The Synthesis of Cyclic Dependencies with Craig Interpolation

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

***Abstract*—The accepted wisdom is that combinational circuits must have *acyclic* (i.e., loop-free or feed-forward) topologies. And yet simple examples suggest that this need not be so. In previous work, we advocated the design of *cyclic* combinational circuits (i.e., circuits with loops or feedback paths). We proposed a 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, at the level of functional dependencies. In the original formulation, the functional dependencies were obtained through SOP minimization with the Berkeley SIS package; validation was performed with BDD-based analysis. In recent work, we presented a SAT-based technique for validation. In this paper, we present a SAT-based method for synthesizing cyclic dependencies, through Craig interpolation. While full synthesis results are not presented in this work, it is our plan to incorporate this methodology into a full synthesis routine soon.**

I. Introduction

A collection of logic gates forms a *combinational* circuit if the outputs can be described as Boolean functions of the current input values only. A common misconception is that combinational circuits must have acyclic topologies; that is to say, they must be designed without any loops or feedback paths. In fact, the idea that "combinational" and "acyclic" are synonymous terms is so thoroughly ingrained that many textbooks provide the latter as a definition of the former (e.g., [9], p. 14; [16], p. 193)

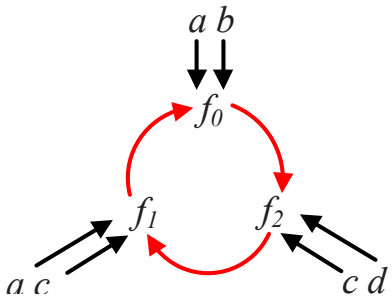
Indeed, any acyclic circuit is clearly combinational. Regardless of the initial values on the wires, once the values of the inputs are fixed, the signals propagate to the outputs. The behavior of a circuit with feedback is generally more complicated. Such a circuit may exhibit sequential behavior, as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

And yet, cyclic circuits can be combinational. Consider the functions shown in Figure 1. Notice that for any assignment of the primary inputs a, b, c, and d, each function f_0, f_1, and f_2 evaluate to a definite Boolean value.

In previous work, we showed that combinational circuits can be optimized significantly if cycles are introduced [12].

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.

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



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

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

$$\begin{aligned} f_0 &= \bar{f}_1 = 0 \\ f_1 &= f_2 = 1 \\ f_2 &= 1 \end{aligned}$$

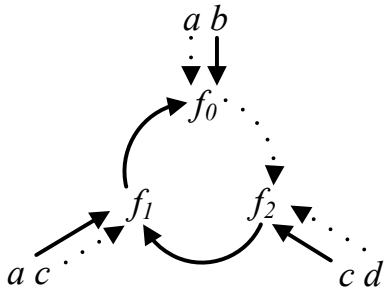


Fig. 2. Network in Figure 1 with $a = 1, b = 0, c = 1, d = 0$.

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

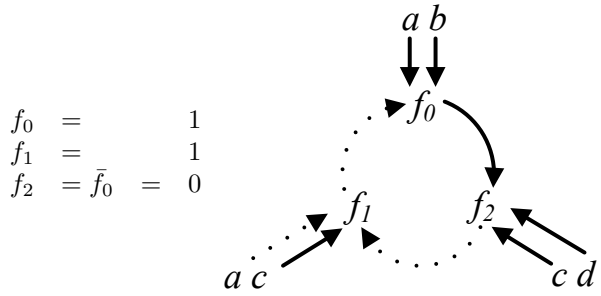


Fig. 3. Network in Figure 1 with $a = 1, b = 1, c = 0, d = 0$.

the level of functional dependencies. In this previous approach we used the *Simplify* command in the Berkely SIS package [14] to compute functional dependencies. This approach is less efficient than the methodology we propose in this work for two reasons. First, the *Simplify* command used SOP manipulations to derive implementations for target functions. This places a structural bias on an implementation which may make the representation appear to be not combinational even though certain manipulations of the SOP representations could yield a combinational result. Second, The algebraic approach to synthesizing dependency functions is not as efficient as newer SAT or BDD based approaches [10], [3]

A. Functional Dependencies

The process of multilevel logic synthesis typically consists of an iterative application of minimization, decomposition, and restructuring operations [2]. An important operation is **substitution** (also sometimes called "resubstitution"), in which node functions are expressed, or re-expressed, in terms of other node functions as well as of their original inputs. The functions in the table in Figure 1 are defined in terms of the primary inputs a, b, c, and d. After substitutions, the functions are redefined in terms of both the primary inputs as well as other primary output functions.

If there are no cycles in the dependencies, any specification that is *consistent* is valid. By consistent we mean that the specification of each node, viewed in isolation, yields the corresponding target function when we apply the other candidate functions at its inputs. With cyclic dependencies, the specification may well be consistent and yet it can be spurious. To wit, suppose we wish to compute some complicated function f and its complement $\bar{f}$. Saying that

$$\begin{aligned} f &= \bar{f}, \\ \bar{f} &= f, \end{aligned}$$

is evidently meaningless.

Indeed, a pivotal step in the synthesis methodology is determining whether cyclic dependencies are valid. Early work on the topic established the theoretical and conceptual framework for analysis: [11], [5], [15], [12], [13]. These papers discussed analysis either in an explicit setting, or else in a symbolic setting based on binary decision diagrams (BDDs) [3]. In recent work, we propose a more efficient technique for analysis based on Boolean satisfiability (SAT) [1]. In this paper, we present a SAT-based method for synthesizing cyclic dependencies, through Craig interpolation [4].

B. Definitions and Notation

We use the standard notation: addition ($+$) denotes disjunction (OR), multiplication ($\cdot$), denotes conjunction (AND), 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. The **composition** operation of a function f with respect to a variable x and a function g,

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

refers to the substitution of g for x in f. 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 may also use the term **depends** in reference to a specific input assignment to a function. When a function depends on some variable for a specific input assignment, this means that toggling the value of the variable while holding the value of the other support set variables constant causes the function to toggle values.

for a function $f(x_0, x_1, \ldots, x_n)$ we use the notation $f(x_0, x_1, \ldots, x_n)^1$ to denote a function's **ON set** (the set of assignments to variables $x_0, x_1, \ldots, x_n$ where f evaluates to 1). Similarly, we use the notation $f(x_0, x_1, \ldots, x_n)^0$ to denote a function's **OFF set** (the set of assignments to variables $x_0, x_1, \ldots, x_n$ where f evaluates to 0).

A **SAT Instance** is a Boolean formula specified in conjunctive normal form (CNF). We will also refer to a circuit with a single primary output as a SAT instance or a *miter*; the satisfiability of the primary output can be represented as a CNF formula.

C. Network Model

Our model is at the level of abstraction applicable in the technology-independent phase of logic synthesis. Our goal is to construct a network that computes Boolean functions of Boolean input variables $x_1, \ldots, x_m$. Internally, the network is specified as a collection of nodes $\mathcal{N}$. Associated with each node is a **node function** f_i and an **internal variable** y_i, $1 \leq i \leq n$. The node functions can depend on input variables as well as on internal variables. In the **dependency graph**, a directed edge is drawn from node i to node j iff the node j is in the support set of node function f_j. The dependency graph for functions in Figure 1 is shown to the right of the truth table.

For a fixed assignment of inputs, call the network the **induced** network, and call the associated dependency graph the **induced** dependency graph. In the induced network, if a node function f_i doesn't depend upon any internal variable (i.e., it evaluates to 0 or 1), then we may substitute this value for the corresponding internal variable y_i in other expressions. In this way, we can continue to simplify the network, until

no further simplifications are possible. Call the result the **simplified induced** network. Figure 2 and Figure 3 show the induced graph for different assignments of the variables a, b, c, and d for the functions defined in Figure 1. In Figure 2, we consider the assignment of the primary input variables $a = 1, b = 0, c = 1, d = 0$. For this assignment, f_2 evaluates to 1 because $c = 1$. f_2 does not depend on d or f_0 for this assignment because f_2 evaluates to 1 regardless of the values of d or f_0. f_1 evaluates to 1 regardless of the assignment of c, and f_0 evaluates to 0 regardless of the value of a. The dependency graph in Figure 3 is generated by this same logic.

D. Definition of Combinationality

A network is **combinational** iff it computes unique Boolean output values for each Boolean input assignment. We sometimes abuse this terminology and say that a network is combinational for a *specific* input assignment, meaning that it computes unique Boolean output values for that input assignment. If there are "don't care" conditions on the inputs, then it is sufficient if the network computes unique Boolean values for input assignments in the "care" set.

This computation must hold:

- regardless of the initial state
- and independently of all timing assumptions.

Proposition 1
A network is combinational iff, for each assignment of Boolean values to the inputs, There is never a cycle in the induced dependency graph

This definition of combinationality is functionally equivalent to that proposed in earlier work [11], [5], [15], [12].

E. Functional Dependencies with Craig Interpolation

A method for synthesizing functional dependencies based on *Craig interpolation* was proposed in [10]. This method scales much better with circuit size than earlier BDD-based methods [3]. We give a brief review of it here. It constructs a miter, as shown Figure 4[1]. 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 the function f_0 for some network. Here f_0 is known as the **target function**.

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. In other words, $f_i(X)$ is equivalent to f_i*(X*) where the assignment of X is equal to the assignment of X*.

[1]The miter that we use differs slightly from the one proposed in [10]. Rather than asserting f_0 to be on and f_0* to be off, we assert the Exclusive-OR of the two functions. Because of the symmetry between the circuits $f_0\ Left$ and $f_0\ Right$, for every assignment of the primary inputs X and X* that cause f_0 to evaluate to 1 and f_0* to evaluate to 0, there will be another assignment of X and X* that cause f_0 to evaluate to 0 and f_0* to evaluate to 1

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 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 $f_0(X)$.

[10] discusses a method using Craig interpolation to find the dependency function h. 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 in interpolant from a SAT instance that is similar to that in Figure 4.

II. Illustration of Algorithm

Previous approaches separated the phases of synthesis and analysis: functional dependencies were first generated; then, if they were cyclic, they were validated. In this paper, we propose a SAT-based approach that does both simultaneously: it verifies whether it is possible to generate target functions with specified support sets and, at the same time, whether or not the selected cyclic representation is valid.

In order to assert that a cyclic dependency is valid, we need to add additional logic to the SAT instance in Figure 4. This logic will cause the SAT instance to be satisfied if the conditions stated in Proposition 1 are not met. In order to assert the conditions of Proposition 1, we must assert that for every assignment of the primary inputs to a given network, the network's induced dependency graph is acyclic.

Let us again choose a function $f(X)$ and its equivalent function f*(X*). We can check to see if f is dependent on variable x_i for some assignment to X by checking the satisfiability of the following clauses:

$$\begin{array}{l}(f \not\equiv f*)\\(x_0 \equiv x_0*)(x_1 \equiv x_1*)\cdots(x_{i-1} \equiv x_{i-1}*)\\(x_{i+1} \equiv x_{i+1}*)\cdots(x_n \equiv x_n*)\end{array} \quad (1)$$

Equ. 1: Clauses that evaluate to true if and only if function f (f*) is dependent on variable x_i (x_i*) for some assignment of X (X*)

These clauses are satisfied when f assumes a different value from f* and every variable in the support set of f assumes the same value as its corresponding variable in f* except for variable x_i. Clearly if the value of f (f*) toggles with the value of x_i (x_i*), then f (f*) is dependent on x_i (x_i*) for that assignment of X (X*).

Assuming we are given the cyclic dependency graph for a network we can add the logic of Equation 1 to the SAT

instance in Figure 4 to determine if the target functions behave combinationally for the given dependency graph.

In Figure 5, we check to see if the dependency graph in Figure 1 can be implemented for some network. Here gates g_1, g_2, and g_3 check to see if possible dependency functions exist. These gates, and the logic that feeds them, are created by the same intiuition as that gate g_1 in Figure 4. In Figure 5, gate g_1 checks for the existence of a dependency function for f_0 in terms of f_1, a, and b. Similarly, g_2 checks for the existence of a dependency function for f_2 in terms of f_0, c, and d. Gate g_3 checks for the existence of a dependency function for f_1 in terms of f_2, a, and c. Clearly if such dependency functions exist for f_0, f_1, and f_2, then gates g_1, g_2, and g_3 will be identically 0.

We assert that Proposition 1 holds true for this network by adding gate g_4. If, for some assignment of the primary inputs of this network, the induced graph contains a cycle then g_4 will evaluate to 1. Notice in Figure 1 that there is a cyclic dependency, indicated by the red arrows. If for every assignment of a, b, c, and d at least one of these red dependency arrows disappears in the induced graph, then the network is combinational. For this specific network we must check to see if f_0 depends on f_1 while f_1 depends on f_2 and f_2 depends on f_0 for some assignment of the primary inputs.

Again we can check this condition by adding the logic of Equation 1 to the SAT instance proposed in Figure 4. Notice that the support set of f_0 is f_1, a, b. Therefore if f_0 depends on f_1 for some assignment of a, b, c, and d then the following clauses will be satisfied:

$$(f_0 \not\equiv f_0*)(a \equiv a*)(b \equiv b*)$$

Likewise, the support set of f_1 is f_2, a, c. Equation 1 tells us that the following clauses are satisfied when f_1 depends on f_2:

$$(f_1 \not\equiv f_1*)(a \equiv a*)(c \equiv c*)$$

Finally, the support set of f_2 is f_0, c, d. Equation 1 tells us that the following clauses are satisfied when f_2 depends on f_0:

$$(f_2 \not\equiv f_2*)(c \equiv c*)(d \equiv d*)$$

In Figure 5 these clauses are the logic of g_4. If g_4 evaluates to 1 for some assignment of a, b, c, d, a*, b*, c*, and d* then the target function for f_0 is dependent on f_1 because f_1 toggles the value of f_0. Furthermore, f_1 must also toggle with the value of f_2 while the value of f_2 is toggled by the value of f_0. If some assignment of the primary inputs causes this mutually dependent behavior, then the network is not behaving combinationally.

If g_1, g_2 and g_3, are shown to be identically 0 then we know the target functions $f_0(f_1,a,b)$, $f_1(f_2,a,c)$, and $f_2(f_0,c,d)$ exist. If g_4 is identically 0, then we know that no induced graph of the network contains any cycles; therefore the representation behaves combinationally.

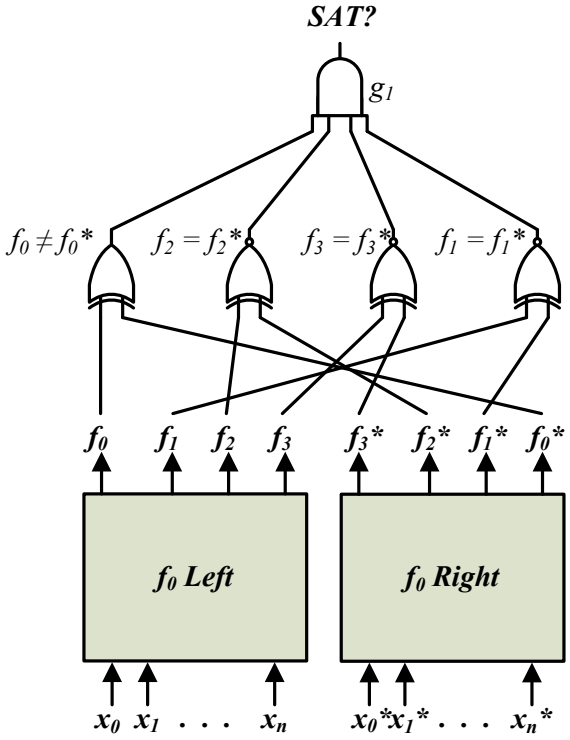


Fig. 4. A miter that checks to see if f_0 can be specified in terms of f_1, f_2, and f_3.

III. General Method

We sketch the steps of the algorithm for the general case of any sort of dependency graph.

1) Choose a dependency graph for target functions (say with a branch and bound approach [12].
2) Locate all of the cycles in the dependency graph.
3) For each target function, create a SAT instance that asserts there is a dependency function for the target function with the given support set (gates g_1, g_2, and g_3 in Figure 5)
4) For each dependency in each cycle, create a SAT instance that asserts that the dependency holds for some assignment of the network's primary inputs (gate g_4 in Figure 5).
5) Create the logical OR of the primary outputs of the circuits produced from Steps 3 and 4. Check the satisfiability of this OR gate.
6) If the OR gate is unsatisfiable, cache the solution. Continue the search with other possible dependency graphs until a good solution is found (branching and bounding).

We can abstract the structure of the SAT instance in Figure 4 to show how we can verify that there is a dependency function for any specified support set for any target function. Given a network with target functions f_0, f_1, …, f_n and primary inputs x_0, x_1, …, x_n, say that we want to see if f_0 can be represented with a support set of f_1, f_2, … f_n. Clearly such a dependency function exists for f_0 if the values of f_1, f_2, …, f_n when f_0 = 0 are not equivalent to the values of f_1, f_2, …, f_n when f_0 = 1. We can check to see if the ON set $f_0(f_1,f_2, \ldots, f_n)^1$ is disjoint from the OFF set $f_0(f_1,f_2, \ldots,$

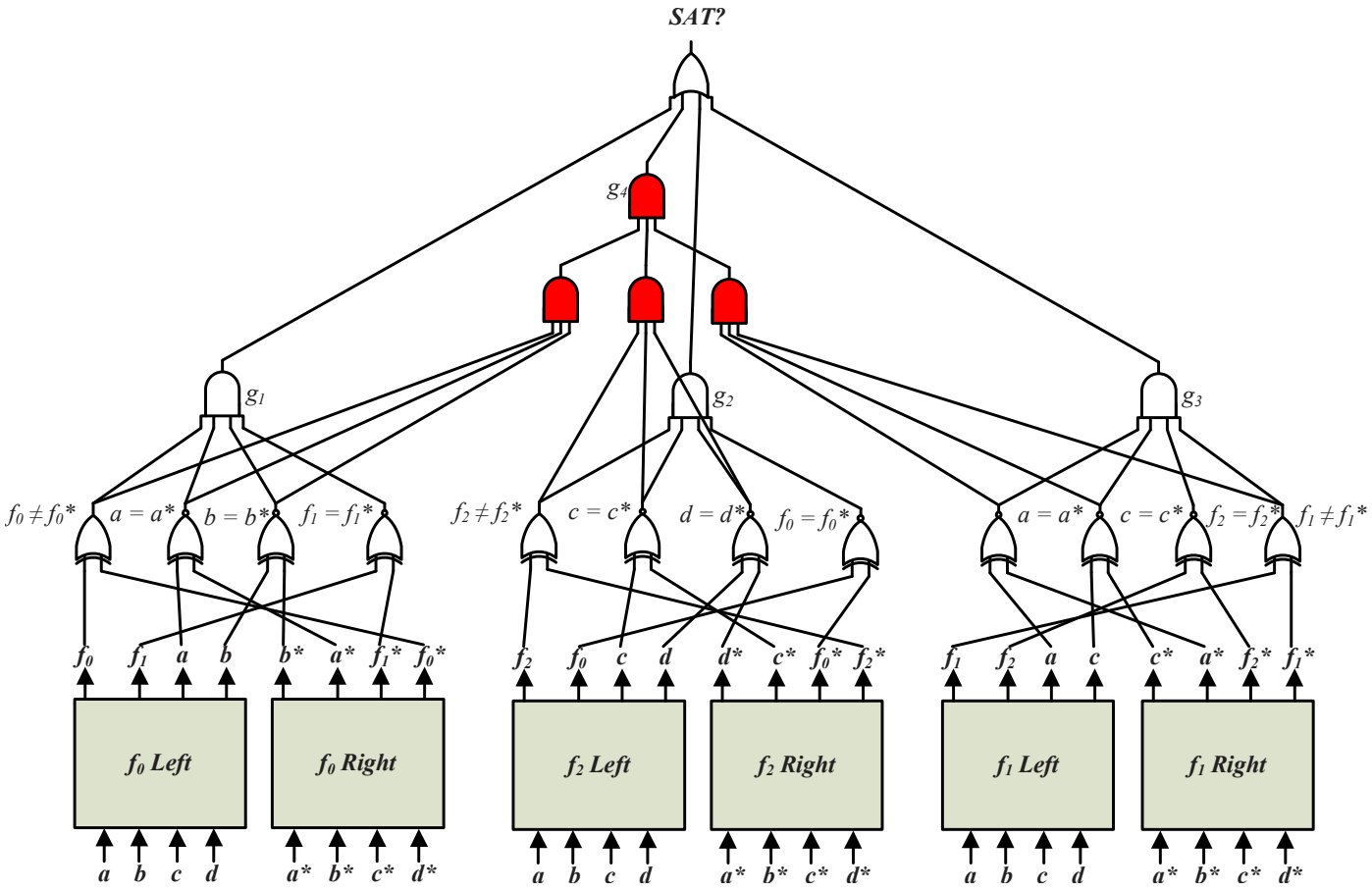


Fig. 5. A miter that checks to see if $f_0(f_1,a,b)$, $f_1(f_2,a,c)$, and $f_2(f_0,c,d)$ can be implemented simultaneously.

$f_n)^0$ by the same logic as in the example in Figure 4. We create two copies of the network, referred to as f_0 *Left* and f_0 *Right*. Let the primary inputs for f_0 *Left* be x_0, x_1, …, x_n and the primary inputs for f_0 *Right* be x_0*, x_1*, …, x_n*. Let the candidate functions of f_0 *Left* be f_0, f_1, …, f_n and the candidate functions of f_0 *Right* be f_0*, f_1*, …, f_n*.

Consider the assertions:

$$(f_1 \equiv f_1*)(f_2 \equiv f_2*)\cdots(f_n \equiv f_n*)(f_0 \not\equiv f_0*).$$

Call these "A". If these assertions are satisfied this indicates that there is some assignment of x_0, x_1, …, x_n and x_0*, x_1*, …, x_n* such that every variable in the support set of f_0 is equal to its corresponding variable in the support set of f_0*. At the same time, f_0 assumes a different value from f_0*. Clearly the ON set and OFF set of the target function f_0 are disjoint for the given support set and therefore a dependency function $h(f_1,f_0,\ldots,f_n)$ that implements f_0 does not exist.

If these assertions are never satisfied then we know that the ON set and OFF set of f_0 are disjoint. Therefore there must be a dependency function $h(f_1,f_0,\ldots,f_n)$ that implements f_0. Further explanation of this approach is given in [10].

To verify that any sort of dependency graph for a network behaves combinationally we simply add the logic of Equation 1 to each dependency in each cycle in the dependency graph.

From Proposition 1 we know that every induced dependency graph must be acyclic for a network to behave combinationally. We also know that the logic in Equation 1 asserts that a function f depends on some variable x_i in the support set of f. If the logical AND of all the clauses in Equation 1 for every dependency in some cycle in the dependency graph of a network is satisfied, this indicates that the induced dependency graph of the network contains a cycle for some primary input assignment. Therefore the specified dependency graph does not behave combinationally. Likewise, if these clauses are never satisfied for any cycle for any assignment to the primary inputs of the network, then the induced dependency graph is always acyclic and therefore combinational. Call the logical AND of the clauses in Equation 1 for every dependency in some cycle "B".

By this line of reasoning, if all conditions A and all conditions B (for every cycle) are false for every assignment to the primary inputs of the network, then the selected dependency graph must be a functionally equivalent, combinational representation for the given network.

IV. Implementation and Results

This algorithm was implemented using the AIG package in Berkley ABC [6]. The SAT solver used was MiniSAT [7]. All the trials were run on a 32-bit Linux machine with a 3 GHz AMD Athlon 64 X2 Dual Core 6000+ CPU. Only one core was utilized for running the algorithm.

Total Support Set Size					
Circuit	Number of PIs	Number of POs	Support Set Size	Number of Cycles	Runtime(s)
5xp1	7	10	43	1	.01
bbsse	11	11	86	6	.01
clip	9	5	49	1	.03
cse	11	11	89	1	.02
dk16	7	8	67	1	.02
inc	7	9	57	4	.01
s1	13	11	150	2	.03
s298	11	14	138	3	.60
sse	11	11	86	6	.01
styr	14	15	167	22	.11
table3	14	14	220	3	.24

TABLE I
BENCHMARK CIRCUITS WITH CYCLIC DEPENDENCIES PRODUCED BY CYCLIFY AND VERIFIED BY OUR NEW ALGORITHM

The cyclic dependencies for these benchmark circuits were produced with our tool CYCLIFY and then verified with this new algorithm. The time that is reported is the time that it takes to generate the SAT instance given the target functions and the support sets and the time it takes for the SAT instance to be solved. The time that it took to solve the SAT instance almost entirely dominated the overall runtime. The table does not include the time that it takes to generate interpolants for the target functions for the verified representation as this would be a final step in a synthesis process. The results show the size of the support set for all primary outputs summed together. The number of cycles that are listed are the number of cycles that exist in the circuit on a *functional level* (if the design were mapped to some technology many other *physical* cycles may exist in the topology of the circuit). The cycles in the dependency graphs were discovered using the algorithm described in [17].

The runtime for the algorithm seems to be correlated best with the size of the support set for the given dependency graph. We have not yet incorporated our methodology into a full synthesis routine, but we plan to use this methodology in conjunction with a branch and bound method similar to the approach proposed in [12]. In this implementation the best "cost" metric may likely be the size of the support set for the selected target functions of the implementation. For example, choosing to compute target functions with a support set size of k would likely be easier to map to a technology that uses k-inputs gates

V. DISCUSSION

Early work in the 1960's and 70's established the premise of combinational circuits with cycles, and suggested the possible benefits. Still, combinational circuits are not designed with cycles in practice. Perhaps designers have eschewed feedback due to the apparent complexity of reasoning about cyclic structures. And yet, feedback provides significant opportunities for optimization, both for area and for delay. Indeed, contrary to the conventional wisdom, cyclic solutions are not a rarity; they can readily be found for most circuits that are not trivially simple or sparse. We have run trials with our program called CYCLIFY, on a range of randomly generated examples and benchmark circuits. We note that solutions for most of the examples have deeply nested loops, with dozens or even hundreds of cycles. The current practice for state of the art synthesis tools is to reject solutions that have cycles in them. It is argued that cyclic solutions are to hard to analyze for them to be useful. In previous work we have proposed a method for timing analysis [13]. It has been pointed out that post silicon testing may proove to be a difficult task for cyclic circuits, as fault fabrication of cyclic circuits may cause them to behave sequentially [11]. Indeed testing seems to remain an open problem for cyclic circuits.

In future work, we will incorporate the strategy discussed here into a full synthesis methodology, based on a branch-and-bound search on the space of possible cyclic dependencies. An important point is ensure *gate-level* correctness when cyclic dependencies at the functional level are mapped, as discussed in [8]. Indeed, technology mapping sometimes invalidates cyclic solutions that were valid at the functional level. This issue can be overcome by simply verifying correctness at every stage, and rejecting solutions that are invalid. Any solution that is combinational on a functional level can be mapped to a gate level solution that is also combinational. Indeed, the algorithm that we suggested in [1] performs gate-level SAT-based validation of cyclic circuits. A more elegant solution that we are pursuing is to perform *cycle-aware* technology mapping.

REFERENCES

[1] J. Backes, B. Fett, and M. Riedel. The analysis of cyclic circuits with boolean satisfiability. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM international conference on Computer-aided design*, San Jose, CA, USA, Nov. 2008.

[2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L.Sangiovanni-Vincentelli. Multilevel logic synthesis. In *Proceedings of the IEEE*, number 2, pages 264–300, Philadelphia, PA, USA, 1990.

[3] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, pages 677–691, August 1986.

[4] W. Craig. Linear reasoning: A new form of the herbrand-gentzen theorem. *Symbolic Logic*, 22:250–268, 1957.

[5] Stephen A. Edwards. Making cyclic circuits acyclic. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 159–162, New York, NY, USA, 2003. ACM.

[6] A. Mishchenko et al. ABC: A system for sequential synthesis and verification.

[7] N. Sörensson et al. Minisat v1.13 - a sat solver with conflict-clause minimization. Technical report.

[8] J.-H. R. Jiang, A. Mishchenko, and R. K. Brayton. On breakable cyclic definitions. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 411–418, Washington, DC, USA, 2004. IEEE Computer Society.

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

[10] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental sat solving. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 227–233, 2007.

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

[12] M. Riedel and J. Bruck. The synthesis of cyclic combinational circuits. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 163–168, 2003.

[13] Marc D. Riedel and Jehoshua Bruck. Timing analysis of cyclic combinational circuits. In *IWLS '04: Int'l Workshop Logic and Synthesis*, 2004.

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

[15] Thomas Robert Shiple. *Formal analysis of synchronous circuits*. PhD thesis, 1996. Chair-Alberto Sangiovanni-Vincentelli.

[16] John F. Wakerly. *Digital Design: Principles and Practices*. Prentice-Hall, Inc., 2000.

[17] Herbert Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *J. ACM*, 19(1):43–56, 1972.