# The Synthesis of Cyclic Dependencies with Boolean Satisfiability

JOHN D. BACKES and MARC D. RIEDEL, University of Minnesota

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 article 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

### 1.1. Cyclic Combinational Circuits

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

And yet, cyclic circuits can be combinational. Consider the truth table of values and the functions shown in Figure 1. The definition of these functions is cyclic. In spite of
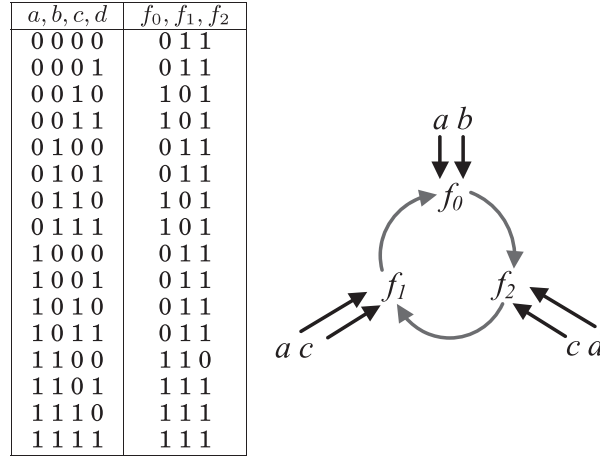
**44**

| $a, b, c, d$ | $f_0, f_1, f_2$ |
|---|---|
| 0 0 0 0 | 0 1 1 |
| 0 0 0 1 | 0 1 1 |
| 0 0 1 0 | 1 0 1 |
| 0 0 1 1 | 1 0 1 |
| 0 1 0 0 | 0 1 1 |
| 0 1 0 1 | 0 1 1 |
| 0 1 1 0 | 1 0 1 |
| 0 1 1 1 | 1 0 1 |
| 1 0 0 0 | 0 1 1 |
| 1 0 0 1 | 0 1 1 |
| 1 0 1 0 | 0 1 1 |
| 1 0 1 1 | 0 1 1 |
| 1 1 0 0 | 1 1 0 |
| 1 1 0 1 | 1 1 1 |
| 1 1 1 0 | 1 1 1 |
| 1 1 1 1 | 1 1 1 |



$$f_0 = ab + \bar{f}_1$$
$$f_1 = \bar{c} + f_2 a$$
$$f_2 = c + d + \bar{f}_0$$

Fig. 1.   Example: A cyclic circuit with 4 primary inputs and 3 primary outputs.



$$f_0 = \bar{f}_1 = 0$$
$$f_1 = f_2 = 1$$
$$f_2 = \phantom{f_2 = } 1$$
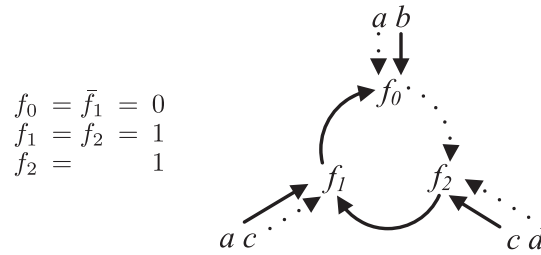
Fig. 2.   Network in Figure 1 with $a = 1, b = 0, c = 1, d = 0$.

this, the network is combinational: it produces the correct outputs, regardless of the initial state and independently of all timing assumptions. To see this, consider specific input values. For instance, with $a = 1, b = 0, c = 1, d = 0$, the network simplifies to that shown in Figure 2, yielding the correct values for $f_0$, $f_1$ and $f_2$. With $a = 1, b = 1, c = 0, d = 0$, the network simplifies to that shown in Figure 3, again yielding the correct values for $f_0$, $f_1$, and $f_2$. The reader may verify that the network implements the correct output values for all input values.

In previous work, we showed that combinational circuits can be optimized significantly if cycles are introduced [Riedel and Bruck 2003]. The intuition behind this is that, with feedback, all nodes can potentially benefit from work done elsewhere; without feedback, nodes at the top of the hierarchy must be constructed from scratch. We proposed a methodology for synthesizing such circuits and demonstrated that it produces significant improvements in area and in delay. Cycles are introduced in the restructuring and minimization phases of logic synthesis at the level of functional dependencies.

$$
\begin{aligned}
f_0 &= & 1 \\
f_1 &= & 1 \\
f_2 &= \bar{f}_0 = & 0
\end{aligned}
$$

Fig. 3. Network in Figure 1 with $a = 1, b = 1, c = 0, d = 0$.

## 1.2. Prior and Related Work

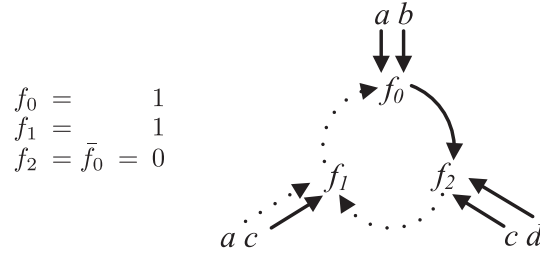Leon Stok lamented that EDA tools were rejecting cyclic designs because there was no way to validate them [Stok 1992]. In response, Malik discussed analysis techniques for cyclic combinational circuits [Malik 1994]. His approach was topological, beginning with a transformation from a cyclic specification 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 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.

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 [Riedel and Bruck 2004]. 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.

## 1.3. SAT-Based Synthesis

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]. Significantly, SAT-based algorithms lend themselves well to *incremental analysis*. Often analysis and verification tasks
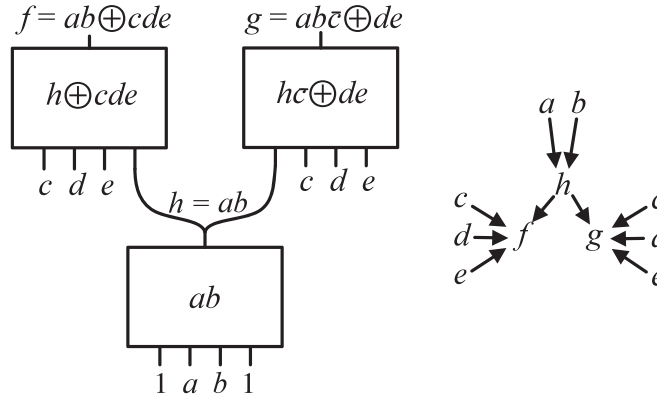
Fig. 4. Three four-input lookup tables implement functions $f = ab \oplus cde$ and $g = ab\bar{c} \oplus de$ using an acyclic topology. The circuit's dependency graph is shown on the right.

are applied iteratively and incrementally in a design flow: small changes are made to improve the circuit and then it is reanalyzed. 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].

In recent work, we have proposed an efficient SAT-based algorithm for analyzing and mapping cyclic circuits [Backes et al. 2008, 2011]. We perform SAT-based valida-tion 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 assign-ments; these assignments are used to modify the mapping in order to ensure that the circuit remains combinational.

This article tackles the problem of synthesizing cyclic combinational circuits with SAT-based techniques. Specifically, we apply a SAT-based technique called Craig in-terpolation for synthesizing functional dependencies [Lee et al. 2007]. This technique is geared towards technologies where the complexity of implementing a function is heavily dependent on the number of support variables.

This is illustrated conceptually in Figure 4. The figure shows three functions: $f(h, c, d, e)$, $g(h, c, d, e)$, and $h(a, b)$. Both $f$ and $g$ can be represented in terms of the support variables $a, b, c, d,$ and $e$. However, if $f$ and $g$ are to be implemented in an acyclic topology in terms of four input lookup tables, at least one additional lookup table must be used (in this case $h(a, b)$).

Whether or not a function can be represented in terms of certain support variables can be cast as a SAT problem. If the answer is affirmative, Craig interpolation pro-vides an implementation [Lee et al. 2007]. Figure 5 demonstrates that an alternative representation exists for $f$, and $g$. Craig interpolation can be used to generate the functions $f(a, b, c, g)$ and $g(f, c, d, e)$, and a SAT solver can verify whether or not this representation behaves combinationally.

## 1.4. Organization

This article is organized as follows. Section 2 provides definitions and describes the notation used throughout the work. Section 3 discusses the underlying circuit and network models. Section 4 presents the core contribution of the article: 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. Fi-nally, Section 7 discusses future directions of research.
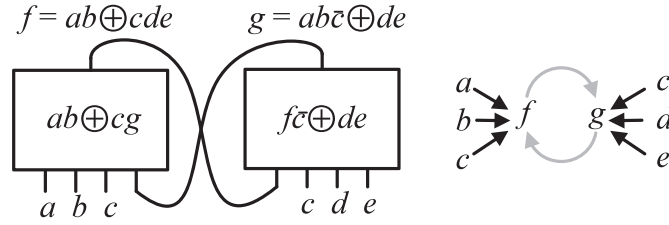
Fig. 5. Two four-input lookup tables implement functions $f = ab \oplus cde$ and $g = ab\bar{c} \oplus de$ using a cyclic topology. The circuit's dependency graph is shown on the right.

## 2. DEFINITIONS AND NOTATION

We use the standard notation: addition (+) denotes disjunction (OR), multiplication (·), 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 refereed 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, that is, 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].
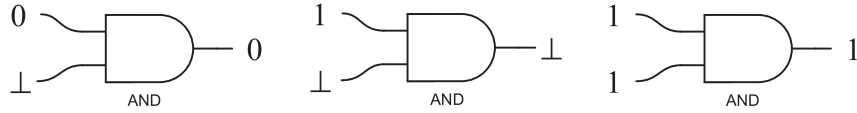
Fig. 6.   An AND gate with 0, 1, and $\perp$ inputs.
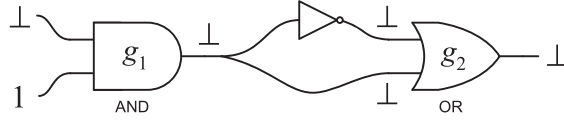


Fig. 7.   An illustration unknown/undefined values $\perp$.

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 6. Input values that determine the gate output are called *controlling*.

Consider the circuit fragment in Figure 7. 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 $\perp$[1].

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 the following example.

*Example* 3.1. Consider the circuit shown in Figure 8, 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 9. The outcome for all eight cases is shown in Figure 10. We conclude that the circuit is not combinational.

--------

[1]In standard CMOS technologies, it is possible for a gate to output a voltage value between the noise margin if its inputs are also somewhere between logical 0 and logical 1. Remaining agnostic about the value of $g_2$ in Figure 7 allows us to invalidate circuits where this could be a concern.
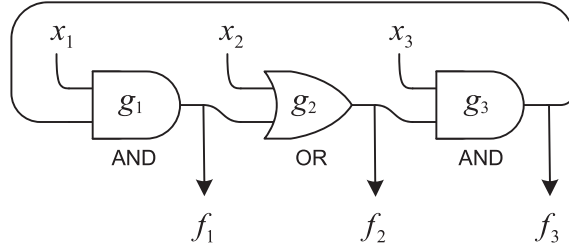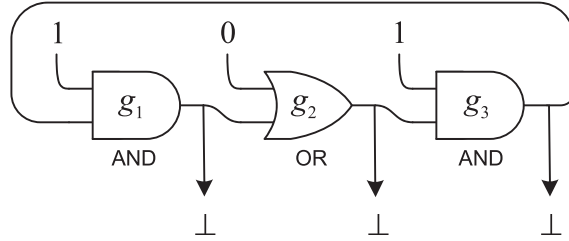
Fig. 8. A cyclic circuit that is not combinational.



Fig. 9. The circuit of Figure 8 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 | $\bot$ | $\bot$ | $\bot$ |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 |

Fig. 10. Analysis of the circuit in Figure 8.

### 3.1. Gate-Level vs. Functional-Level Analysis

The algorithms and concepts presented in this article 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. This was first demonstrated in Jiang et al. [2004].

Figure 11 demonstrates an example of a function that may behave differently depending on its gate-level mapping. Before the function $f(a, b, c) = ab + c\bar{b}$ is mapped to gates, $f(1, b, 1) = b + \bar{b} = 1$. However, the axiom $b + \bar{b} \equiv 1$ only holds if it is assumed that the values on the wires are truly Boolean (as demonstrated in Figure 7). In the case where $b = \bot$, it is possible that $b$ is some value between 1 and 0, and in this case the mapped circuit shown in Figure 11 will evaluate to a different value than the unmapped function $f(a, b, c)$.
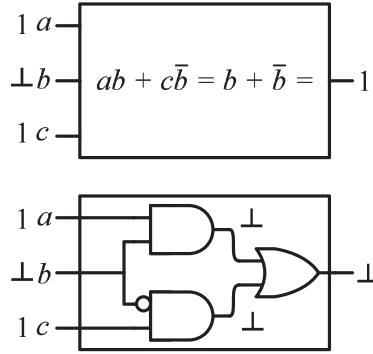
Fig. 11. The function $ab + c\bar{b}$ and a gate-level implementation.

An assignment of a subset of a function's support variables is said to be a *controlling assignment* if the function evaluates to the same value regardless of the assignment of the other variables in the function's support set. We sometimes say that a variable assignment *controls* a function, if that variable assignment is a controlling assignment for that function.

In this work, analysis is performed on the level of Boolean functions. We assume that a function evaluates to definite values for all controlling assignments to that function's support variables. In Backes et al. [2011], we explore methods of mapping and analyzing cyclic circuits at the level of gates. In that paper, we prove that any set of cyclic functions that is deemed combinational can be mapped to a gate-level design. We provide a constructive method for performing the mapping.

## 4. FUNCTIONAL DEPENDENCIES

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* $y_i$, $0 \le i \le n - 1$. (We sometimes abuse the notation by using the same name for the function and the corresponding internal variable, 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 reexpressed 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 article, 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 12 shows four different expressions for the functions and the corresponding dependency graphs. Figure 12(a) shows expressions for $f_1$ and $f_2$, both in terms of the

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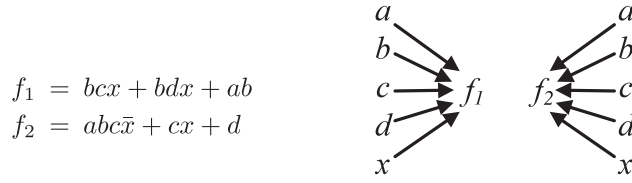


(a) $f_1(a, b, c, d, x)$ and $f_2(a, b, c, d, x)$

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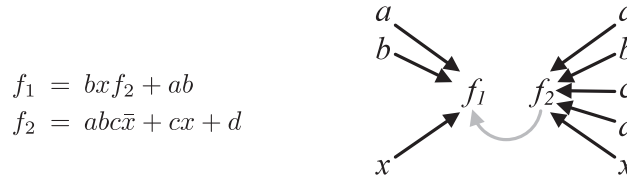


(b) $f_1(a, b, x, f_2)$ and $f_2(a, b, c, d, x)$

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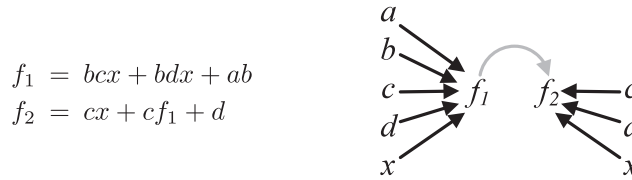


(c) $f_1(a, b, c, d, x)$ and $f_2(c, d, x, f_1)$

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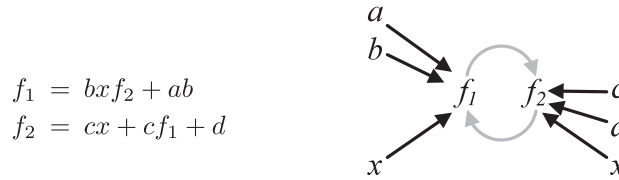


(d) $f_1(a, b, x, f_2)$ and $f_2(c, d, x, f_1)$

Fig. 12. Four different implementations of two functions, $f_1$ and $f_2$, of five variables $a$, $b$, $c$, $d$, and $x$.

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 12(b) and 12(c) show alternate expressions, obtained by introducing functional dependencies. In Figure 12(b), $f_1$ is expressed in terms of $f_2$ and $\{a, b, x\}$. Accordingly, the total cost is 9. In Figure 12(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 $f_2 \sqsubseteq f_1$ would produce the expressions in Figure 12(b); the ordering of $f_1 \sqsubseteq f_2$ would produce the expressions in Figure 12(c). Insisting upon an ordering means that we would have to choose one of these two results.
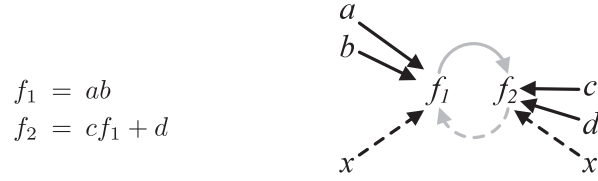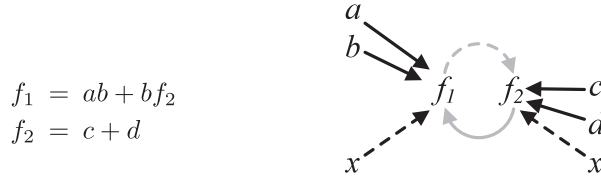
$$f_1 = ab$$
$$f_2 = cf_1 + d$$

(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. 13. 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.

However, if we allow cyclic dependencies, we can find a better solution. Figure 12(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 12(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 13. We see that, for any assignment of $x$, the cyclic dependency between $f_1$ and $f_2$ is broken, so the result is combinational.

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 article 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.
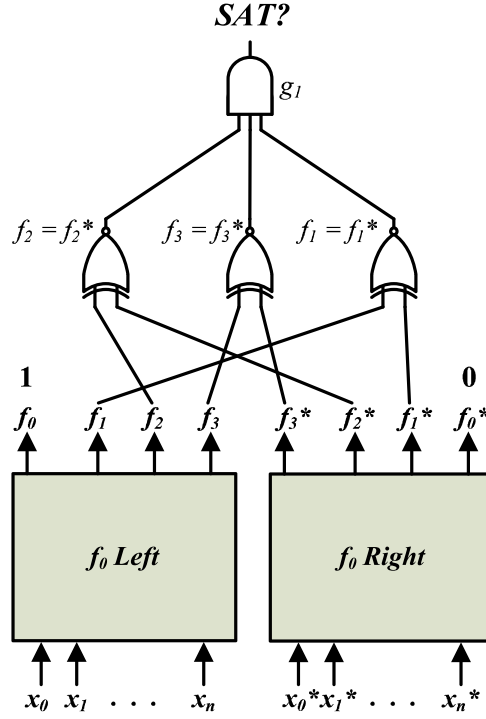
Fig. 14.   A miter that checks to see if $f_0$ can be specified in terms of $f_1$, $f_2$, and $f_3$.

The method constructs a miter, as shown Figure 14. 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 satisfiable, then there exists a pair of input assignments $X$ and $X^*$ such that $f_0(X) \neq f_0^*(X^*)$ and $f_1(X) = f_1^*(X^*)$, $f_2(X) = f_2^*(X^*)$, $f_3(X) = f_3^*(X^*)$. Thus the value of $f_0$ cannot be determined solely from the values of $f_1$, $f_2$, and $f_3$.

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

| $a\,b\,f_1$ | $f_0$ |
|---|---|
| 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 |



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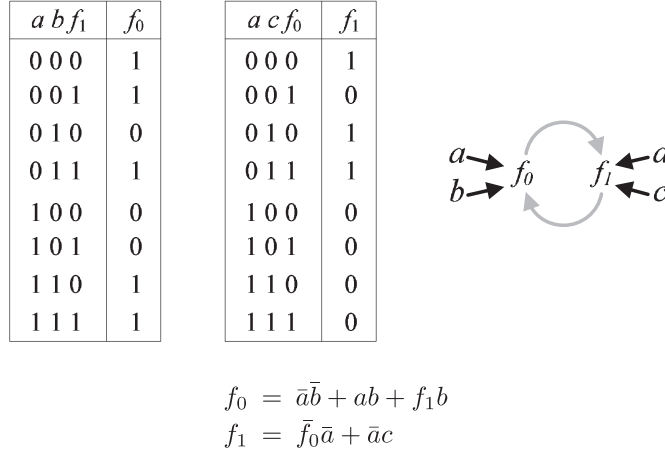$$f_1 = \bar{f}_0 \bar{a} + \bar{a}c$$

Fig. 15. The truth tables for two functions. The cyclic dependency graph containing these two functions is not combinational.

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 14.

Using Craig interpolation to generate functional dependencies has proven much more scalable than the previous SOP- and BDD-based methods. However, the structure of the dependencies that are generated are often overly large and redundant. For this reason, Craig interpolation is generally used for architectures based on lookup tables (e.g., FPGAs) where no matter how complex a function is, it can be implemented by a lookup table as long as its support set size is less than or equal to the size of the lookup table.

## 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 15 shows the truth tables for two functions $f_0$ and $f_1$. In this implementation, $f_0$ has support variables $a, b$, and $f_1$, while $f_1$ has support variables $a, c$, and $f_0$. 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 16 shows the functions $f_0$ and $f_1$ and the resulting dependency graph under this assignment.

Informally, a cyclic dependency graph is not combinational if there exists a selection of pairs of rows from the truth tables for the functions that satisfies three conditions.

(1) The primary input variables are the same value in every row.
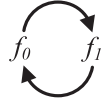
$$f_0 = f_1 \qquad\qquad (3)$$
$$f_1 = \bar{f}_0 \qquad\qquad (4)$$

Fig. 16. The dependency graph for the functions in Figure 15 for the assignment: $a = c = 0, b = 1$. The dependency graph is not combinational.



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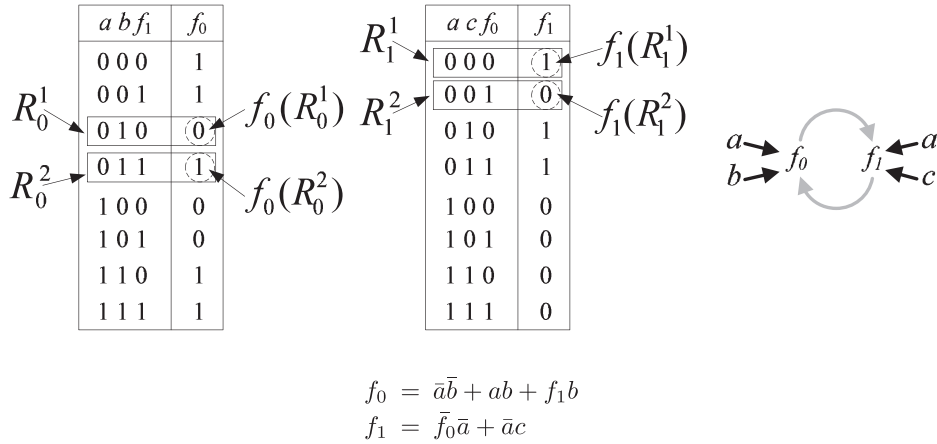$$f_1 = \bar{f}_0 \bar{a} + \bar{a}c$$

Fig. 17. The truth tables for two functions. The cyclic dependency graph containing these two functions is not combinational. This figure also illustrates a specific selection of rows that proves that the cyclic dependency graph is not combinational.

(2) If the value of some function is the same for some pair of rows, then the variable corresponding to this function in every other pair of rows assumes this value (i.e., if the value of some function is controlled, then this value propagates to the input of other functions that contain this function as a support variable).
(3) The values of some function toggles between 1 and 0 for some pair.

Finding a selection of pairs of rows that holds these properties is necessary and sufficient to show that there is a primary input assignment that causes the circuit to reach a fixed point with a $\perp$ value. This is stated more formally with Proposition 4.1. In Proposition 4.1, we consider a function's truth table to be a set of rows. Each row is an assignment of the function's support variables (which can contain primary input variables, or other internal variables). Each function has a value associated with a row. For example, consider the truth table for function $f_0$ in Figure 15. Let $r_0$ be the first row of this truth table. The variable assignment associated with $r_0$ is $a = b = f_1 = 0$. The value of $f_0(r_0)$ is 1. It may help the reader to refer to Figure 17 and to the example listed after the proof to help make sense of the constructs in Proposition 4.1.

PROPOSITION 4.1. *Let G be a cyclic dependency graph and let $T = \{t_0, t_1, \ldots, t_{n-1}\}$ be the set of truth tables for the functions $F = \{f_0, f_1, \ldots, f_{n-1}\}$ in G. Let $R^1 = \{r_0 \in t_0, r_1 \in t_1, \ldots, r_{n-1} \in t_{n-1}\}$ and $R^2 = \{r_0 \in t_0, r_1 \in t_1, \ldots, r_{n-1} \in t_{n-1}\}$ be sets of rows from the truth tables in T. G is not combinational if and only if, for some choice of $R^1$ and $R^2$ (some selection of rows) the following conditions hold.*

(1) *Every row in $R^1$ and $R^2$ has the same values for its primary input variables.*
(2) *Let $R_i^1$ and $R_i^2$ be the ith row in $R^1$ and $R^2$ respectively. $\forall i \in \{0, 1, \ldots, n-1\}$. If the value of $f_i(R_i^1)$ is the same as $f_i(R_i^2)$ then $f_i$ is this value in every other row in $R^1$ and $R^2$. This only for rows that contain $f_i$ as a support variable.*
(3) *$\exists i \in \{0, 1, \ldots, n-1\}$ such that the value of $f_i(R_i^1)$ differs from $f_i(R_i^2)$.*

PROOF. The first two conditions force the choice of $R^1$ and $R^2$ to correspond to a fixed point in $G$ reached by some primary input assignment.

The first condition asserts that the assignment of the primary input variables must be the same in every row of every element of $R^1$ and $R^2$. If the primary input assignment is a controlling assignment for some function $f_i$, then that function's output value will not differ between the two rows $R_i^1$ and $R_i^2$.

The second condition asserts that if the output value of some function $f_i$ is the same between two rows $R_i^1$ and $R_i^2$, then the variable corresponding to this function in other rows of $R^1$ and $R^2$ 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 $f_i$ differs between two rows $R_i^1$ and $R_i^2$, 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 $R^1$ and $R^2$ 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 $R^1$ and $R^2$ for a network that is combinational.  □

Because the example in Figure 15 is not combinational, there must be some choice of pairs of rows ($R^1$ and $R^2$) that satisfies the three conditions in Proposition 4.1. As stated before, the conditions can be satisfied by selecting the third and fourth rows of the truth table for $f_0$ and the first and second rows of the truth table for $f_1$: $R^1 = \{\{a = 0, b = 1, f_1 = 0\}, \{a = 0, c = 0, f_0 = 0\}\}$ and $R^2 = \{\{a = 0, b = 1, f_1 = 1\}, \{a = 0, c = 0, f_0 = 1\}\}$. The first condition is satisfied because $a = c = 0, b = 1$ for every element of $R^1$ and $R^2$. The second condition is satisfied because $f_0(R_0^1) \neq f_0(R_0^2)$ and $f_1(R_1^1) \neq f_1(R_1^2)$ (i.e., $f_0(0, 1, 0) \neq f_0(0, 1, 1)$ and $f_1(0, 0, 0) \neq f_1(0, 0, 1)$). Finally, both functions $f_0$ and $f_1$ are toggling for this primary input assignment ($f_0(R_0^1) \neq f_0(R_0^2)$ and $f_1(R_1^1) \neq f_1(R_1^2)$), satisfying the third condition. Figure 17 illustrates this specific selection of rows for the example in Figure 15.

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 aforesaid conditions are met for the functions in Figure 15 is shown in Figure 18.
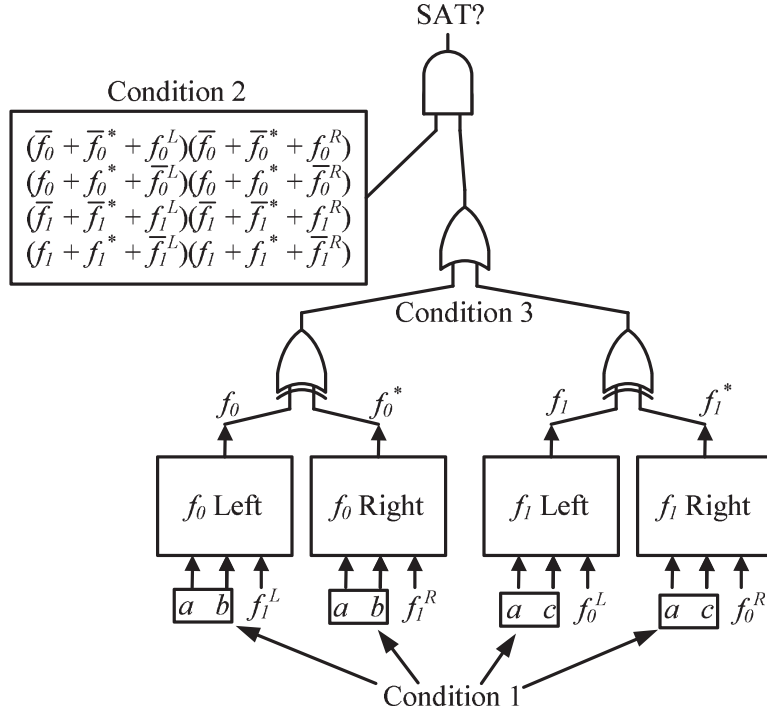
Fig. 18. A SAT instance that verifies whether or not the functions described in Figure 15 are combinational.

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 $f_1(a, c, f_0)$ is proven to be noncombinational.

### 4.3. General Method

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

(1) Generate an implementation for each target function in terms of its support variables via Craig interpolation (The same way as discussed in Section 4.1). 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 $CNF_i^L(X, F)$ and $CNF_i^R(X, F)$ 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
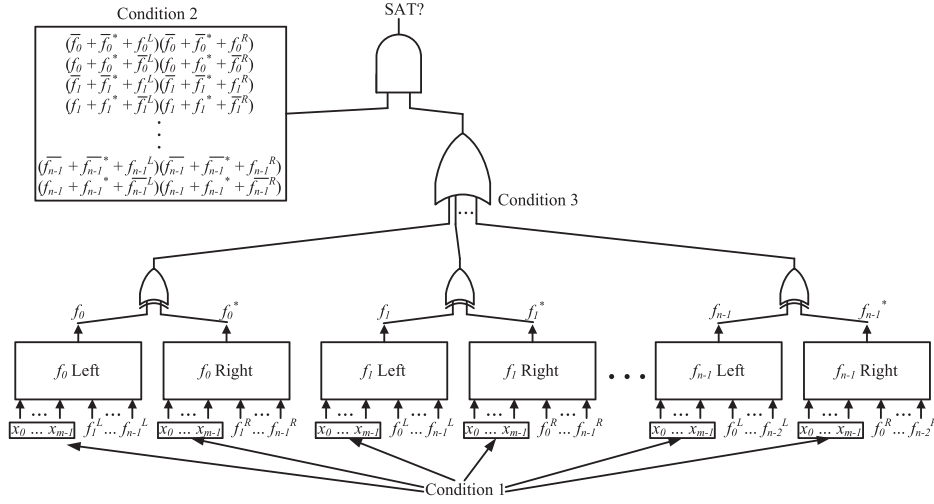
Fig. 19. A SAT instance that checks if the set of functions $F = \{f_0, f_1, \ldots, f_{n-1}\}$ of variables $X = \{x_0, x_1, \ldots, x_{m-1}\}$ is combinational.

between 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.

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

Figure 19 shows a graphical representation of the general SAT instance for $n$ functions of $m$ variables[2]. Similarly to Figure 18, the conditions stated in Proposition 4.1 are shown in this figure as well.

PROPOSITION 4.2. *Some choice of $R^1$ and $R^2$, 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 $R^1$ and $R^2$. The conditions in $c_1$ assert that the primary input variables

---

[2]This figure is drawn assuming $n > 2$.

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 Proposition 4.1 can be met for some choice of $R^1$ and $R^2$. 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 20. 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 noncombinational.[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 than the best current solution, 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.

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.

---

[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 $\bot$.

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

**Synthesis**($Functions$)**:**
  $SupportSetsList \Leftarrow ComputeSupportSets(Functions)$
  $SupportSetSize(SmallestDepGraph) \Leftarrow \infty$
  **for** $i = 1$ **to** $|Functions|$ **do**
    $DepGraph_i \Leftarrow SmallestSupportSet(Functions_i, SupportSetsList)$
  **end for**
  **return** $BreakDown(Functions, DepGraph, SupportSetsList)$

Fig. 20. 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 SmallestDepGraph) 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 $\perp$.

A visual illustration of the synthesis algorithm is shown in Figure 21. In this example there are three functions, $f_0$, $f_1$, and $f_2$, of 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.

## 5.1. Finding Support Sets

Our synthesis algorithm requires that a list of possible support sets be provided. To the best of our knowledge, there has been no research that directly deals with the problem of quickly finding possible support sets for target functions. Work on cut enumeration for FPGA mapping is somewhat related, but is heavily biased by the initial structure of a netlist [Mishchenko et al. 2007b; Takata and Matsunaga 2009]. For this work, we used a relatively simple algorithm for parsing the search space of possible support sets for a target function. The algorithm is described by the psuedocode in Figure 22. The algorithm starts with a large list of possible support variables. It recursively removes variables for this list, finding smaller support sets that can be used to represent the target function. These smaller support sets have the property that if any variable was
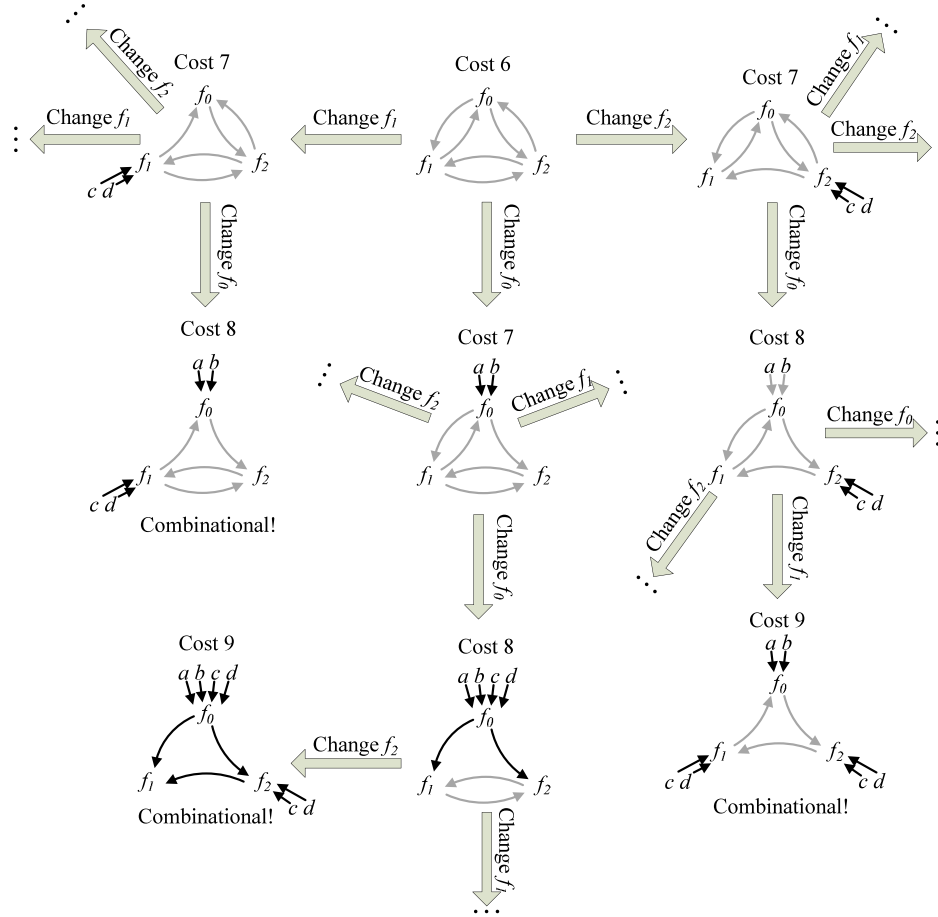
Fig. 21. An illustration of the synthesis algorithm on an example consisting of 3 functions and 4 primary input variables. The thin gray arrows indicate cyclic dependencies in the dependency graphs. Some branches are omitted for clarity, as indicated by "...".

removed, the resulting support set could not be used to represent the target function. In the experiments run in Table I of Section 6 we limited the number of possible support sets for each target function to 100. We use incremental SAT solving to improve the speed of subsequent calls to the SAT solver.

## 6. IMPLEMENTATION AND RESULTS

We present two sets of synthesis results on standard benchmarks [Benchmarks 2005]. 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. 2007a]. CYCLIFY is based on an earlier tool, 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

**SupportSetsHelper(**$Function, SupSetVars, SupSets$**):**
  **for** $Set \in SupSets$ **do**
    **if** $Set \subseteq SupSetVars$ **then**
      **return** $TRUE$
    **end if**
  **end for**
  **if** $IsNotValidSupSet(Function, SupSetVars)$ **then**
    **return** $FALSE$
  **end if**
  **for** $v \in SupSetVars$ **do**
    $PosSupSet \Leftarrow SupSetVars - \{v\}$
    **if** $SupportSetsHelper(Function, PosSupSet, SupSets)$ **then**
      **return** $TRUE$
    **end if**
  **end for**
  $SupSets \Leftarrow SupSets \cup SupSetVars$
  **return** $TRUE$

**SupportSets(**$Function$**):**
  $SupSets \Leftarrow \emptyset$
  $SupSetVars \Leftarrow PossibleSupportVars()$
  $SupportSetsHelper(Function, SupSetVars, SupSets)$
  **return** $SupSets$

Fig. 22. The two functions "SupportSets" and "SupportSetsHelper" are used to generate a list of valid support sets for a target function. The function "PossibleSupportVars" returns a list of variables that could possibly be used as a support variable for the target function. The "SupportSets" function initializes the list of support sets and the list of possible support set variables before calling the "SupportSetsHelper" function. The "SupportSetsHelper" function checks to see if the set of current variables is a superset of some already found support set. If it is not, the SAT based check discussed in Section 4.1 is performed to determine if the current set of variables can be used to represent the target function. If they can, then the function is called recursively with each variable removed once from the set of current support variables. If none of these support sets can be used to represent the target function, then this indicates that no subset of the current support set variables can be used to represent the target function. In this case, the current set of support variables is added to the list of support sets.

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. We note that these numbers reflect the size of the circuits before they are mapped to some technology, These numbers are subject to some change after mapping. This holds for the numbers reported in Table II as well.

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

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 |

Table II. Benchmark Circuits with Cyclic Dependencies

| Synthesis Results | | | | | | | |
|---|---|---|---|---|---|---|---|
| B enchmark | 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 |

dependencies yields significant reductions in area as well as delay.[4] The runtime of CYCLIFY is greatly influenced by the size of the circuit (as benchmarks `table5` and `s298` demonstrate).

Table II presents synthesis results from SAT-based trials, using support set size as the cost metric. The algorithm described in Figure 20 was implemented in Berkeley ABC [Mishchenko et al. 2007a]. The SAT solver used was MiniSAT [Sörensson and Een 2012]. 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

---

[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].

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 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.

The new SAT-based synthesis methodology scales much better with circuit size than that of CYCLIFY. However, the cost metric for the comparison is different (support set size versus AIG size). Modern FPGA mapping algorithms have a similar aim as the synthesis methodology presented in this article; they attempt to reorganize groups of functions into blocks with a fixed support set size. Currently, the state-of-the-art tools do not allow cyclic dependencies. The results presented in this work demonstrate that cyclic dependencies with smaller support set size than their acyclic equivalents can be found in benchmark circuits, and they can be found in a scalable manner. Modern synthesis algorithms, such as those targeting FPGAs, can be adapted to consider cyclic solutions using the method presented in this article, increasing the space of possible solutions that these tools can produce.

## 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, 2011]. This article presented an 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 of 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. In *Correct Hardware Design and Verification Methods*, 254–268.

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

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

BACKES, J., FETT, B., AND RIEDEL, M. D. 2011. The analysis and mapping of cyclic circuits with boolean satisfiability. In *Proceedings of the S. J. Satisf.* (to appear).

BENCHMARKS. 2005. Benchmarks from the 2005 international workshop on logic synthesis. http://iwls.org/iwls2005/benchmarks.html.

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

BRYANT, R. E. 1987. Boolean analysis of MOS circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 6,* 4, 634–649.

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

EDWARDS, S. A. 2003. Making cyclic circuits acyclic. In *Proceedings of the 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, vol. 2919. Springer, 502–518.

JIANG, J.-H. R., MISCHENKO, A., AND BRAYTON, R. K. 2004. On breakable cyclic definitions. In *Proceedings of the International Conference on Computer-Aided Design*. 411–418.

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

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

LARRABEE, T. 1992. Test pattern generation using Boolean satisfiability. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 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 *Proceedings of the International Conference on Computer-Aided Design*. 227–233.

MALIK, S. 1994. Analysis of cyclic combinational circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 13,* 7, 950–956.

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

MISHCHENKO, A. ET AL. 2007a. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/abc.htm.

MISHCHENKO, A., CHATTERJEE, S., AND BRAYTON, R. 2007b. Improvements to technology mapping for lut-based fpgas. *IEEE Trans. Comput.-Aid. Des*. 41–49.

NEIROUKH, O., EDWARDS, S. A., AND XIAOYU, S. 2008. Transforming cyclic circuits into acyclic equivalents. *IEEE Trans. Comput.-Aid. Des. 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 *Proceedings of the Design Automation Conference*. 163–168.

RIEDEL, M. D. AND BRUCK, J. 2004. Timing analysis of cyclic combinational circuits. In *Proceedings of the International Workshop on Logic and Synthesis*.

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. AND EEN, N. 2012. Minisat v1.13 – A SAT solver with conflict-clause minimization. http://minisat.se/downloads/.

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

TAKATA, T. AND MATSUNAGA, Y. 2009. An efficient cut enumeration for depth-optimum technology mapping for lut-based fpgas. In *Proceedings of the ACM Great Lakes Symposium on VLSI*. 351–356.

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. *J. ACM 11,* 1, 84–97.