A

The Synthesis of Cyclic Dependencies with Boolean Satisfiability[1]

John D. Backes and Marc D. Riedel

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

The accepted wisdom is that combinational circuits must have *acyclic* (i.e., feed-forward) topologies. Yet simple examples suggest that this is incorrect. In fact, introducing cycles (i.e., feedback) into combinational designs can lead to significant savings in area and in delay. Prior work described methodologies for synthesizing cyclic circuits with sum-of-product (SOP) and binary-decision diagram (BDD)-based formulations. Recently, techniques for *analyzing* and *mapping* cyclic circuits based on Boolean satisfiability (SAT) were proposed. This paper presents a SAT-based methodology for *synthesizing* cyclic dependencies. The strategy is to generate cyclic functional dependencies through a technique called Craig interpolation. Given a choice of different functional dependencies, a branch-and-bound search is performed to pick the best one. Experiments on benchmark circuits demonstrate the effectiveness of the approach.

1. INTRODUCTION

New ideas pass through three periods:
(*1*) *"It can't be done."*
(*2*) *"It probably can be done, but it's not worth doing."*
(*3*) *"I knew it was a good idea all along!"*

–Arthur C. Clarke (1917–2008)

1.1. Cyclic Combinational Circuits

Digital circuits are classified into two types:

— **Combinational** circuits have outputs that depend only on the current inputs.
— **Sequential** circuits have outputs that depend on past as well as current inputs.

Thus, sequential circuits maintain state, whereas combinational circuits do not.

In order to maintain state, sequential circuits must have *cyclic* configurations (i.e., loops or feedback paths). Indeed, sequential circuits are typically built with the outputs of state-holding elements feeding back to the inputs of blocks of combinational circuitry.

"It can't be done."

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. Indeed, any acyclic circuit is clearly combinational: once the current values of the inputs are set, the signals propagate to the outputs; the outputs are determined regardless of the prior values on the wires, making them independent of the past sequence of inputs. The idea that "combinational" and "acyclic" are synonymous terms is so thoroughly in-

[1]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 by an NSF CAREER Award, No. 0845650.

grained that many textbooks provide the latter as a definition of the former (e.g., [Katz 1992], p. 14; [Wakerly 2000], p. 193).

And yet, circuits with cyclic topologies can be combinational. Consider the circuit in Figure 1. It consists of six alternating AND and OR gates, with inputs x_1, x_2, x_3 repeated. To demonstrate that the circuit is combinational, we label the feedback path with an unknown value y, as shown in Figure 2.

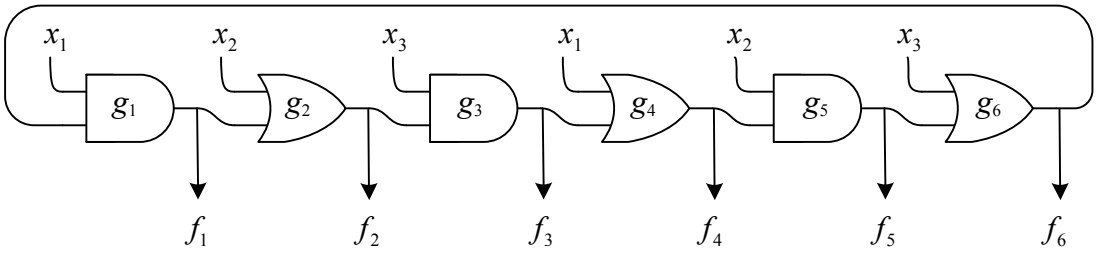


Fig. 1. A cyclic combinational circuit.

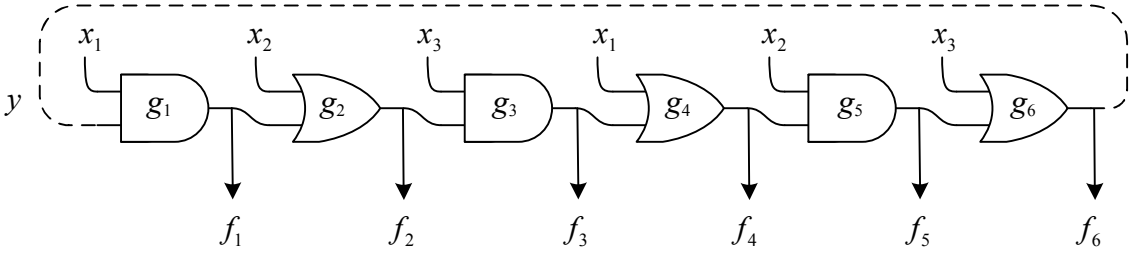


Fig. 2. Analyzing the circuit of Figure 1.

We compute

$$\begin{array}{lllll} f_1 &=& x_1 y & & \\ f_2 &=& x_2 + f_1 &=& x_2 + x_1 y \\ f_3 &=& x_3 f_2 &=& x_3(x_2 + x_1 y) \\ f_4 &=& x_1 + f_3 &=& x_1 + x_3(x_2 + x_1 y) \;=\; x_1 + x_2 x_3 \\ f_5 &=& x_2 f_4 &=& x_2(x_1 + x_2 x_3) \;=\; x_2(x_1 + x_3) \\ f_6 &=& x_3 + f_5 &=& x_3 + x_2(x_1 + x_3) \;=\; x_3 + x_1 x_2. \end{array}$$

(Here addition represents OR and multiplication represents AND.) We see that f_4, and consequently f_5 and f_6, do not depend upon the unknown value. Thus, we compute

$$\begin{array}{lllll} f_1 &=& x_1 f_6 &=& x_1(x_3 + x_1 x_2) \;=\; x_1(x_2 + x_3) \\ f_2 &=& x_2 + f_1 &=& x_2 + x_1(x_2 + x_3) \;=\; x_2 + x_1 x_3 \\ f_3 &=& x_3 f_2 &=& x_3(x_2 + x_1 x_3) \;=\; x_3(x_1 + x_2). \end{array}$$

Each output depends on the current input values, not on the prior values, and so the circuit is combinational. This example demonstrates that "combinational" and "acyclic" are not synonymous terms.

In 1977, Rivest presented the example in Figure 1, in a paper less than a page long [Rivest 1977]. His work on the topic seems to have gone largely unnoticed by theoreticians and practitioners alike.

"It probably can be done, but it's not worth doing."

Although conceptually possible, one might argue that there is *no point* in designing combinational circuits with feedback. Why should one incorporate feedback paths in the computation of the output values? By definition, the values fed back depend upon the prior state of the circuit, which we want to *ignore* when designing combinational circuits.

Counter-intuitively, it can be advantageous to design combinational circuits with feedback. Consider again the circuit in Figure 1. Note that the six output functions are distinct, and each depends on all three input variables. Moreover, we can show that this cyclic circuit has *fewer* gates than any equivalent acyclic circuit. To see this, note that any acyclic configuration contains at least one gate producing an output function that does not depend on the output of any other gate producing an output function. (If this were not the case, then every output gate would depend upon another and so the circuit would be cyclic.)

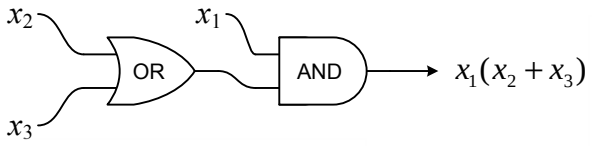


Fig. 3. With fan-in two gates, two gates are needed to compute $x_1(x_2 + x_3)$.

With fan-in two gates, it takes two gates to compute any one of the six functions by itself. This is illustrated in Figure 3. We conclude that an acyclic implementation of the six functions requires at least seven gates, compared to the six in the cyclic circuit.

Generalizing this example, for any odd integer n greater than 1, consider a circuit consisting of n two-input AND gates alternating with n two-input OR gates in a single cycle, with inputs $x_1, \ldots, x_n$ repeated, as shown in Figure 4. Analyzing the general

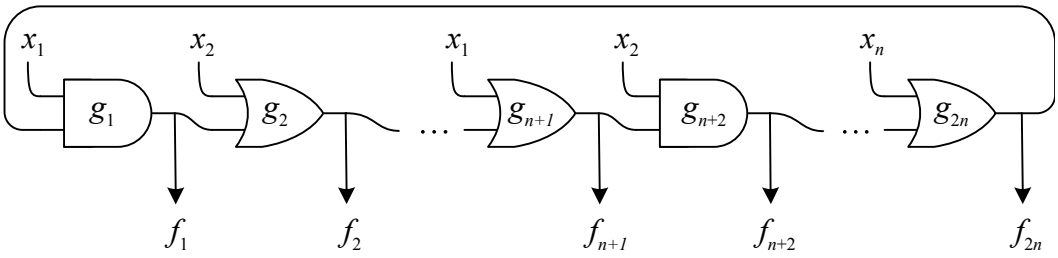


Fig. 4. A cyclic combinational circuit with n inputs (for any odd $n \geq 3$) due to Rivest.

circuit in the same manner as above, we find that it implements the functions

$$\begin{aligned} f_1 &= x_1(x_n + x_{n-1}(\cdots(x_3 + x_2)\cdots)) \\ f_2 &= x_2 + x_1(x_n + \cdots(x_4 x_3)\cdots) \\ &\vdots \\ f_{2n} &= x_n + x_{n-1}(x_{n-2} + \cdots(x_2 x_1)\cdots). \end{aligned}$$

Note that the functions are symmetrical with respect to a cyclic permutation of the variables.

This circuit produces $2n$ distinct output functions, each of which depends on all n input variables. Any acyclic circuit implementing the same $2n$ output functions requires at least $3n - 2$ fan-in two gates. Thus, asymptotically, this circuit is at most two-thirds the size of the smallest possible acyclic circuit computing the same functions.

Inspired by the work of Rivest, we have generated a variety of cyclic examples with the same property as his circuit: they have provably fewer gates than equivalent acyclic circuits. Most notably, we have found a family of circuits that are asymptotically one-half the size [Riedel 2004]. Admittedly, such constructs are of theoretical interest only.

As a practical example, consider the circuit shown in Figure 5, ubiquitous in introductory logic design courses: a 7-segment display decoder. The inputs are four bits, x_0, x_1, x_2, and x_3, specifying a number from 0 to 9. The outputs are 7 bits, a, b, c, d, e, f, and g, specifying which segments to light up in a display – such as that of a digital alarm clock – to form the image of this number. Our goal is to design a circuit that implements the functions shown in Figure 6.

inputs					outputs						
x_3	x_2	x_1	x_0	Digit	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	1	0	0	0	0	0	1	1
0	0	1	0	2	0	1	1	1	1	1	0
0	0	1	1	3	0	0	1	1	1	1	1
0	1	0	0	4	1	0	0	1	0	1	1
0	1	0	1	5	1	0	1	1	1	0	1
0	1	1	0	6	1	1	0	1	1	0	1
0	1	1	1	7	0	0	1	0	0	1	1
1	0	0	0	8	1	1	1	1	1	1	1
1	0	0	1	9	1	0	1	1	0	1	1

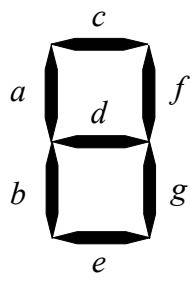


Fig. 5. 7-Segment Display Decoder.

$$
\begin{aligned}
a &= \bar{x}_0 x_2 \bar{x}_3 + \bar{x}_1(\bar{x}_2(x_3 + \bar{x}_0) + x_2\bar{x}_3) \\
b &= \bar{x}_0(x_1\bar{x}_3 + \bar{x}_1\bar{x}_2) \\
c &= \bar{x}_1\bar{x}_2 x_3 + \bar{x}_3(x_0(x_2 + x_1) + \bar{x}_0\bar{x}_2) \\
d &= \bar{x}_1\bar{x}_2 x_3 + \bar{x}_3(x_2(\bar{x}_1 + \bar{x}_0) + x_1\bar{x}_2) \\
e &= \bar{x}_0\bar{x}_1\bar{x}_2 + \bar{x}_3(x_0\bar{x}_1 x_2 + x_1(\bar{x}_2 + \bar{x}_0)) \\
f &= \bar{x}_3(\bar{x}_0\bar{x}_1 + x_0 x_1 + \bar{x}_2) + \bar{x}_1\bar{x}_2 \\
g &= \bar{x}_3(x_2 + x_0) + \bar{x}_1\bar{x}_2.
\end{aligned}
$$

Fig. 6. Target functions for 7-Segment Display Decoder.

With our synthesis methodology, we arrive at the network shown in Figure 7, with the ordering illustrated. This network translates into a cyclic circuit with 27 fan-in two gates. In contrast, standard synthesis techniques produce an acyclic circuit with

$$
\begin{aligned}
a &= \bar{x}_3\bar{x}_0\,\bar{c} + \bar{x}_1\,c \\
b &= \bar{x}_0 e \\
c &= \bar{x}_3 x_2 x_0 + \bar{x}_2(x_3\bar{x}_1 + e) \\
d &= (x_3 + x_2)\,a + x_1\,e \\
e &= \bar{x}_2(x_1 + \bar{x}_0)\,f + \bar{x}_3\,\bar{f} \\
f &= (\bar{x}_2 + \bar{x}_1\bar{x}_0)\,g + \bar{x}_3\,\bar{a} \\
g &= \bar{x}_3\bar{b} + a
\end{aligned}
$$

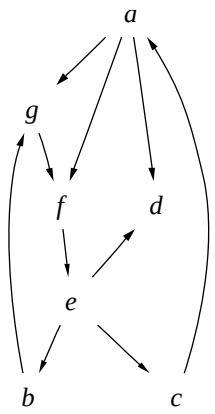


Fig. 7. A cyclic network for the example in Figure 6.

32 fan-in two gates. Note that the network in Figure 7 contains cyclic dependencies; in fact, all the functions except d form a strongly connected component.

Practitioners have observed that cycles sometimes occur in combinational circuits synthesized from high-level descriptions. For instance, cycles can be introduced during resource-sharing optimizations at the level of functional units. Consider the example in Figure 8. There are two functional units, F and G, operating on a datapath X, based on a selecting input c. (Here $F(X)$ and $G(X)$ might be bit-level or word-level arithmetic operations.) If c is 1, then the circuit computes

$$G(F(X)),$$

while if it is 0, it computes

$$F(G(X)).$$

Clearly, this is a valid and efficient design; yet it is all combinational logic and it is cyclic.

Leon Stok lamented that EDA tools were rejecting such designs because there was no way to validate them [Stok 1992]. In response, Malik discussed analysis techniques for cyclic combinational circuits [Malik 1994]. 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 [Shiple 1996].

More recently, Neiroukh and Edwards discussed analysis strategies targeting cyclic circuits that are produced inadvertently during design [Edwards 2003; Neiroukh et al. 2008]. Following a strategy similar to Malik's, they proposed techniques for transforming valid cyclic circuits into functionally equivalent acyclic circuits [Neiroukh et al. 2008]. 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 method for synthesizing cyclic circuits [Riedel and Bruck 2003]. We implemented the method in a package called CYCLIFY, built within

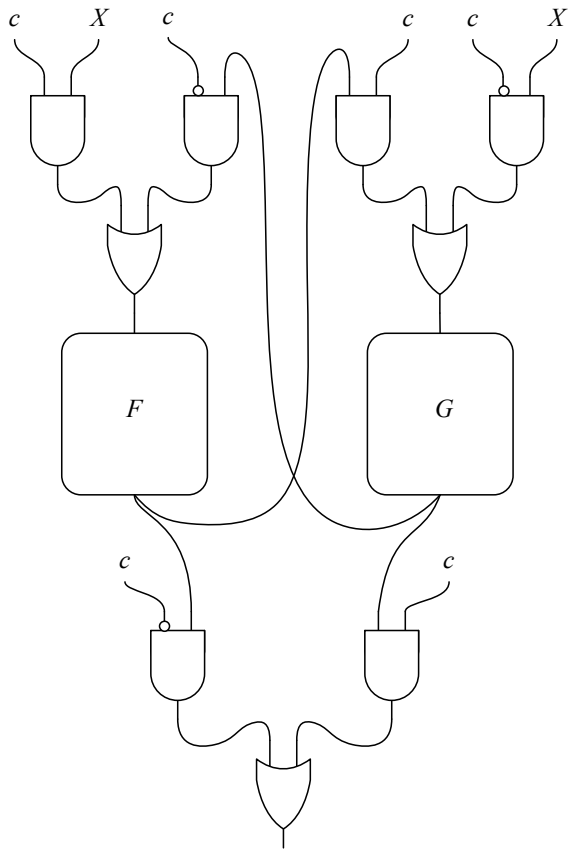


Fig. 8. Functional units connected in a cyclic topology.

the Berkeley SIS environment [Sentovich et al. 1992]. The tool was successful: it reduced the area of benchmark circuits by as much as 30% and the delay by as much as 25%. However, being based on SIS, the analysis routines in CYCLIFY used sum-of-products (SOP) and binary decision diagram (BDD) representations for Boolean functions. These representations limited the size of the circuits that could be analyzed and optimized effectively.

"I knew it was a good idea all along!"

This paper aims to bring the topic of synthesizing cyclic combinational circuits into the modern era, by exploiting the efficiency of SAT solving. So-called SAT-based techniques, based on heuristic solutions to the Boolean satisfiability problem, have proved very successful for tasks such as logic verification and model checking [Amla et al. 2005], [Larrabee 1992].

In related work, we have proposed an efficient SAT-based algorithm for analyzing and mapping cyclic circuits [Backes et al. 2008; Backes and Riedel 2011]. We perform SAT-based validation of cyclic designs at a gate level, after mapping to a library. When mapping breaks the validity of a combinational circuit, SAT-based analysis returns

satisfying assignments; these assignments are used to modify the mapping in order to ensure that the circuit remains combinational.

Admittedly, the task of analyzing cyclic circuits is complex. Yet there is no fundamental obstacle to performing tasks such as verification, mapping, and timing analysis on cyclic circuits. So-called "false-path aware" algorithms for timing analysis take into account false paths, providing tighter bounds on delay than purely topological methods [Kukimoto and Brayton 1997]. The complexity of this sort of timing analysis is, in fact, the same for cyclic circuits as for acyclic circuits. Early formulations based on SOPs and BDDs were never up to the task, but modern SAT-based algorithms are powerful enough to perform false-path aware analysis.

Significantly, SAT-based algorithms lend themselves well to *incremental analysis*. Often analysis and verification tasks are applied iteratively and incrementally in a design flow: small changes are made to improve the circuit and then it is re-analyzed. With incremental SAT solving, new queries can take advantage of cached results of previous queries, making SAT-based analysis very efficient [Eén and Sörensson 2003].

Recently, a technique called Craig Interpolation has been used to leverage the information generated by SAT solvers from unsatisfiable SAT instances [McMillan 2003]. In [Lee et al. 2007] the authors present a methodology for using Craig Interpolation to generate functional dependencies. In this work, we show how this technique can be used to generate cyclic dependencies.

1.2. Organization

This paper is organized as follows. Section 2 provides definitions and describes the notation used throughout the paper. Section 3 discusses the underlying circuit and network models. Section 4 presents the core contribution of the paper: a method for generating cyclic functional dependencies via Craig interpolation. Section 5 describes a branch-and-bound search technique for exploring the space of possible functional dependencies in a network. Section 6 presents synthesis results on benchmarks. Finally, Section 7 discusses future directions of research.

2. DEFINITIONS AND NOTATION

We use the standard notation: addition ($+$) denotes disjunction (OR), multiplication ($\cdot$), denotes conjunction (AND), an $\oplus$ denotes inequivalence (exclusive OR), and an overbar ($\bar{x}$) denotes negation (NOT). The *restriction* operation (also known as the cofactor) of a function f with respect to a variable x,

$$f|_{x=v},$$

refers to the assignment of the constant value $v \in \{0,1\}$ to x. A function f *depends* upon a variable x iff $f|_{x=0}$ is not identically equal to $f|_{x=1}$. Call the variables that a function depends upon its *support set*.

We use superscripts to denote a function's ON and OFF sets: for a function $f(x_0, x_1, \ldots, x_n)$, we write $f(x_0, x_1, \ldots, x_n)^1$ to denote its ON set (i.e., the set of assignments to variables $x_0, x_1, \ldots, x_n$ where f evaluates to 1); we write $f(x_0, x_1, \ldots, x_n)^0$ to denote its OFF set (i.e., the set of assignments to variables $x_0, x_1, \ldots, x_n$ where f evaluates to 0).

An appearance of a variable in a Boolean formula, either negated or nonnegated, is refered to as a *literal*. A *clause* is an OR of literals. A Boolean formula is in conjunctive normal form (CNF) if it is an AND of clauses. A CNF formula is said to be *satisfiable* if there is some assignment of its variables that causes the formula to evaluate to true. A CNF formula is said to be *unsatisfiable* if there is no assignment of its variables that causes the formula to evaluate to true. We sometimes refer to a CNF formula as a *SAT*

Instance. We will also refer to a circuit with a single primary output as a SAT instance; the satisfiability of the primary output can be represented as a CNF formula.

3. CIRCUIT AND NETWORK MODEL

Analysis of an acyclic circuit is transparent. We first evaluate the gates connected only to primary inputs, and then gates connected to these and primary inputs, and so on, until we have evaluated all gates. The previous values of the internal signals do not enter into play.

We adopt a *ternary framework* for analysis. We assume that, at the outset, all wires in a circuit have *undefined* values, which we denote with the symbol $\perp$. Here $\perp$ captures both uncertainty as well as possible ambiguity: the signal might be 0 or 1 – but we do not know which; or it might not even have logical value, i.e., it could be a voltage value between logical 0 and logical 1. We say that a variable's value is *definite* or *known* if its value is 0 or 1 and that it is *indefinite* or *ambiguous* if it is $\perp$. The idea of three-valued logic for circuit analysis is well established. It was originally proposed for the analysis of *hazards* in combinational logic [Yoeli and Rinon 1964]. Bryant popularized its use for verification [Bryant 1987], and it has been widely adopted for the analysis of asynchronous circuits [Brzozowski and Seger 1995].

Conceptually, when validating a cyclic circuit, we apply definite values to the inputs, and track the propagation of signal values. Initially, each gate has an output value of $\perp$. We ask: is there sufficient information to conclude that the gate output is 0 or 1? If yes, we assign this value as the output; otherwise, the value $\perp$ persists. For instance, with an AND gate, if the inputs include a 0, then the output is 0, regardless of other $\perp$ inputs. If the inputs consist of 1 and $\perp$ values, then the output is $\perp$. Only if all the inputs are 1 is the output 1. This is illustrated in Figure 9. Input values that determine the gate output are called *controlling*.

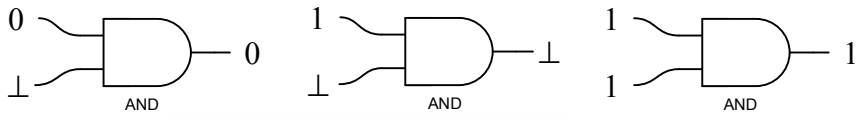


Fig. 9. An AND gate with 0, 1, and $\perp$ inputs.

Consider the circuit fragment in Figure 10. 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 unknown, possibly undefined value. (For instance, in an actual circuit, it could indicate a voltage value exactly half way between logical 0 and logical 1.) In our analysis, we remain agnostic: the output of the OR gate is $\perp$.

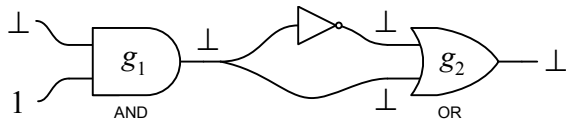


Fig. 10. An illustration unknown/undefined values $\perp$.

In the analysis, we track the propagation of well-defined signal values. Once a definite value is assigned to an internal wire, this value persists for the duration (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 [Brzozowski and Seger 1995]. We adopt the following definition.

> A circuit is *combinational* iff, for every assignment of input values, with all the wires initially set to $\perp$, the circuit reaches a fixed point that does not contain any $\perp$ values.

We illustrate our circuit model with cyclic examples: one that is not combinational and one that is.

Example 3.1.

Consider the circuit shown in Figure 11, consisting of an AND *gate* g_1*, an* OR *gate* g_2*, and an* AND *gate* g_3*, in a cycle. By inspection, note that if* $x_1 = 0$ *then* f_1 *assumes value 0; if* $x_2 = 1$ *then* f_2 *assumes value 1; and if* $x_3 = 0$ *then* f_3 *assumes value 0. But what happens if* $x_1 = 1$*,* $x_2 = 0$ *and* $x_3 = 1$*? In this case, all the outputs equal* $\perp$*, as illustrated in Figure 12. The outcome for all eight cases is shown in Figure 13. We conclude that the circuit is not combinational.*

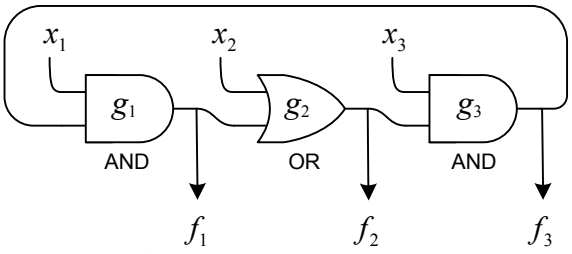


Fig. 11. A cyclic circuit that is not combinational.

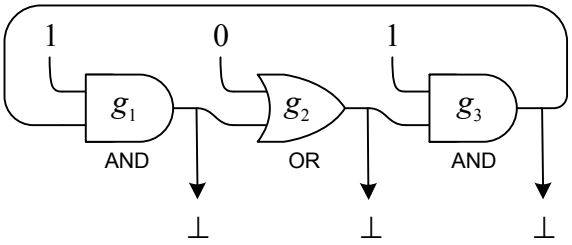


Fig. 12. The circuit of Figure 11 with $x_1 = 1$, $x_2 = 0$ and $x_3 = 1$.

x_1	x_2	x_3	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	$\perp$	$\perp$	$\perp$
1	1	0	0	1	0
1	1	1	1	1	1

Fig. 13. Analysis of the circuit in Figure 11.

Example 3.2. *Consider the circuit in Figure 14. Let us consider a specific assignment of values to the inputs: suppose that we assign* $x_1 = 1, x_2 = 0, x_3 = 1$*, as shown in Figure 15. Gates* g_1*,* g_3*,* g_5 *and* g_7 *produce outputs of 1, 0, 0, and 1, respectively. Gate* g_2 *produces an output of 1. Gate* g_8 *produces an output of 0. Gate* g_9 *produces an output of 0. Gate* g_6 *produces an output of 0. Finally, gate* g_4 *produces an output of 0. The analysis for all eight input combinations is summarized in Table 16. We conclude that the circuit is combinational.*

4. FUNCTIONAL DEPENDENCIES

The algorithms and concepts presented in this paper are applicable to technology-independent synthesis. At this level, a circuit is specified as a network that computes Boolean functions. Ultimately, such a network gets mapped to gates in a specific technology. The validity of a cyclic combinational circuit is properly established in terms of *controlling values* at the technology level. At the network level, we validate circuits in terms of *functional dependencies*. The notion of a function depending on a variable is similar but not identical to the concept of a Boolean value controlling the output of a gate. There can be subtle issues when mapping valid network-level cyclic specifications to gate-level specifications. For a discussion of theses issues, we refer the reader to [Backes et al. 2008; Backes and Riedel 2011].

At the network level, a circuit is specified as a collection of nodes $\mathcal{N}$. Associated with each node is a *node function* f_i and a corresponding *internal variable* y_i, $0 \leq i \leq n-1$. (We sometimes abuse the notation by using the same name for the function and the corresponding internal variable, saying calling them both f_i). The node functions can depend on input variables as well as on other internal variables. In a network's *dependency graph*, a directed edge is drawn from node i to node j iff the node i is in the support set of node function f_j.

The process of multilevel logic synthesis typically consists of an iterative application of minimization, decomposition, and restructuring operations [Brayton et al. 1990]. An important step at the technology-independent stage is the task of structuring *functional dependencies*. (With SOP representations, this step was called *substitution* or *resubstitution*.) In this step, node functions are expressed or re-expressed in terms of other node functions as well as the primary inputs.

For each node function, different choices might be available as dependencies yielding alternative expressions of varying cost. Throughout this paper, we will focus on *support set size* as our cost metric. Given the focus on technology-independent synthesis algorithms, based on Boolean satisfiability, this metric is appropriate. (If we were using an SOP representation, we could use literal counts instead.) Consider the functions f_1 and f_2,

$$f_1 = bcx + bdx + ab \tag{1}$$

$$f_2 = abc\bar{x} + cx + d. \tag{2}$$

Figure 17 shows four different expressions for the functions and the corresponding dependency graphs. Figure 17.a shows expressions for f_1 and f_2, both in terms of the primary input variables only. With a support set of $\{a, b, c, d, x\}$, the cost of both of these expressions is 5, so the total cost is 10.

Figures 17.b and 17.c show alternate expressions, obtained by introducing functional dependencies. In Figure 17.b, f_1 is expressed in terms of f_2 and $\{a, b, x\}$. Accordingly, the total cost is 9. In Figure 17.c, f_2 is expressed in terms of f_1 and $\{c, d, x\}$. Accordingly, the total cost is also 9.

In existing methodologies, a total ordering is enforced among the functions in this phase in order to ensure that no cycles occur. In this example, the ordering of $f_2 \sqsubseteq f_1$

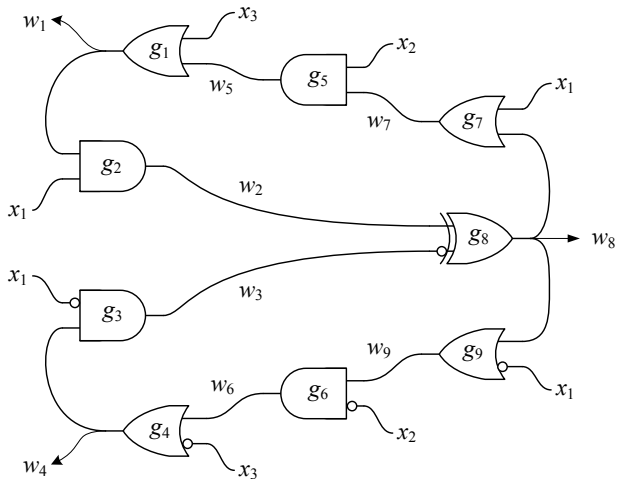


Fig. 14. A cyclic combinational circuit with two cycles.

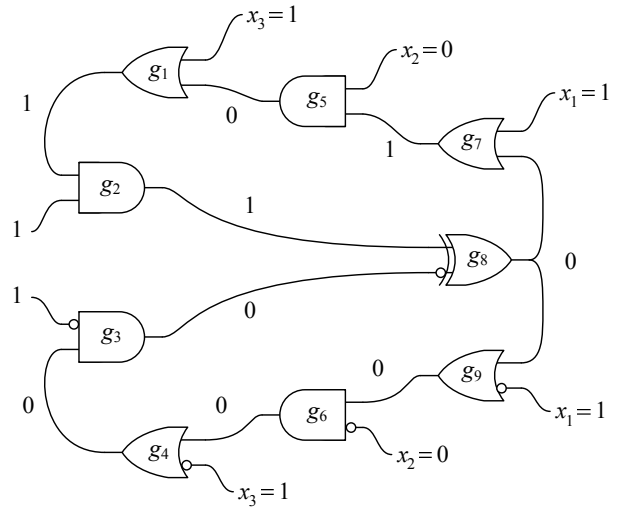
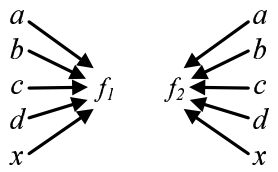


Fig. 15. The circuit in Figure 15 with $x_1 = 1, x_2 = 0, x_3 = 1$.

x_1	x_2	x_3	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9
0	0	0	0	0	1	1	0	1	0	0	1
0	0	1	1	0	1	1	0	1	0	0	1
0	1	0	0	0	1	1	0	0	0	0	1
0	1	1	1	0	0	0	1	0	1	1	1
1	0	0	0	0	0	1	0	1	1	1	1
1	0	1	1	1	0	0	0	0	1	0	0
1	1	0	1	1	0	1	1	0	1	0	0
1	1	1	1	1	0	0	1	0	1	0	0

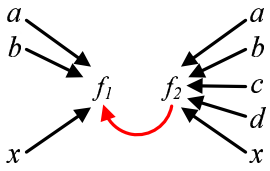
Fig. 16. Analysis summary for the circuit of Figure 14.

$$f_1 = bcx + bdx + ab$$
$$f_2 = abc\bar{x} + cx + d$$



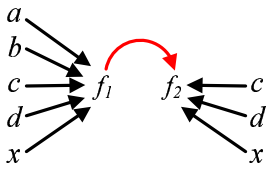
(a) $f_1(a, b, c, d, x)$ and $f_2(a, b, c, d, x)$

$$f_1 = bxf_2 + ab$$
$$f_2 = abc\bar{x} + cx + d$$



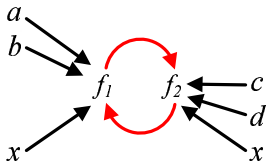
(b) $f_1(a, b, x, f_2)$ and $f_2(a, b, c, d, x)$

$$f_1 = bcx + bdx + ab$$
$$f_2 = cx + cf_1 + d$$



(c) $f_1(a, b, c, d, x)$ and $f_2(c, d, x, f_1)$

$$f_1 = bxf_2 + ab$$
$$f_2 = cx + cf_1 + d$$



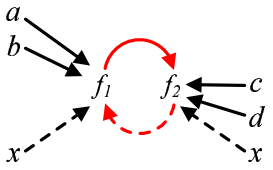
(d) $f_1(a, b, x, f_2)$ and $f_2(c, d, x, f_1)$

Fig. 17. Four different implementations of two functions, f_1 and f_2, of five variables a, b, c, d, and x.

would produce the expressions in Figure 17.b; the ordering of $f_1 \sqsubseteq f_2$ would produce the expressions in Figure 17.c. Insisting upon an ordering means that we would have to choose one of these two results.

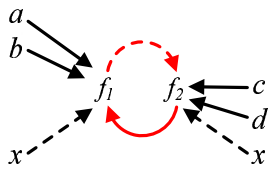
However, if we allow cyclic dependencies, we can find a better solution. Figure 17.d show expressions for f_1 and f_2 with support sets of $\{a, b, x, f_2\}$ and $\{c, d, x, f_1\}$, so a total cost 8. As the dependency graph in Figure 17.d illustrates, the functional dependencies are cyclic. Yet for every assignment of the primary input variables a, b, c, d, and x, the functions evaluate to definite Boolean values. The functions and dependency graphs for functions f_1 and f_2 when x is 0 and x is 1 are shown in Figure 18. We see that, for any assignment of x, the cyclic dependency between f_1 and f_2 is broken, so the result is combinational.

$$f_1 = ab$$
$$f_2 = cf_1 + d$$



(a) $f_1(a, b, 0, f_2)$ and $f_2(c, d, 0, f_1)$

$$f_1 = ab + bf_2$$
$$f_2 = c + d$$



(b) $f_1(a, b, 1, f_2)$ and $f_2(c, d, 1, f_1)$

Fig. 18. Functions $f_1(a, b, x, f_2)$ and $f_2(c, d, x, f_1)$ with $x = 0$ and $x = 1$. For both values of x, the dependency graphs become acyclic.

Of course, not all choices of cyclic dependencies are valid. Many will result in networks that are not combinational. Suppose we wish to compute some complicated function f and its complement $\bar{f}$. Saying that

$$f = \bar{f},$$
$$\bar{f} = f,$$

is evidently meaningless.

In an earlier era, functional dependencies were generated through SOP minimization with don't cares [Brayton et al. 1990]. The main contribution of this paper is an efficient strategy for synthesizing valid cyclic dependencies, based on the modern concepts of Craig interpolation and Boolean satisfiability.

4.1. Functional Dependencies with Craig Interpolation

In a seminal paper, McMillan proposed a SAT-based method for symbolic model checking based on computing so called Craig interpolants [McMillan 2003]. In [Lee et al. 2007], the method was applied to the problem of synthesizing functional dependencies. Broadly, the strategy is to formulate an instance of Boolean satisfiability (SAT) that asks whether or not a target function can be implemented with a certain support set. A proof of unsatisfiability, returned by a SAT solver, is converted into a circuit that computes the target function. We give a brief review of the method here, noting that in its current form, it is only applicable to acyclic orderings. In the next section, we generalize the method to cyclic orderings.

The method constructs a miter, as shown Figure 19. Here f_0 is the target function. The satisfiability of the primary output of this circuit indicates whether or not there exists a dependency function $h(f_1,f_2,f_3)$ that can be used to represent f_0 for some network. Here f_0 *Left* and f_0 *Right* are two copies of the same network. The primary inputs x_0, x_1, ..., x_n (referred to as X) are the primary inputs to f_0 *Left*. The primary inputs x_0*, x_1*, ..., x_n* (referred to as X*) are the primary inputs to f_0 *Right*; these

are distinct sets of variables, but in direct correspondence with one another: $f_i(X)$ is equivalent to f_i*(X*) where the assignment of X is equal to the assignment of X*.

If the primary output of this circuit is satisfied, then this indicates that f_0 evaluates to a different value from f_0* while functions f_1, f_2, and f_3 evaluate to the same values of f_1*, f_2*, f_3*, respectively, on each side of the circuit for some assignment of X and X*. Clearly this indicates that the ON set $f_0(f_1,f_2,f_3)^1$ is not disjoint from the OFF set $f_0(f_1,f_2,f_3)^0$. Accordingly, there is no function $h(f_1,f_2,f_3)$ that is equivalent to $f_0(X)$ (or to f_0*(X*)).

If the primary output of the circuit is unsatisfiable for all assignments of X and X*, this indicates that either f_0 (or f_0*) is a constant 1 or 0, or that the ON set $f_0(f_1,f_2,f_3)^1$ is disjoint from the OFF set $f_0(f_1,f_2,f_3)^0$. This indicates that there is some function $h(f_1,f_2,f_3)$ that is functionally equivalent to $f_0(X)$.

In [Lee et al. 2007], a method is proposed for finding the dependency function h using Craig interpolation. The underlying details of the approach to computing h are not important; it is only important that the reader understands that if the ON set of a function $f(f_0,f_1,\ldots,f_n)^1$ is disjoint from the OFF set $f(f_0,f_1,\ldots,f_n)^0$ then a function h can be computed by generating an interpolant from a SAT instance that is similar to that in Figure 19.

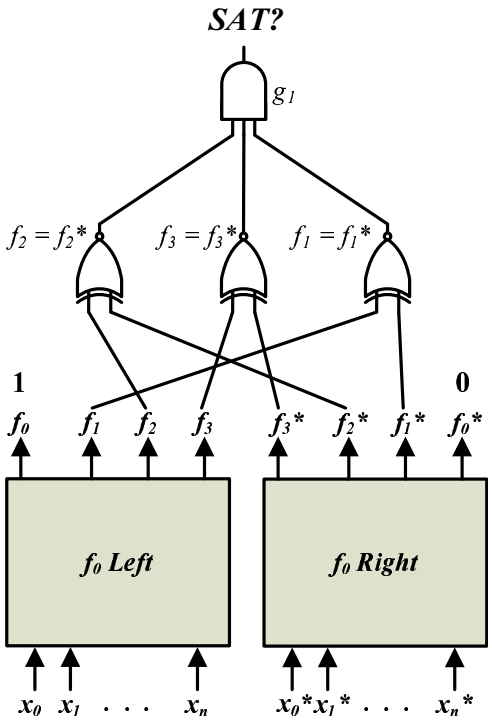


Fig. 19. A miter that checks to see if f_0 can be specified in terms of f_1, f_2, and f_3.

4.2. Generating Cyclic Functional Dependencies

A cyclic circuit is not combinational if, for some assignment of the circuit's primary inputs, the value of some function in the circuit remains ambiguous. In a sense, determining whether or not a cyclic circuit is combinational is a similar problem to that of

determining whether or not a target function can be implemented in terms of a specific support set. In both problems, a negative answer can be proven by comparing pairs of rows of a function's truth table. This is illustrated in the following example.

Figure 20 shows the truth tables for two functions f_0 and f_1. Consider the third and fourth rows of the truth table for function f_0 and the first and second rows of the truth table for function f_1. For each pair of rows, the primary input variables are assigned the same values ($a = c = 0, b = 1$). However, the output values of f_0 and f_1 both toggle between 1 and 0. So, for this assignment, the value of f_0 depends on the value of f_1 and the value of f_1 depends on the value of f_0. A fixed point is reached; because of the mutual dependence, the values of f_0 and f_1 are both $\perp$ in the fixed point. Figure 21 shows the functions f_0 and f_1 and the resulting dependency graph under this assignment.

PROPOSITION 4.1.

Let G be a cyclic dependency graph and let T be the set of truth tables for each function in G. Let R be a set of doubles where each double corresponds to a pair of rows in one of the truth tables. More formally, R is expressed as:

$$R = \{\ \{x, y\} \mid \exists t((t \in T) \wedge (x \in t) \wedge (y \in t) \wedge (x \neq y))\ \}$$

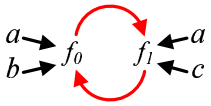
with the added condition that there is only one pair $\{x, y\} \in R$ for each truth table.

G is not combinational if and only if, for some choice of R, the following three conditions hold.

(*1*) *The primary input variables have the same value in every row in every element of R.*
(*2*) *If the output value of some function in some element of R is the same for each of the two rows, then the function also has this value in every other element of R for which this function appears*[2].
(*3*) *The output value of some function differs between the rows in some element of R.*

a, b, f_1	f_0
0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	1

a, c, f_0	f_1
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	0
1 1 1	0



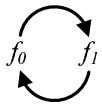
$$f_0 = \bar{a}\bar{b} + ab + f_1 b$$
$$f_1 = \bar{f}_0\bar{a} + \bar{a}c$$

Fig. 20. The truth tables for two functions. The cyclic dependency graph containing these two functions is not combinational.

PROOF.

The first two conditions force the choice of R to correspond to a fixed point in G reached by some primary input assignment.

[2]If some function f is not in the support set of some function g, then f will not appear in the pair of rows selected from g's truth table.



$$f_0 = f_1 \quad (3)$$
$$f_1 = \bar{f}_0 \quad (4)$$

Fig. 21. The dependency graph for the functions in Figure 20 for the assignment: $a = c = 0, b = 1$. The dependency graph is not combinational.

The first condition asserts that the assignment of the primary input variables must be the same in every row of every element of R. If the primary input assignment is a controlling assignment for some function, then that function's output value will not differ between the two rows in that function's corresponding element in R.

The second condition asserts that if the output value of some function is the same between two rows in some element of R, then the variable corresponding to this function in other rows of other elements of R must also be assigned this value. Essentially this condition guarantees that if the value of some function is controlled to either 0 or 1, then this value is propagated to every other function that contains the function as a support variable. If this value causes another function to be controlled, then the value of that function propagates to other functions containing that function as a support variable. As was discussed in Section 3, eventually this propagation halts, and the circuit reaches a unique fixed point.

However, the value of some function might not be controlled by the value of its support variables. If the output value of some function differs between two rows in some element of R, this indicates that the output value of the function is ambiguous. In other words, if a function's output value differs between two rows, this corresponds to that function evaluating to $\perp$.

The third condition asserts that one of the functions evaluates to $\perp$ in the fixed point. Our definition of combinationality states that if a $\perp$ value persists in a fixed point reached by some primary input assignment, then the dependency graph is not combinational. For a network that is not combinational, a choice of R that corresponds to this fixed point will satisfy all three of these conditions.

Similarly, a combinational dependency graph never contains a $\perp$ value in its fixed point for any assignment of its primary input variables. Therefore these three conditions can never be satisfied for any choice of R for a network that is combinational. □

Craig Interpolation provides an implementation for each target function in a dependency graph [Lee et al. 2007]. Given this implementation, a SAT instance can be formulated that is satisfiable if and only if the three conditions above are met. A circuit whose satisfiability indicates that these three conditions are met for the functions in Figure 20 is shown in Figure 22.

The SAT instance contains two copies of functions $f_0(a, b, f_1)$ and $f_1(a, b, f_0)$. In each copy of these two circuits, the primary input variables are kept the same (satisfying Condition 1 of Proposition 4.1). Additional logic is added that computes the *OR* of the *Exclusive OR* of each copy of each function (satisfying Condition 3 of Proposition 4.1). Finally, the additional clauses shown in the box on the upper left-hand side of the figure can be added to the SAT instance to assert that Condition 2 holds. If the SAT instance is satisfiable, then all three conditions are satisfied and the cyclic dependency between functions $f_0(a, b, f_1)$ and $f_1(a, b, f_0)$ is proven to be non-combinational.

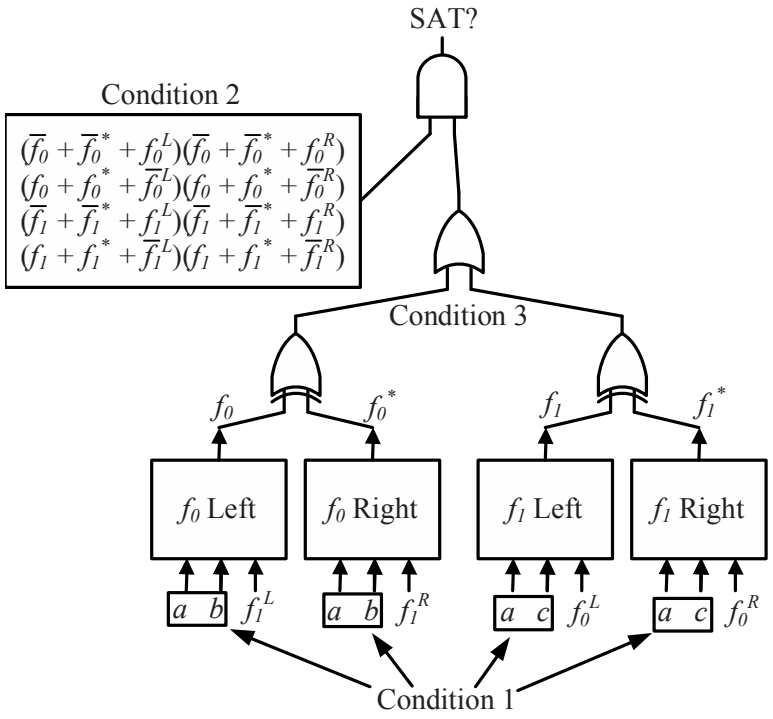


Fig. 22. A SAT instance that verifies whether or not the functions described in Figure 20 are combinational.

4.3. General Method

We sketch the steps to generate the SAT instance for any set of functions $F = \{f_0, f_1, \ldots, f_{n-1}\}$ of variables $X = \{x_0, x_1, \ldots, x_{m-1}\}$

(1) Generate an implementation for each target function in terms of its support variables via Craig interpolation. Create two copies of each of these implementations. Refer to one copy as the *left* copy and the other copy as the *right* copy. We define $CNF_i^R(X, F)$ and $CNF_i^L(X, F)$ to be the set of clauses representing the logic for the left and right copies respectively, of function f_i. Here X is the set of primary input variables in the support set of function f_i and F is the set of internal variables in the support set of function f_i.

(2) Share the same primary input variables X between every copy. Share the same internal variables between every left copy and share the same internal variables between of every right copy. Let $F^L = \{f_0^L, f_1^L, \ldots, f_{n-1}^L\}$ be the set of left internal variables and let $F^R = \{f_0^R, f_1^R, \ldots, f_{n-1}^R\}$ be the set of right internal variables.

$$c_1 = \prod_{i=0}^{n-1}(CNF_i^L(X, F^L) \leftrightarrow f_i)(CNF_i^R(X, F^R) \leftrightarrow f_i^*) \tag{5}$$

(3) Assert the *OR* of the *Exclusive OR* of each left and right copy of each function:

$$c_3 = \sum_{i=0}^{n-1}(f_i \oplus f_i^*) \tag{6}$$

(4) For each function, assert that the corresponding left internal variable is *TRUE* if the left and right copies of the function are both *TRUE*. For each function, assert that the corresponding left internal variable is *FALSE* if the left and right copies

of the function are both *FALSE*. The analogous assertions must also be made for each right internal variable.

$$c_2 = \prod_{i=0}^{n-1}(\bar{f}_i + \bar{f}_i^* + f_i^L)(\bar{f}_i + \bar{f}_i^* + f_i^R)(f_i + f_i^* + \bar{f}_i^L)(f_i + f_i^* + \bar{f}_i^R) \quad (7)$$

PROPOSITION 4.2.
Some choice of R for some set of functions satisfies the three conditions in Proposition 4.1 if and only if $(c_1)(c_2)(c_3)$ is satisfiable.

PROOF.
Step 1 of the general method creates two copies of every function. The value of the support variables in each copy corresponds to the value of the variables in each element of R. The conditions in c_1 assert that the primary input variables must be assigned the same value in every copy of every function. This corresponds to Condition 1 in Proposition 4.1. The conditions in c_3 assert that some function's output value differs between its left and right copies. This corresponds to Condition 3 in Proposition 4.1.

Finally, c_2 asserts that if the value of some function is the same between its left and right copies, then the support variables corresponding to this function in every other copy are also assigned this value. This corresponds to Condition 2 of Proposition 4.1. If the SAT instance $(c_1)(c_2)(c_3)$ is satisfiable, then all the conditions of 4.1 can be met for some choice of R. If $(c_1)(c_2)(c_3)$ is unsatisfiable, then the three conditions from Proposition 4.1 can never be simultaneously satisfied, and the network is deemed combinational.

□

5. SYNTHESIZING CYCLIC DEPENDENCIES

Given a choice of functional dependencies, that is to say, a choice for the support set of each target function, the algorithm in the previous section provides a constructive method for synthesis: if the answer to the SAT-based query is "unsatisfiable" then, through Craig interpolation, the algorithm provides the logic that implements the target functions with the specified support set.

In this section, we describe a synthesis methodology for finding the best choice of functional dependencies. Our cost metric is the size of the support set of each function. In the corresponding dependency graphs, this corresponds to the fewest possible edges. To accomplish this task, we use a branch-and-bound algorithm that searches through the space of possible dependency graphs.

This algorithm is described with pseudocode in Figure 23. The routine "Synthesis" receives a set of Boolean functions as arguments. It first constructs a list of possible support sets for each function. Initially, it chooses a dependency graph containing the smallest possible support set for each function. This solution, as well as the list of possible support sets for each function, is sent to the "BreakDown" routine.

The "BreakDown" routine checks to see if the dependency graph that it is given is combinational. If the graph is not combinational, it iterates over all the functions that are found to be non-combinational.[3] For each of these functions, the current support set is replaced by the next smallest support set available in the list. If the dependency graph containing this next smallest solution is smaller that the best current solution,

[3]This can be accomplished by repeatedly solving a slightly modified version of the SAT instance described in the previous section. The SAT instance is modified so that the only the function that it considers is the one included in the OR gate described in Step 3 of the general method. This way, if the SAT instance is satisfiable, it indicates that there is a primary input assignment where the function we are considering evaluates to $\perp$.

then a copy of this new dependency graph is sent recursively to the "BreakDown" routine as a potential new best solution. The "BreakDown" routine returns when it reaches a combinational solution. The smallest dependency graph is returned to the "Synthesis" routine and the algorithm terminates.

```
BreakDown(Functions,DepGraph,SupportSetList):
  if DepGraphIsCombinational(DepGraph) then
    return DepGraph
  else
    for i = 0 to |Functions| do
      if FunctionIsNotCombinational(Functions_i, DepGraph) then
        DepGraphCopy ⇐ DepGraph
        DepGraphCopy_i ⇐ NextSmallestSupportSet(Functions_i, SupportSetsList)
        if SupportSetSize(DepGraphCopy) < SupportSetSize(SmallestDepGraph) then
          DepGraphCopy ⇐ BreakDown(Functions, DepGraphCopy, SupportSetsList)
          if SupportSetSize(DepGraphCopy) < SupportSetSize(SmallestDepGraph) then
            SmallestDepGraph ⇐ DepGraphCopy
          end if
        end if
      end if
    end for
    return SmallestDepGraph
  end if

Synthesis(Functions):
  SupportSetsList ⇐ ComputeSupportSets(Functions)
  SupportSetSize(SmallestDepGraph) ⇐ ∞
  for i = 1 to |Functions| do
    DepGraph_i ⇐ SmallestSupportSet(Functions_i, SupportSetsList)
  end for
  return BreakDown(Functions, DepGraph, SupportSetsList)
```

Fig. 23. Pseudocode for our synthesis algorithm. Magnitude symbols ($|magnitude|$) are used to indicate the size of a list. The subscript i, when applied to a list, indicates an access to the i-th element of the list. The dependency graph variables (e.g., DepGraph, DepGraphCopy, and Smallest Depgraph) are lists of support sets for each function. The routine "SmallestSupportSet" returns the smallest support set for a particular function from a list of support sets. The routine "NextSmallestSupportSet" returns the next smallest support set from a list of support sets for a particular function. The routine "SupportSetSize" returns the sum of the size of all the support sets for a given dependency graph. The routine "DepGraphIsCombinational" performs the SAT-based analysis described in the previous section; it returns *True* if the dependency graph is combinational. The routine "FunctionIsNotCombinational" returns *True* if there is a primary input assignment that causes the given function to evaluate to $\perp$.

Given a list of possible support sets, the search begins with the smallest support set for each function. This is the most compact representation possible. In practice, the initial solution is usually a very dense ball of dependencies. This initial solution is almost always not combinational. Generally, as the support sets increase in size, there are fewer cycles. The algorithm always terminates, because it must eventually hit a solution containing only the primary inputs in the supports sets for each function. Of course, in practice it likely finds much better solutions than this and terminates before this point.

A visual illustration of the synthesis algorithm is shown in Figure 24. In this example there are three functions, f_0, f_1, and f_2, and four primary input variables a, b, c, and d. In the initial dependency graph, there are primary input assignments that cause all three functions to evaluate $\perp$. The algorithm proceeds to search for solutions

by trying different support sets for all three functions. In this example, three combinational solutions are found. The smallest combinational solution has two cycles and a total support set size of 8.

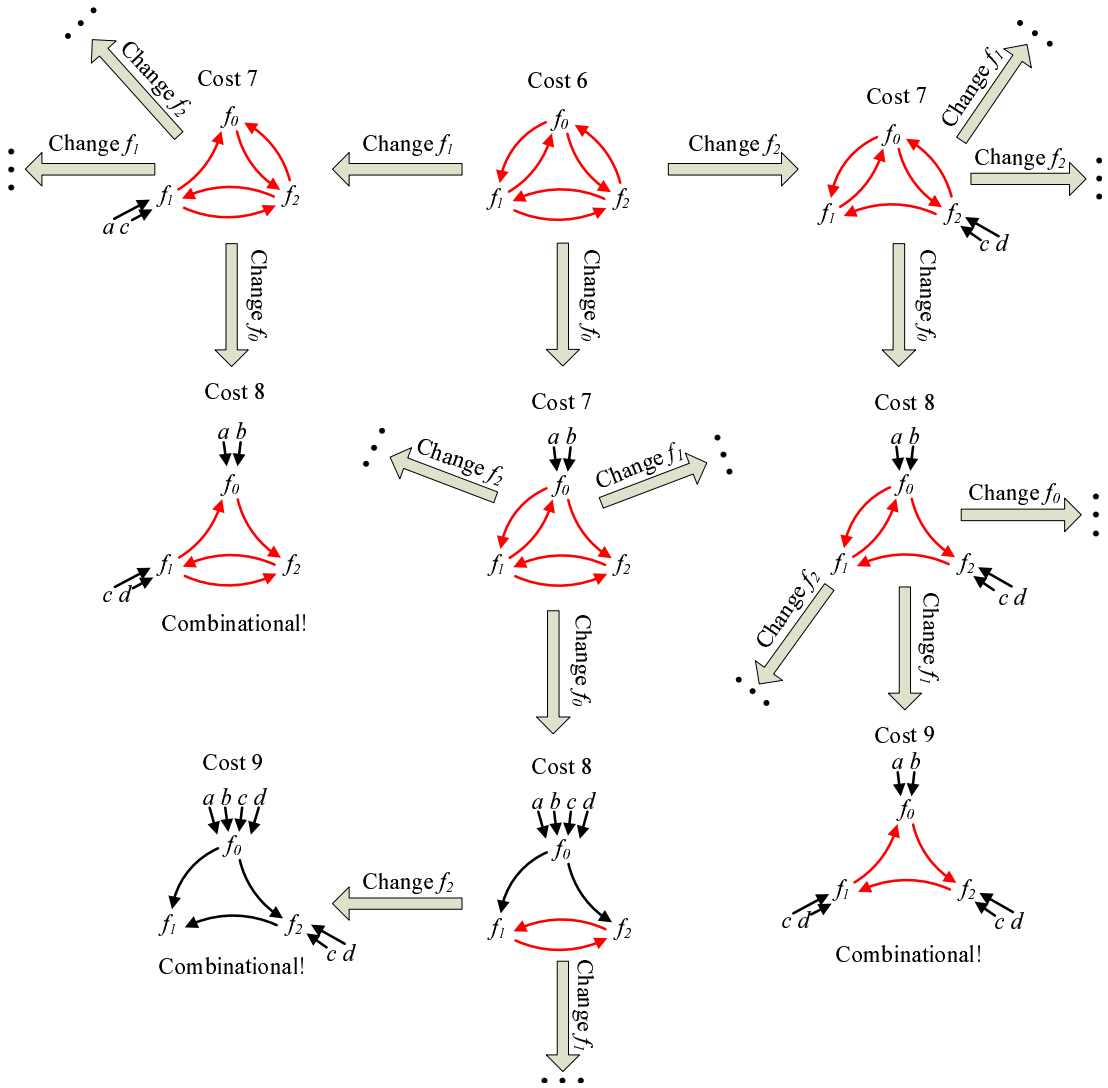


Fig. 24. An illustration of the synthesis algorithm on an example consisting of 3 functions and 4 primary input variables. Red arrows indicate cyclic dependencies in the dependency graphs. Some branches are ommited for clarity, as indicated by "...".

6. IMPLEMENTATION AND RESULTS

We present two sets of synthesis results on standard benchmarks [Benchmarks from the 2005 Int'l Workshop on Logic Synthesis]. In Table I we report results for cyclic circuits that were first synthesized with our tool CYCLIFY and then optimized using the Berkeley tool ABC [Mishchenko et al. 2007]. CYCLIFY is based on an earlier tool,

Table I. Results of circuits synthesized with CYCLIFY and then optimized with ABC.

CYCLIFY Results							
Benchmark	Gates Cyclic	Gates Acyclic	Delay Cyclic	Delay Acyclic	Size Ratio	Delay Ratio	Synthesis Time (s)
bbsse	90	96	5	8	0.94	0.63	8
bw	110	183	9	9	0.6	1	941
clip	113	181	5	9	0.62	0.56	1
cse	128	152	6	9	0.84	0.67	5
duke2	309	301	11	11	1.03	1	178
ex1	205	210	14	8	0.98	1.75	551
ex6	61	116	8	7	0.53	1.14	6
inc	87	115	6	8	0.76	0.75	4
planet	381	419	7	9	0.91	0.78	10667
planet1	377	433	7	9	0.87	0.78	18559
pma	167	161	5	8	1.03	0.63	270
s1	254	339	6	11	0.75	0.55	214
s298	1806	1823	7	14	0.99	0.50	41679
s386	91	102	5	7	0.89	0.71	8
s510	189	199	5	9	0.95	0.56	5
s526	129	135	9	9	0.96	1	25
s526n	130	117	8	10	1.11	0.80	29
s1488	431	500	9	9	0.86	1	2793
sse	87	102	5	8	0.85	0.63	10
styr	344	380	8	10	0.91	0.80	204
table5	686	639	8	13	1.07	0.62	51010

Berkeley SIS [Sentovich et al. 1992], and so uses SOPs and BDDs as the underlying data structures. Accordingly, the size of the benchmarks that it can tackle is limited. CYCLIFY uses a similar branch-and-bound algorithm to the one described in Section 5. (Instead of support set size, it uses literal counts as its cost function.) For Table I, we selected benchmarks where CYCLIFY produced cyclic solutions. Before reading these circuits into ABC, dummy primary inputs were introduced at the feedback locations (implicitly removing the cycles). The circuits were then run through 10 iterations of `compress2`, a very aggressive optimization script. The original acyclic versions of the circuit were also run through 10 iterations of `compress2`.

The "Gates" columns report the number of AND2 gates in ABC's AND-inverter graph (AIG) representation. AIGs are the standard representation at the technology-independent level for most modern synthesis algorithms, including those based on SAT. The "Size Ratio" column is calculated as "Gates Cyclic / Gates Acyclic." The "Synthesis Time" is the time it took CYCLIFY to produce the circuits.

The "Delay" columns report the delay for the cyclic and acyclic circuits. We assume that nodes in the AIG (corresponding to AND gates) have unit delay; edges in the AIG, including those with inversions, have zero delay. The "Delay Ratio" column is calculated as "Delay Cyclic / Delay Acyclic." For the cyclic circuits, we use the algorithm presented in [Riedel 2004], based on symbolic event propagation, to compute the delay. For the acyclic circuits, we compute the delay as the longest path from the primary outputs to the primary inputs in the AIG. As Table I demonstrates, introducing cyclic dependencies yields significant reductions in area as well as delay.[4]

Table II presents synthesis results from SAT-based trials, using support set size as the cost metric. The algorithm described in Figure 23 was implemented in Berkeley

[4]Although counterintuitive, cycles can be used to optimize circuits for delay as well as for area. The extra flexibility of allowing cycles when structuring functional dependencies makes it possible to move logic off of true critical paths, reducing the delay [Riedel 2004].

Table II. Benchmark circuits with cyclic dependencies.

Synthesis Results							
Benchmark	Num PIs	Num POs	Orig AIG Size	Num Cycles	Acyclic SS Size	Cyclic SS Size	Synthesis Time (s)
amd	14	25	1625	7	69	69	2
apex3	54	50	1655	1	29	27	19
duke2	22	29	577	4	57	55	10
ex6	8	25	88	1	32	32	< 1
gary	15	11	821	1	33	32	1

ABC [Mishchenko et al. 2007]. The SAT solver used was MiniSAT [Sörensson et al.]. All the trials were run on a 32-bit Linux machine with 3.2 GHz AMD Phenom(tm) II X6 1090T Processor. Only one core was utilized for running the algorithm.

Table II lists benchmarks that were run through the synthesis routine described in Section 5. The algorithm generated support sets for each of the benchmarks with primary output functions expressed in terms of other primary output functions and primary inputs. (For benchmarks that had less than 40 primary outputs, additional primary outputs were added to intermediate nodes until the benchmark contained exactly 40. This was done to increase the number of possible dependency graphs.) We ran the *BreakDown* procedure described in Section 5 until either 40 combinational solutions were found, or until a total of 200 dependency graphs were explored and none of these were deemed to be combinational. Table II reports results for the smallest cyclic and acyclic representations that were found.

The columns "Num PIs" and "Num POs" list the number of primary inputs and primary outputs, respectively. The column "Orig AIG Size" lists the number of nodes in the AIG representation. The column "Cyclic SS Size" lists the sum of the number of support variables in functions that are part of strongly-connected components in cyclic solutions. The column "Acyclic SS Size" lists the sum of the number of support variables in these same functions in the acyclic solutions. The column "Num Cycles" lists the number of cycles in the corresponding dependency graph. The column "Synthesis Time" lists the time spent searching through the space of dependency graphs and checking if solutions were combinational. In all trials, the size of all support sets was limited to 100. For most of the benchmarks, the smallest combinational solution was found relatively quickly when searching through the space of possible dependency graphs. As anyone familiar with SAT-based methods might have expected, SAT-based synthesis is very efficient.

7. DISCUSSION

Early work suggested the possible benefits of cyclic designs, and yet still, combinational circuits are not designed with cycles in practice. As early as 1992, Leon Stok predicted that EDA tools would not readily be coaxed into accepting cyclic circuits [Stok 1992]. Many of the analysis and verification routines in modern EDA tools balk when given cyclic designs. (Some check a design compulsively after every transformation to see if it contains cycles. If it does, the program screeches to a halt.) Significantly, engines for static timing analysis demand acyclic circuit topologies.

The requisite algorithmic approach is to perform "false-path" aware analysis. Early formulations based on SOPs and BDDs were never up to the task, but modern SAT-based algorithms are powerful enough to perform such analysis. In our view, the analysis engines of modern EDA tools should be made not only "false-path" aware but also "false-cycle" aware. Introducing cycles provides significant opportunities for optimization, both for area and for delay. (Since power is generally correlated with area, we expect gains in this metric as well.)

In related work, we have described SAT-based algorithms for gate-level analysis and mapping of cyclic circuits [Backes et al. 2008; Backes and Riedel 2011]. This paper presented a SAT-based method for synthesizing cyclic functional dependencies, at a technology-independent level. It is an application of a very promising new idea for synthesizing functional dependencies with Craig interpolation [Lee et al. 2007].

The topic structuring functional dependencies, whether cyclic or acyclic, is one that has not garnered sufficient attention in the logic synthesis community, in our opinion. Given the remarkable scalability of the approach, Craig interpolation provides the opportunity to explore large changes in the structure of functional dependencies, early in the synthesis process. In applications to date, interpolants have been generated directly from the proofs of unsatisfiability that are provided by SAT solvers. We have proposed efficient methods based on incremental SAT solving for modifying resolution proofs in order to obtain more compact interpolants. This reduces the cost of the logic that is generated for functional dependencies [Backes and Riedel 2010].

In future work, we will study techniques for manipulating and minimizing the resolution proofs obtained through incremental SAT calls, with the aim of effecting large optimizations in circuit structure through changes in functional dependencies. In our view, the resolution proofs from SAT solving could be used as an underlying data structure for performing technology-independent synthesis, as opposed to just the front-end step.

REFERENCES

AMLA, N., DU, X., KUEHLMANN, A., KURSHAN, R., AND MCMILLAN, K. 2005. An analysis of SAT-based model checking techniques in an industrial environment. *Correct Hardware Design and Verification Methods*, 254–268.

BACKES, J., FETT, B., AND RIEDEL, M. D. 2008. The analysis of cyclic circuits with Boolean satisfiability. In *International Conference on Computer-Aided Design*. 143–148.

BACKES, J. AND RIEDEL, M. D. 2010. Reduction of interpolants for logic synthesis. In *International Conference on Computer-Aided Design*.

BACKES, J. AND RIEDEL, M. D. 2011. The analysis and mapping of cyclic circuits with boolean satisfiability. *submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

BENCHMARKS FROM THE 2005 INT'L WORKSHOP ON LOGIC SYNTHESIS. Available at http://iwls.org/iwls2005/benchmarks.html.

BRAYTON, R. K., HACHTEL, G. D., MCMULLEN, C. T., AND SANGIOVANNI-VINCENTELLI, A. L. 1990. Multilevel logic synthesis. *Proceedings of the IEEE 78,* 2, 264–300.

BRYANT, R. E. 1987. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6,* 4, 634–649.

BRZOZOWSKI, J. AND SEGER, C.-J. 1995. *Asynchronous Circuits*. Springer-Verlag.

EDWARDS, S. A. 2003. Making cyclic circuits acyclic. In *Design Automation Conference*. 159–162.

EÉN, N. AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *SAT*, E. Giunchiglia and A. Tacchella, Eds. Lecture Notes in Computer Science Series, vol. 2919. Springer, 502–518.

KATZ, R. 1992. *Contemporary Logic Design*. Benjamin/Cummings.

KUKIMOTO, Y. AND BRAYTON, R. 1997. Exact required time analysis via false path detection. In *Design Automation Conference*. 220–225.

LARRABEE, T. 1992. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11,* 1, 4–15.

LEE, C.-C., JIANG, J.-H. R., HUANG, C.-Y., AND MISHCHENKO, A. 2007. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *International Conference on Computer-Aided Design*. 227–233.

MALIK, S. 1994. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13,* 7, 950–956.

MCMILLAN, K. L. 2003. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*. 1–13.

MISHCHENKO, A. ET AL. 2007. ABC: A system for sequential synthesis and verification.

NEIROUKH, O., EDWARDS, S. A., AND XIAOYU, S. 2008. Transforming cyclic circuits into acyclic equivalents. *IEEE Transactions on Computer-Aided Design 27*, 17750–1787.

RIEDEL, M. D. 2004. Cyclic combinational circuits. Ph.D. thesis, Caltech.

RIEDEL, M. D. AND BRUCK, J. 2003. The synthesis of cyclic combinational circuits. In *Design Automation Conference*. 163–168.

RIVEST, R. L. 1977. The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers 26,* 6, 606–607.

SENTOVICH, E. M., SINGH, K. J., LAVAGNO, L., MOON, C., MURGAI, R., SALDANHA, A., SAVOJ, H., STEPHAN, P. R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1992. SIS: A system for sequential circuit synthesis. Tech. rep., University of California, Berkeley.

SHIPLE, T. 1996. Formal analysis of synchronous circuits. Ph.D. thesis, U.C. Berkeley.

SÖRENSSON, N. ET AL. Minisat v1.13 – a SAT solver with conflict-clause minimization available at http://minisat.se/downloads/.

STOK, L. 1992. False loops through resource sharing. In *International Conference on Computer-Aided Design*. 345–348.

WAKERLY, J. F. 2000. *Digital Design: Principles and Practices*. Prentice-Hall.

YOELI, M. AND RINON, S. 1964. Application of ternary algebra to the study of static hazards. *Journal of ACM 11,* 1, 84–97.