmark, the SAT-Based analysis is considerably faster than the old BDD-Based analysis. Note that, for almost every benchmark, CYCLIFY produced a smaller cyclic circuit than the acyclic equivalent.

Table II lists benhmarks that needed to have their mappings corrected. The circuits `5xp1` and `table3` were initially larger than the smallest acyclic representation. For circuit `dk16`, we ran both the acyclic circuit and the cyclic circuit (after remapping) through an additional 10 iterations of compress2. The remapped cyclic circuit still remained slightly larger.

For Table III, we generated random Boolean functions to test our mapping algorithm. For each choice of different numbers of inputs and outputs, we randomly generated 300 circuits. The column "Num Cyclic" lists the number of circuits where a cyclic solution was found. The columns "Num Gates Cyclic" and "Num Gates Acyclic" list the average number of gates in the smallest cyclic and acyclic implementations of the circuit. The "Num Gates Cyclic" column includes the additional gates that were added to fix incorrect mappings. "Num Remapped" lists the number of circuits that needed to have their mappings fixed. "Num Vectors" and "Num Gates Added" list the average number of satisfying assignments returned from the SAT solver and the average number of gates added to the mapping. These two fields were only averaged

over circuits whose initial mapping needed to be fixed. Finally, the last column gives the ratio of the average number of gates in the smallest cyclic and acyclic implementations that were found.

While few of the benchmark circuits required fixes to their mappings, between 10% and 30% of the randomly generated circuits required such fixes. The size reduction of the cyclic versions of the random circuits was, in general, not as significant as the size reduction of the benchmark circuits presented in Table I. Perhaps this was because we were impatient: we set a relatively small timeout when synthesizing the circuits in Table III compared to the timeout when synthesizing the circuits in Tables I and II.

VII. 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 dependencies. 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.

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 CYCLIFY. The SAT-based methodology proposed here can tackle much larger benchmark circuits and it runs orders of magnitudes faster (as anyone familiar with SAT-based methods might have expected).

Synthesizing *cyclic* dependencies is specific concept within a broader topic, synthesizing *functional* dependencies – a topic that has not garnered sufficient attention in the logic synthesis community, in our opinion. BDD's were never up to task: the problem of constructing BDD's with don't cares was never solved. SAT-based techniques are showing much more promise. In particular, *Craig Interpolation* is a very interesting new technique for synthesizing functional dependencies from the *resolution proofs* produced by SAT solvers [19]. We are studying techniques for manipulating and minimizing the resolution proofs obtained through incremental SAT calls, with the aim of affecting large optimizations in circuit structure through changes in functional dependencies [3], [4].

References

[1] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. *Correct Hardware Design and Verification Methods*, pages 254–268, 2005.

[2] J. Backes, B. Fett, and M. D. Riedel. The analysis of cyclic circuits with Boolean satisfiability. In *International Conference on Computer-Aided Design*, pages 143–148, 2008.

[3] J. Backes and M. D. Riedel. The synthesis of cyclic dependencies with Craig interpolation. In *International Workshop on Logic and Synthesis*, pages 24–30, 2009.

[4] J. Backes and M. D. Riedel. Reduction of interpolants for logic synthesis. In *International Conference on Computer-Aided Design*, 2010.

[5] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990.

[6] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[7] R. E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(4):634–649, 1987.

[8] J. Brzozowski and C.-J. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

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

[10] N. Eén and N. Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[11] E. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, 1965.

[12] Benchmarks from the 1993 Int'l Workshop on Logic Synthesis available at http://www.cbl.ncsu.edu/.

[13] Malay K. Ganai, Pranav Ashar, Aarti Gupta, Lintao Zhang, and Sharad Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Design Automation Conference*, pages 747–750, 2002.

[14] D. A. Huffman. Combinational circuits with feedback. In A. Mukhopadhyay, editor, *Recent Developments in Switching Theory*, pages 27–55. Academic Press, 1971.

[15] J.-H R. Jiang, A. Mischenko, and R. K. Brayton. On breakable cyclic definitions. In *International Conference on Computer-Aided Design*, pages 411–418, 2004.

[16] R. Katz. *Contemporary Logic Design*. Benjamin/Cummings, 1992.

[17] W. H. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Transactions on Computers*, C-19(2):162–164, 1970.

[18] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[19] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *International Conference on Computer-Aided Design*, pages 227–233, 2007.

[20] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, 1994.

[21] C. McCaw. *Loops in Directed Combinational Switching Networks*. PhD thesis, Stanford University, 1963.

[22] M. Mendler and M. Fairlough. Ternary simulation: A refinement of binary functions or an abstraction of real-time behavior. *Workshop on Designing Correct Circuits*, 1996.

[23] A. Mishchenko et al. ABC: A system for sequential synthesis and verification, 2007.

[24] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[25] O. Neiroukh, S. A. Edwards, and S. Xiaoyu. Transforming cyclic circuits into acyclic equivalents. *IEEE Transactions on Computer-Aided Design*, 27:17750–1787, 2008.

[26] W. V. O Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59:521–531, 1952.

[27] M. D. Riedel. *Cyclic Combinational Circuits*. PhD thesis, Caltech, 2004.

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

[29] M. D. Riedel and J. Bruck. Timing analysis of cyclic combinational circuits. In *International Workshop on Logic and Synthesis*, 2004.

[30] R. L. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers*, 26(6):606–607, 1977.

[31] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of California, Berkeley, 1992.

[32] T. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, U.C. Berkeley, 1996.

Runtimes (Mapping Initially Correct)						
Benchmark Name	Num Gates Cyclic	Num Gates Acyclic	Time BDD (s)	Time SAT (s)	Size Ratio	Time Ratio
bbsse	90	96	0.08	0.01	0.94	0.13
bw	110	183	0.02	$< .01$	0.6	-
clip	113	181	0.02	0.01	0.62	0.5
cse	128	152	0.23	0.01	0.84	0.04
duke2	309	301	0.49	0.06	1.03	0.12
ex1	205	210	0.26	0.03	0.98	0.12
ex6	61	116	$< .01$	0.01	0.53	-
inc	87	115	$< .01$	$< .01$	0.76	1
planet	381	419	0.25	0.06	0.91	0.24
planet1	377	433	0.13	0.05	0.87	0.38
pma	167	161	0.17	0.02	1.03	0.12
s1	254	339	4.92	0.05	0.75	0.01
s298	1806	1823	106.62	2.07	0.99	0.02
s386	91	102	0.02	0.01	0.89	0.5
s510	189	199	0.03	0.03	0.95	1
s526	129	135	0.01	0.02	0.96	2
s526n	130	117	$< .01$	0.02	1.11	-
s1488	431	500	0.34	0.07	0.86	0.21
sse	87	102	0.02	0.01	0.85	0.5
styr	344	380	0.59	0.06	0.91	0.1
table5	686	639	50.32	0.28	1.07	0.01

TABLE I
RUNTIME COMPARISON FOR CIRCUITS WHOSE INITIAL MAPPING WAS COMBINATIONAL

Runtimes (Mapping Fixed)				
Benchmark Name	Num Added Gates	Num Gates Cyclic	Num Gates Acyclic	Size Ratio
5xp1	6	98	67	1.37
table3	26	833	771	1.06
dk16	17	208	199	1.04

TABLE II
RUNTIME COMPARISON FOR CIRCUITS WHOSE INITIAL MAPPING WAS NOT COMBINATIONAL.

Random Circuits								
Num Inputs	Num Outputs	Num Cyclic	Num Gates Cyclic	Num Gates Acyclic	Num Remapped	Num Vectors	Num Gates Added	Size Ratio
5	9	291	106	108	34	1	8	.98
5	8	300	92	100	66	2	11	.92
5	7	300	84	90	59	3	12	.93
5	6	298	75	78	64	2	8	.96
6	8	300	200	204	84	3	17	.98
6	7	300	180	182	105	3	5	.99
6	6	300	153	160	106	3	15	.96

TABLE III
RESULTS FOR RANDOMLY GENERATED FUNCTIONS OF FIVE VARIABLES.

[33] R. Short. *A Theory of Relations Between Sequential and Combinational Realizations of Switching Functions*. PhD thesis, Stanford University, 1961.

[34] N. Sörensson et al. Minisat v1.13 – a SAT solver with conflict-clause minimization available at http://minisat.se/downloads/.

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

[36] R. Tarjan. Depth-First search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[37] J. F. Wakerly. *Digital Design: Principles and Practices*. Prentice-Hall, 2000.

[38] M. Yoeli and S. Rinon. Application of ternary algebra to the study of static hazards. *Journal of ACM*, 11(1):84–97, 1964.